INFORMIX–4GL

SQL-BASED
APPLICATION DEVELOPMENT LANGUAGE

USER GUIDE

Copyright © 1987
Informix Software, Inc.
June 1987

Part No. 200-501-0001-0
Rev. A
PRINTED IN U.S.A.

INFORMIX–4GL
Version 1.10

THE INFORMIX SOFTWARE AND USER MANUAL ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE INFORMIX SOFTWARE AND USER MANUAL IS WITH YOU. SHOULD THE INFORMIX SOFTWARE AND USER MANUAL PROVE DEFECTIVE, YOU (AND NOT INFORMIX OR ANY AUTHORIZED REPRESENTATIVE OF INFORMIX) ASSUME THE ENTIRE COST OF ALL NECESSARY SERVICING, REPAIR, OR CORRECTION. IN NO EVENT WILL INFORMIX BE LIABLE TO YOU FOR ANY DAMAGES, INCLUDING ANY LOST PROFITS, LOST SAVINGS, OR OTHER INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF OR INABILITY TO USE SUCH INFORMIX SOFTWARE OR USER MANUAL, EVEN IF INFORMIX OR AN AUTHORIZED REPRESENTATIVE OF INFORMIX HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES, OR FOR ANY CLAIM BY ANY OTHER PARTY. IN ADDITION, INFORMIX SHALL NOT BE LIABLE FOR ANY CLAIM ARISING OUT OF THE USE OF OR INABILITY TO USE SUCH INFORMIX SOFTWARE OR USER MANUAL BASED UPON STRICT LIABILITY OR INFORMIX'S NEGLIGENCE. SOME STATES DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO THE ABOVE EXCLUSION MAY NOT APPLY TO YOU. THIS WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS AND YOU MAY ALSO HAVE OTHER RIGHTS, WHICH VARY FROM STATE TO STATE.

All rights reserved. No part of this work covered by the copyright hereon may be reproduced or used in any form or by any means—graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems—without permission of the publisher.

Published by:
Informix Software, Inc.
4100 Bohannon Drive
Menlo Park, CA 94025

INFORMIX is a registered trademark of Informix Software, Inc. **RDSQL**, **C-ISAM**, and **File-it!** are trademarks of Informix Software, Inc.

UNIX is a trademark of AT&T.
MS is a trademark of Microsoft Corporation.

RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the 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 (and any other applicable license provisions set forth in the Government contract).

Copyright © 1981-1987 by Informix Software, Inc.

NOTE: ANY REFERENCES IN DOCUMENTATION TO "RELATIONAL DATABASE, INC." OR "RDS" SHALL MEAN INFORMIX SOFTWARE, INC.

Table of Contents

Preface

About This Guide 5
Chapter Summary 5
Operating System References 7
Conventions Used in This Guide 8
Related Reading 10
Preparing to Use INFORMIX-4GL 11
Setting Up the Operating Environment 11
Getting Started with INFORMIX-4GL on DOS Systems 16
Using Your Terminal 17
The Demonstration Database 19
Restricted and Public Databases 21

Chapter 1. Introduction

Chapter Overview 1-5
Introducing INFORMIX-4GL 1-6
What Are Fourth-Generation Languages? 1-6
Features of INFORMIX-4GL 1-8
A Database Language 1-9
A Programming Language 1-10
A Screen-Building Utility 1-11
A Menu-Building Utility 1-15
A Report Writer 1-16
A Window Manager 1-18
The INFORMIX-4GL Programmer's Environment 1-22
A Creative Solution for Database Applications 1-22

Chapter 2. Creating a Database

Chapter Overview 2-5
Basic Database Concepts 2-6
What is a Database? 2-6
Database Tables 2-7
Rows and Columns 2-8
Joining Tables 2-10
Indexes 2-17
Creating a Database 2-18
The MAIN Statement 2-18
The CREATE DATABASE Statement 2-19
The CREATE TABLE Statement 2-20
Creating an Index 2-31
The CREATE INDEX Statement 2-33
The Programmers Environment 2-37
Entering the Programmer's Environment 2-37
A Program That Creates a Database 2-39
Chapter Summary 2-48

Chapter 3. Working with a Database

Chapter Overview 3-5
Selecting a Database: The DATABASE Statement 3-6
Defining Program Variables 3-7
The DEFINE Statement 3-7
Sample DEFINE Statements 3-10
Assignment Statements: LET 3-12
Expressions 3-12
Interactive Programs 3-21
Getting Input from the User 3-21
Displaying Output: the DISPLAY Statement 3-23
Basic Database Operations 3-25
Inserting Rows into a Table 3-25
Selecting Rows from a Table 3-28
Updating Rows in a Table 3-35
Deleting Rows from a Table 3-38
Chapter Summary 3-40

Chapter 4. The SELECT Statement

Chapter Overview 4-5
Defining Records 4-6
Using a Record in a SELECT Statement 4-7
SELECT Statements That Retrieve More than One Row 4-10
Assigning a Cursor to a SELECT Statement 4-10
Retrieving and Processing Rows 4-11
Complex SELECT Statements 4-27
More About the SELECT Clause 4-27
More About the WHERE Clause 4-29
The ORDER BY Clause 4-34
Selecting Data from Several Tables 4-36
Chapter Summary 4-39

Chapter 5. Functions

Chapter Overview 5-5
Writing Functions 5-6
The FUNCTION Statement 5-6
Calling Functions 5-7
Passing Values Between the Main Program and a Function 5-8
Global and Local Variables 5-9
Parameters 5-16
Using Parameters in the FUNCTION Statement 5-16
Returning Values from a Function to the Main Program 5-21
Library Functions 5-25
Chapter Summary 5-26

Chapter 6. Designing Screen Forms

Chapter Overview 6-5
Using Screen Forms 6-6
The Form Specification File 6-9
Sections of a Form Specification File 6-12
Parts of the Form Specification File 6-13

The DATABASE Section 6-13
The SCREEN Section 6-14
TABLES Section 6-17
ATTRIBUTES Section 6-18
The INSTRUCTIONS Section 6-31
The Programmers Environment 6-35
Creating a Default Form Specification File 6-36
Modifying a Form Specification File 6-39
Creating a Form Specification File with a Text Editor 6-43
Chapter Summary 6-46

Chapter 7. Working with Screen Forms

Chapter Overview 7-5
Working with Screen Forms 7-6
Displaying a Screen Form 7-6
Prompting for Input 7-8
Writing Interactive Programs that Use a Screen Form 7-12
The INPUT Statement 7-12
The DISPLAY Statement 7-18
Erasing Values in Screen Fields: the CLEAR Statement 7-20
More About INPUT: the WITHOUT DEFAULTS Clause 7-20
The SQLCA Record 7-22
Programs for Entering Data 7-23
Programs for Querying the customer Table 7-27
A Program for Updating Rows in the customer Table 7-32
A Program for Deleting Rows from the customer Table 7-35
Chapter Summary 7-38

Chapter 8. Menus, Options, and Help

Chapter Overview 8-5
Creating Menus with INFORMIX-4GL 8-6
The Appearance of a Menu 8-6

Selecting Menu Options 8-8
Requesting Help 8-9
Creating a Menu 8-9
The MENU Statement 8-10
Changing the Message and Prompt Lines 8-18
The OPTIONS Statement 8-18
Creating Help Messages 8-21
Requesting Online Help 8-21
Creating a Help File 8-23
The mkmessage Utility 8-25
Using the Help File with a Program 8-25
A Sample Menu Program 8-29
The GLOBALS Statement 8-35
The MAIN Statement 8-35
The show_menu Function 8-36
The enter_row Function 8-38
The query_data Function 8-39
The change_data Function 8-40
The delete_row Function 8-41
Nested Menus 8-42
The show_menu Function 8-44
The query_data Function 8-45
The view_cust Function 8-46
Chapter Summary 8-48

Chapter 9. Designing Reports

Chapter Overview 9-5
Designing Reports 9-6
How to Create a Report 9-7
The REPORT Statement 9-8
Selecting Data to Include in the Report 9-11
Running a Report 9-12
Basic Report Formatting 9-14
Page Headers and Trailers 9-16
Printing Rows of Data 9-20
Grouping Data 9-22
Calculations in a Report 9-27
Arithmetic Operators 9-27
Controlling the Appearance of Numbers 9-29

Math Functions 9-32
Group Functions 9-36
Displaying a Report 9-39
Printing the Report on a Printer 9-39
Writing the Report to a File 9-40
Piping the Report to Another Program 9-40
Running ACE Reports Using INFORMIX-4GL 9-41
Chapter Summary 9-42

Chapter 10. Handling Errors and Interrupts from the User

Chapter Overview 10-5
Anticipating Errors 10-6
Determining Whether an Error Occurred: the status Variable 10-6
Trapping Errors: the WHENEVER Statement 10-7
A Sample Program for Handling Errors 10-12
Handling Inappropriate Input 10-13
Handling Interrupts from the User 10-16
The INTERRUPT and QUIT Keys 10-16
The DEFER Statement 10-16
An Alternative to the INTERRUPT Key 10-23
Chapter Summary 10-30

Chapter 11. Using Multiple-Table Forms and Screen Arrays

Chapter Overview 11-5
Working with Multiple-Table Forms 11-6
The order Form 11-6
Defining Program Arrays 11-12
Working with Arrays: the FOR Statement 11-15
Using a Multiple-Table Form in a Program 11-19
The INPUT ARRAY Statement 11-19
Scrolling and Editing 11-21
Built-in Functions for Program Arrays and Screen Arrays 11-23
Using INPUT ARRAY with Optional Clauses 11-25

The DISPLAY ARRAY Statement 11-51
The ordmenu Program 11-52
Chapter Summary 11-69

Chapter 12. Screen Forms: Query by Example

Chapter Overview 12-5
Query by Example 12-6
Specifying Search Criteria 12-7
Searching with Relational Operators 12-8
Using Wildcard Characters 12-10
Searching with the Alternation Operator 12-11
Searching for a Range of Values 12-11
Query Operators and Short Fields 12-11
Entering Search Criteria in SMALLFLOAT and FLOAT Fields 12-12
Constructing a SELECT Statement from Search Criteria 12-13
The CONSTRUCT Statement 12-14
Constructing a SELECT Statement 12-16
The PREPARE Statement 12-18
Executing a Prepared Statement 12-19
Sample Programs for Performing Queries by Example 12-20
Chapter Summary 12-28

Chapter 13. Windows

Chapter Overview 13-5
Window Management 13-6
Demonstration Programs Containing Windows 13-9
Using the Order Entry Program 13-10
Including a Window in a Program 13-12
Naming a Window 13-12
Establishing the Location of a Window 13-13
Specifying the Size of a Window 13-14
Indicating Window Attributes 13-19
Closing a Window 13-26
Displaying Information in a Window 13-27

Displaying a Message 13-27
Prompting for Input 13-29
Displaying a Form in a Window 13-30
Displaying a Menu in a Window 13-31
Clearing a Window of Text 13-33
Examining the Order Entry Program 13-35
The wind1.4gl Program 13-36
The enter_order Function 13-44
The get_cust Function 13-46
The show_cust Function 13-47
The stock_help Function 13-51
The get_stock Function 13-51
Using a Program with Multiple Windows 13-56
Working with Multiple Windows 13-60
Indicating the Current Window 13-60
More About the CLEAR WINDOW Statement 13-63
More About the CLOSE WINDOW Statement 13-65
Examining a Program with Multiple Windows 13-68
The wind2.4gl Program 13-69
The GLOBALS Statement 13-75
The MAIN Statement 13-75
The show_menu Function 13-76
The set_curs Function 13-77
The get_cust Function 13-79
The view_cust Function 13-79
The get_ord Function 13-80
The view_ord Function 13-80
The det_cust Function 13-81
The det_ord Function 13-82
The change_wind Function 13-83
The clear_menu Function 13-83
The mess Function 13-83
Chapter Summary 13-84

Chapter 14. Learning More About INFORMIX-4GL

Chapter Overview 14-5
Access Privileges 14-6
Granting and Revoking Database Privileges 14-6
Granting and Revoking Table Privileges 14-8

Modifying a Database 14-9
Changing the Structure of a Database 14-9
Changing the Structure of a Table 14-12
Transactions 14-15
Creating a Database with Transactions 14-15
Specifying Transactions 14-16
Starting a New Transaction Log 14-17
Recovering a Database 14-17
Audit Trails 14-18
Comparing Audit Trails and the Transaction Log 14-18
Views 14-19
Creating and Dropping Views 14-19
Input and Display Attributes 14-20
The upscol Utility 14-21
Chapter Summary 14-22

Appendix A. The stores Database

Appendix B. Environment Variables

Appendix C. Reserved Words

Appendix D. Sample Reports

Index

Preface

Preface Table of Contents

About This Guide 5
Chapter Summary 5
Operating System References 7
Conventions Used in This Guide 8
Syntax 8
Related Reading 10

Preparing to Use INFORMIX-4GL 11
Setting Up the Operating Environment 11
Setting Environment Variables on UNIX Systems 12
Setting Environment Variables on DOS Systems 14
Getting Started with INFORMIX-4GL on DOS Systems 16
Loading the Database Agent 16
Entering INFORMIX-4GL 17
Using Your Terminal 17
The Demonstration Database 19
Creating the Demonstration Database 19
Restoring the Database 20
Restricted and Public Databases 21

About This Guide

The *INFORMIX-4GL User Guide* and its companion volumes, the *INFORMIX-4GL Reference Manual*, volumes one and two, comprise the comprehensive documentation for the **INFORMIX-4GL** SQL-based application development language.

This manual includes three additional documents:

- *Getting Started: A Guide to Products and Services* contains customer registration and warranty information, along with descriptions of other RDS products.

- *INFORMIX-4GL: A Twenty-Minute Guide* briefly describes the important features of **INFORMIX-4GL** and includes a sample program that shows how easily you can produce sophisticated applications with **INFORMIX-4GL**.

- The *INFORMIX-4GL Quick Reference* card displays syntax statements for the **INFORMIX-4GL** programming language (including the **RDSQL** query language), and form and report routines.

Chapter Summary

Each chapter in this book begins with a section describing chapter contents. This section includes a paragraph to help you identify topics appropriate for your needs. The remaining part of the chapter explains a topic area and demonstrates its use in **INFORMIX-4GL**. A concluding summary highlights the most important points of the chapter.

The *INFORMIX-4GL User Guide* includes the following chapters and appendixes:

Chapter 1	introduces **INFORMIX-4GL** and explains what fourth-generation languages are.

Chapter 2	explains basic database concepts and describes how to create databases, tables, and indexes using **INFORMIX-4GL**. It also explains how to create and compile programs in the **INFORMIX-4GL** Programmer's Environment.
Chapter 3	describes how to write **INFORMIX-4GL** programs that insert, select, update, and delete information in a database.
Chapter 4	explains how to retrieve groups of rows from a database using the SELECT statement with cursor management statements.
Chapter 5	explains how to create subroutines using the FUNCTION statement. It also describes how to pass information between routines using global variables and parameters.
Chapter 6	explains how to create screen forms in the **INFORMIX-4GL** Programmer's Environment.
Chapter 7	explains how to write programs that allow the user to add, retrieve, modify, and delete data through a screen form.
Chapter 8	explains how to use the MENU, OPTIONS, and HELP statements to create a menu-driven program.
Chapter 9	explains how to use **INFORMIX-4GL** statements to create custom-formatted reports.
Chapter 10	describes the error-handling facilities provided by **INFORMIX-4GL**.
Chapter 11	explains how to create a multiple-table screen form that includes a screen array. It also explains how to write programs that allow the user to process data through the screen form.

Chapter 12	explains how to write INFORMIX-4GL programs that allow the user to perform queries by example.
Chapter 13	describes the window management statements available on INFORMIX-4GL and examines two programs that use windows.
Chapter 14	briefly describes important INFORMIX-4GL topics that were not covered in earlier chapters, including access privileges, modifying the structure of the database, transactions, audit trails, views, and the **upscol** utility.
Appendix A	discusses the demonstration database that is included with the INFORMIX-4GL software.
Appendix B	explains the environment variables available with INFORMIX-4GL.
Appendix C	lists the INFORMIX-4GL reserved words.
Appendix D	shows programs that produce the sample reports in Chapter 9.

Operating System References

INFORMIX-4GL is available for both UNIX and DOS (PC-DOS and MS-DOS) operating systems. The few differences in performance and screen display between the two systems are noted whenever they occur.

For ease of reading, PC-DOS and MS-DOS both are referred to as "DOS" systems.

Conventions Used in This Guide

The *INFORMIX-4GL User Guide* and the *INFORMIX-4GL Reference Manual* use a standard set of conventions to introduce new terms, show screen displays, describe command syntax, and so forth.

When new terms are introduced, they are printed in italics, *like this*. The documentation also includes illustrations that show what you see on the screen as you use **INFORMIX-4GL**. Information that **INFORMIX-4GL** displays and information that you enter are printed in a computer typeface, like this.

Syntax statements describe the format of **INFORMIX-4GL** statements or commands, including alternate forms of a statement, required and optional parts of the statement, and so forth. Syntax statements have their own rules; these are defined in detail in the following section. Briefly, all *keywords* are shown in uppercase letters for ease of identification, even though you need not enter them that way. Words for which you must supply values are in italics. When keywords or values are enclosed in square brackets ([ ]), they are optional parts of the statement (unless otherwise indicated).

Syntax

The following notational conventions are used in syntax statements that appear in this book, as well as in the *INFORMIX-4GL Reference Manual*:

ABC — Any term in the syntax in uppercase letters is a keyword. Enter it exactly as shown, disregarding case. For example,

CREATE TABLE *table-name*

means you must enter the keywords CREATE TABLE or create table.

abc Substitute a value for any term that appears in lowercase italic letters. In the previous example, you should substitute a value for *table-name*.

( ) Enter parentheses as shown. They are part of the syntax of a statement and are not special symbols.

[ ] Do not enter brackets as part of a statement; they surround any part of a statement that is optional. The statement

```
CREATE [UNIQUE] INDEX
```

indicates that you may enter either create index or create unique index.

| The vertical bar indicates a choice among several options. For example,

```
[ONE | TWO [THREE] | FOUR]
```

means that you can enter either one or two or four and that, if you enter two, you may also enter three. (Since the choices in this example are surrounded by square brackets, you can also omit them entirely.)

{ } When you must choose one of several options, the options are enclosed in braces and are separated by vertical bars.

```
{ONE | TWO | THREE}
```

means that you must enter one or two or three and that you may not enter more than one of them.

<u>ABC</u> When one of several options is the default, it appears underlined.

```
[CHOCOLATE | VANILLA | STRAWBERRY]
```

means that you may select any of the three options, but that if you do not enter one of them, VANILLA is the default.

...

Enter additional items like those preceding the ellipsis, if you want. The ellipsis indicates that the immediately preceding item may be repeated indefinitely. For example,

statement
...

means that series of statements can follow the one that is listed.

Related Reading

Users who want to learn more about efficient techniques for programming in fourth-generation languages may want to refer to the book *Building Applications Using a 4GL* (Sobell Associates, 1986) by Mark G. Sobell. The sample application described in this book is written in **INFORMIX-4GL**.

Users who have had no prior experience with database management may want to refer to an introductory text like C. J. Date's *Database: A Primer* (Addison-Wesley Publishing, 1983).

Readers desiring more technical information may want to consult *An Introduction to Database Systems, Volume I* (Addison-Wesley Publishing, 1981) and *An Introduction to Database Systems, Volume II* (Addison-Wesley Publishing, 1983), also by C. J. Date.

This manual assumes you are familiar with your computer operating system. Readers with little operating system experience may want to look at their operating system manual or a good introductory text before starting to learn about **INFORMIX-4GL**.

Suggested texts on UNIX are *A Practical Guide to the UNIX System* by M. Sobell (Benjamin/Cummings Publishing, 1984), *A Practical Guide to UNIX System V* by M. Sobell (Benjamin/Cummings Publishing, 1985), and *UNIX for People* by Birns, Brown, and Muster (Prentice-Hall, 1985).

The documentation that comes with all DOS systems is a useful source of information about that operating system.

Preparing to Use INFORMIX-4GL

This section describes the steps you must follow before running INFORMIX-4GL, summarizes the conventions for using your terminal with the program, and explains how to use the demonstration database provided as part of INFORMIX-4GL. It also explains how INFORMIX-4GL uses operating system file and directory permissions to control access to databases, and the ways in which operating system constraints may affect the performance of the program.

This section assumes that INFORMIX-4GL is installed on your computer according to the installation instructions in the letter that comes with this software package.

Setting Up the Operating Environment

Several environment variables effect how INFORMIX-4GL acts on your system. These variables are not identical across DOS and UNIX systems. Each system is addressed separately in the following two sections.

Setting Environment Variables on UNIX Systems

Before you can use INFORMIX-4GL, you must follow these steps:

- Set the INFORMIXDIR environment variable so that INFORMIX-4GL can locate its programs.
- Set the TERM environment variable so that INFORMIX-4GL recognizes the kind of terminal you are using.
- Set the TERMCAP environment variable so INFORMIX-4GL can communicate with your terminal.
- Set the PATH environment variable so that the shell searches the correct bin for executable INFORMIX-4GL programs.

You may set these environment variables at the system prompt or in your **.profile** (Bourne shell) or **.login** (C shell) file. If you set these variables at the system prompt, you will have to assign them again the next time you log onto the system. If you set these variables in your **.profile** or **.login** file, UNIX will assign them automatically every time you log in.

The following set of instructions assumes that INFORMIX-4GL resides in the directory **/usr/informix**. If, for some reason, INFORMIX-4GL has been installed in another directory, substitute the new directory name for **/usr/informix**.

1. Log in.
2. Set the INFORMIXDIR environment variable.

   If you are using the Bourne shell, enter

   ```
   INFORMIXDIR=/usr/informix
   export INFORMIXDIR
   ```

   If you are using the C shell, enter

   ```
   setenv INFORMIXDIR /usr/informix
   ```

3. Set the TERM environment variable. To do this, you need to obtain the code for your terminal from the system administrator.

   If you are using the Bourne shell, enter

   ```
   TERM=name
   export TERM
   ```

   where *name* is the code for the terminal you are using.

   If you are using the C shell, enter

   ```
   setenv TERM name
   ```

4. Set the TERMCAP environment variable. If the code for your terminal is listed in the TERMCAP file included with **INFORMIX-4GL**, follow the instructions below. Otherwise, consult your system administrator about the procedure for your system.

   If you are using the Bourne shell, enter

   ```
   TERMCAP=/usr/informix/etc/termcap
   export TERMCAP
   ```

   If you are using the C shell, enter

   ```
   setenv TERMCAP /usr/informix/etc/termcap
   ```

5. Include the directory containing **INFORMIX-4GL** in your PATH environment variable.

   If you are using the Bourne shell, enter

   ```
   PATH=$PATH:/usr/informix/bin
   export PATH
   ```

   If you are using the C shell, enter

   ```
   setenv PATH ${PATH}:/usr/informix/bin
   ```

6. If you placed these environment settings in your **.login** or **.profile** file, you should log out and then log back in. (This allows the shell to read the changes you have made.) The new settings will take effect each time you log in.

   If you entered these environment settings from the system prompt, the new settings will remain in effect until you log out. You will need to re-enter them every time you log onto the system.

In addition to the environment variables listed above, you may assign other environment variables by following the instructions in Appendix B.

Setting Environment Variables on DOS Systems

Before you can use **INFORMIX-4GL**, you must follow these steps:

- Use your system editor to set FILES to 20 (or more) and BUFFERS to 8 (or more) in your **CONFIG.SYS** file.

- Set the INFORMIXDIR environment variable so that **INFORMIX-4GL** can locate its programs.

- Use the PATH command to have DOS search **\informix\bin** for executable programs.

You may set the INFORMIXDIR environment variable and run the PATH command at the system prompt or include them in your **AUTOEXEC.BAT** file. If you execute them at the system prompt, you will have to execute them again the next time you start the system. If you set these variables in your **AUTOEXEC.BAT** file, DOS will assign them automatically every time you start the system.

1. Turn your system on.

2. Use your system editor to assign the following values to FILES and BUFFERS in your **CONFIG.SYS** file:

```
FILES=20
BUFFERS=8
```

   These are suggested values only. If you anticipate establishing a large database, you may want to increase the FILES setting beyond 20. You may also want to increase the BUFFERS setting beyond 8. This increases performance, in most instances, by reducing the delay caused by excessive disk accesses.

   After you change the **CONFIG.SYS** file, restart your computer.

   Note: The following instructions assume that **INFORMIX-4GL** resides in the **\informix** directory. If, for some reason, **INFORMIX-4GL** has been installed in another directory, substitute the new directory name for **\informix**.

3. Set the INFORMIXDIR environment variable by entering

```
set INFORMIXDIR=\informix
```

4. Use the PATH command to have DOS search the **\informix\bin** directory for executable programs:

```
set PATH = \informix\bin[;dirname]
```

   The brackets indicate an optional part of the command. Do not enter the brackets if you want to include other directories in the search path. Also, make sure you enter a semicolon between directory names.

5. If you placed the INFORMIXDIR and PATH command settings in your **AUTOEXEC.BAT** file, you should restart your computer. This allows the operating system to read the changes you have made.

If you did not include the INFORMIXDIR and PATH commands in your **AUTOEXEC.BAT** file, you will have to enter them again when you restart the system.

In addition to the environment variables listed above, you may assign other environment variables by following the instructions in Appendix B.

Getting Started with INFORMIX-4GL on DOS Systems

To start working with **INFORMIX-4GL** on DOS systems, be sure that your computer is up and running, and that the operating system prompt appears on your screen.

To compile and execute programs that use **INFORMIX-4GL**, you must perform two steps:

1. Load the *database agent*. The database agent is the part of **INFORMIX-4GL** that performs the data access and file-manipulation tasks necessary to complete your database management operations.

2. Compile and run your **INFORMIX-4GL** programs.

Loading the Database Agent

To load the database agent, enter

```
startsql
```

You need to execute this command only once per session. The database agent remains in memory until you execute the command

```
exit
```

from the command line. When you execute this command, it removes the database agent from memory. Because the database agent requires space in memory, you should remove it when you have completed all the desired **INFORMIX-4GL**, or any SQL, operations. This frees memory for running other programs on your system.

If you attempt to load the database agent when it is already installed in memory, INFORMIX-4GL issues the message STARTSQL has already been executed.

Entering INFORMIX-4GL

After you have loaded the database agent, enter

```
i4gl
```

If you try to run i4gl without first loading the database agent, INFORMIX-4GL issues the message Please run STARTSQL.

If you want to check to see if the database agent has been loaded, enter

```
set
```

This gives you a list of your environment variables. If the database agent has been loaded, the variable SQLCADDR is listed. The following is an example of SQLCADDR as it appears in a list of environment variables. The numbers represent the address in memory where the database agent is loaded.

```
SQLCADDR=333744a8
```

Using Your Terminal

Your computer keyboard looks something like a typewriter keyboard, except that the computer keyboard includes some special keys. You use some of these special keys to give instructions to INFORMIX-4GL. Before you begin using INFORMIX-4GL, locate the following keys on your keyboard:

RETURN	The RETURN key is sometimes labeled CR or NEWLINE, or is marked with a bent arrow. It is located on the right side of the keyboard.

CONTROL	The **CONTROL** key is often labeled **CTRL** or **CNTRL** and is located at the left of the keyboard. In this manual it is called **CONTROL**.
LEFT ARROW	The **LEFT ARROW** key moves the cursor back one position on the screen. If your terminal does not have a **LEFT ARROW** key, use **CONTROL-H**.
DOWN ARROW	The **DOWN ARROW** key moves the cursor down one line. If your terminal does not have a **DOWN ARROW** key, use **CONTROL-J**.
UP ARROW	The **UP ARROW** key moves the cursor up one line. If your terminal does not have an **UP ARROW** key, use **CONTROL-K**.
RIGHT ARROW	This key moves the cursor forward one position on the screen. If your terminal does not have a **RIGHT ARROW** key, use **CONTROL-L**.
DELETE	The **DELETE** key is sometimes labeled **RUBOUT**, **CANCEL**, or **DELETE**, and is referred to as **DEL**. On UNIX systems the **DEL** key is used to abort a program or cancel a prompt in the **INFORMIX-4GL** Programmer's Environment. In this guide it is called the INTERRUPT key.
CONTROL-C	On DOS systems the **CONTROL-C** key is used to abort a program or cancel a prompt in the **INFORMIX-4GL** Programmer's Environment. In this guide it is called the INTERRUPT key.
BACKSPACE	The **BACKSPACE** key may be labeled with a left-pointing arrow, and is usually located at the top right of the keyboard.
ESCAPE	The **ESCAPE** key is sometimes labeled **ESC**.
SPACEBAR	The **SPACEBAR** is generally unlabeled.

In addition, DOS systems have ten function keys marked F1 to F10. The function keys are usually placed at the left of the keyboard.

The Demonstration Database

The INFORMIX-4GL package includes a demonstration database called **stores** that contains information about a fictitious wholesale sporting-goods distributor. You can use it to follow the examples in this manual.

Creating the Demonstration Database

As you learn about INFORMIX-4GL, you will want to experiment with the **stores** database. To do this, you need to make a copy of the database. To make a copy of the database, select the directory where you want to store the copy (often your home directory) and make it your current working directory. Then, from the operating system command line, enter

```
i4gldemo
```

The **i4gldemo** program creates a subdirectory called **stores.dbs** in your current directory and places the **stores** database files there. It also copies all the demonstration programs, forms, and help files into the current directory. If you list the contents of your current directory, you will see filenames similar to these:

```
c_menu2.4gl
ch10def.4gl
ch10def2.4gl
ch10key.4gl
ch10key2.4gl
ch10notf.4gl
ch10when.4gl
ch12cust.4gl
ch12ord.4gl
ch7add.4gl
ch7add2.4gl
ch7del.4gl
ch7qry2.4gl
ch7query.4gl
ch7upd.4gl
cust.per
custcur.per
custhelp.ex
custhelp.msg
custmenu.4gl
customer.per
ordcur.per
order.per
ordmenu.4gl
report1.4gl
report2.4gl
report3.4gl
report4.4gl
report5.4gl
report6.4gl
report7.4gl
stock1.per
wind1.4gl
wind2.4gl
```

The remaining files in the current directory belong to the demonstration application described in Appendix A of the *INFORMIX-4GL Reference Manual.*

Restoring the Database

As you work with your copy of the **stores** database, you may change it in such a way that the illustrations in this manual no longer reflect what you actually see on your screen. This can happen if you enter new information into the demonstration database, delete the information that came with the database, or alter the structure of the tables, forms, or reports.

You can restore the database to its original condition (the one upon which the examples are based) by recreating the database with the i4gldemo command.

You may want to make a fresh copy of the demonstration database each time you start a new chapter.

If you installed your INFORMIX-4GL software according to the instructions provided in your installation letter, the files that make up the **stores** database are protected so that you cannot make any changes to the original copy of the database.

Restricted and Public Databases

You may make a database accessible to other users by giving them read and execute permissions on all the directories in the database path (UNIX systems only) and by using the GRANT CONNECT statement in the **INFORMIX-4GL** program that creates the database.

1. If you are using the UNIX operating system, you must give users of your database read and execute permissions on all the directories in the database path. For example, you may make the directory containing the database accessible to all users by entering the command

   ```
   chmod 755 dirname
   ```

   assuming that all directories above *dirname* have at least the same permissions.

2. Under both the UNIX and DOS operating systems, you must use the GRANT CONNECT statement in the **INFORMIX-4GL** program that creates the database to give other users access to the database. The forms of the GRANT CONNECT statement are

   ```
   GRANT CONNECT TO PUBLIC
   GRANT CONNECT TO user-list
   ```

   where PUBLIC refers to all users and *user-list* is a list of login names that corresponds to specific users. Once you have granted the CONNECT privilege to one or more users, they also have default table privileges unless someone has explicitly changed them. This means that you can insert, select, update, and delete rows in a table. (See Chapter 14 of this guide and the *INFORMIX-4GL Reference Manual* for more information about the GRANT statement.)

Chapter 1

Introduction

Chapter 1 Table of Contents

Chapter Overview 5

Introducing INFORMIX-4GL 6
What Are Fourth-Generation Languages? 6
Procedural and Non-Procedural Languages 7

Features of INFORMIX-4GL 8
A Database Language 9
A Programming Language 10
A Screen-Building Utility 11
Entering Data 11
Query by Example 13
A Menu-Building Utility 15
A Report Writer 16
A Window Manager 18
The INFORMIX-4GL Programmer's Environment 22
A Creative Solution for Database Applications 22

Chapter Overview

This chapter introduces important concepts about INFORMIX-4GL, including

- What INFORMIX-4GL is
- How you can use INFORMIX-4GL to create and maintain databases, design screen forms, create menus, produce reports, and manage windows
- Why INFORMIX-4GL is the best fourth-generation language for custom database applications

Introducing INFORMIX-4GL

INFORMIX-4GL is a fourth-generation language developed by Relational Database Systems, Inc. (RDS) and designed specifically for database applications. Fourth-generation languages such as INFORMIX-4GL represent the latest advancement in programming. In this section, you will learn

- What fourth-generation languages are
- How they differ from general-purpose, third-generation languages
- How you can benefit by using a fourth-generation language such as INFORMIX-4GL to develop database applications

What Are Fourth-Generation Languages?

Fourth-generation programming languages such as INFORMIX-4GL are designed for a particular class of applications. They are less complex than general-purpose languages like COBOL or C, and more closely approximate "natural language." Because they focus on a specific type of application, fourth-generation languages can anticipate what you want to accomplish in your programs. Also, fourth-generation languages are very powerful; one simple statement generates a great deal of machine code. As a result, programs written in a fourth-generation language such as INFORMIX-4GL do not contain nearly as many statements as programs written in a general-purpose language.

Some advantages of fourth-generation languages are

- They are simple, which speeds up the process of building and maintaining applications.
- They are generally interactive, which simplifies the debugging process.

- They appeal to a wide audience because they require no special training.

- The resulting applications are easy to use and solve problems efficiently.

Procedural and Non-Procedural Languages

Programming languages are sometimes referred to as procedural or non-procedural. When you use a procedural language, you specify in your program *how* you want to accomplish something. This step-by-step approach makes a procedural language very flexible; it can be used for a variety of applications.

For example, if you are designing a menu-driven program using a procedural language such as COBOL or C, you have to specify, step by step, how to display the menu and handle input from the user. Such a program would include statements for displaying the menu title and options and for moving the cursor from one option to another. Such a program would also include conditional statements like IF or CASE that perform a series of operations, depending on the user's input.

On the other hand, when you use a non-procedural language, you specify the result you want and the language supplies the procedure. Now, imagine you want to design a menu-driven program using a non-procedural language. In this case, you would create a menu by using a statement such as the INFORMIX-4GL MENU statement. You would not have to use print statements to display the menu title and options because MENU has built-in procedures that display the menu for you. Likewise, you would not have to use conditional statements to handle requests from the user because MENU, essentially, creates a CASE-like statement.

INFORMIX-4GL combines the features of procedural and non-procedural languages. You have seen how INFORMIX-4GL offers non-procedural statements like the MENU statement to make building applications simple. INFORMIX-4GL also provides

procedural statements like IF, FOR, and WHILE so that you can do things that the designers of INFORMIX-4GL could not predict. Thus, INFORMIX-4GL combines the speed and simplicity of non-procedural languages with the flexibility of procedural languages.

Features of INFORMIX-4GL

INFORMIX-4GL is a very powerful fourth-generation language, providing all the tools you need to create relational database management systems. In the next section, you will see that INFORMIX-4GL is

- A database language that you can use to store, retrieve, update, and delete information in databases that you have created
- A programming language
- A screen-building utility
- A menu-building utility
- A report writer
- A window manager

A Database Language

INFORMIX-4GL is a development tool designed specifically for writing programs that create relational databases and provide the user with facilities to manipulate the data stored in these databases.

INFORMIX-4GL is built upon RDSQL, the RDS extension of the Structured Query Language (SQL) developed by IBM. SQL is rapidly becoming the standard query language for database management systems, and RDSQL meets the standards established by ANSI for SQL implementations. RDSQL offers statements such as CREATE DATABASE, INSERT, SELECT, UPDATE, and DELETE so that you can write programs that allow the user to add, retrieve, update, and delete information in a database.

Figure 1-1 shows three simple RDSQL statements that might appear in an INFORMIX-4GL program.

```
CREATE DATABASE phones

CREATE TABLE home_phones
     (fname    CHAR(15),
      lname    CHAR(15),
      phone    CHAR(12) )

INSERT INTO home_phones
     VALUES ("Enid", "Smythe", "415-322-1000")
```

Figure 1-1. RDSQL Statements

The first two statements create a database called **phones** that contains a table called **home_phones**, which is designed to store names and phone numbers. The third statement inserts Enid Smythe's name and phone number into the **home_phones** table.

A Programming Language

Though it is designed specifically for database applications, **INFORMIX-4GL** still has many features of a programming language. In addition to database-specific statements, **INFORMIX-4GL** includes statements similar to those found in general-purpose languages. Basic statements for assigning values, looping, and conditional branching are all provided by **INFORMIX-4GL**.

For example, you assign values to **INFORMIX-4GL** program variables using the LET statement. Statements such as WHILE and FOR set up program loops that function like those in third-generation languages. The IF and CASE statements in **INFORMIX-4GL** perform in virtually the same fashion as the corresponding statements in C and Pascal. **INFORMIX-4GL** also provides data structures such as records and arrays that allow you to manipulate many values simultaneously.

As programs become larger and more complex, you can use the FUNCTION statement to create subroutines. Figure 1-2 shows a function from an **INFORMIX-4GL** program that uses statements similar to those found in third-generation languages to set up a loop and call a nested function.

```
FUNCTION enter_data()

   DEFINE answer CHAR(1)

   LET answer = "y"

   WHILE answer = "y"

     CALL enter_cust()

     PROMPT "Do you want to enter another customer (y/n)? "
       FOR ANSWER

   END WHILE

END FUNCTION
```

Figure 1-2. A Function That Includes a WHILE Loop

Many database-management systems provide screen forms through which the user can enter or display information. **INFORMIX-4GL** includes a utility called **FORM4GL** that enables you to create screen forms with a minimum of effort. **FORM4GL** can automatically generate a default form for any table(s) in a database. You can also use **FORM4GL** to design custom screen forms that are both complex and aesthetically pleasing.

Entering Data

Once you have created a screen form, you can use the INPUT statement to allow the user to enter data on the form and transfer it to program variables. Or, you can use the DISPLAY statement to display data in program variables on the form. **INFORMIX-4GL** handles cursor movement automatically and provides built-in editing capabilities.

Following are two screen forms that were created using **FORM4GL**:

```
                        CUSTOMER  FORM

      Number: [          ]

  First Name: [              ]     Last Name: [              ]

     Company: [                    ]

     Address: [                    ]
              [                    ]

        City: [              ]

       State: [  ]  Zipcode: [      ]

   Telephone: [                ]
```

Figure 1-3. A **customer** Form

The **customer** screen form in Figure 1-3 allows the user to enter customer information.

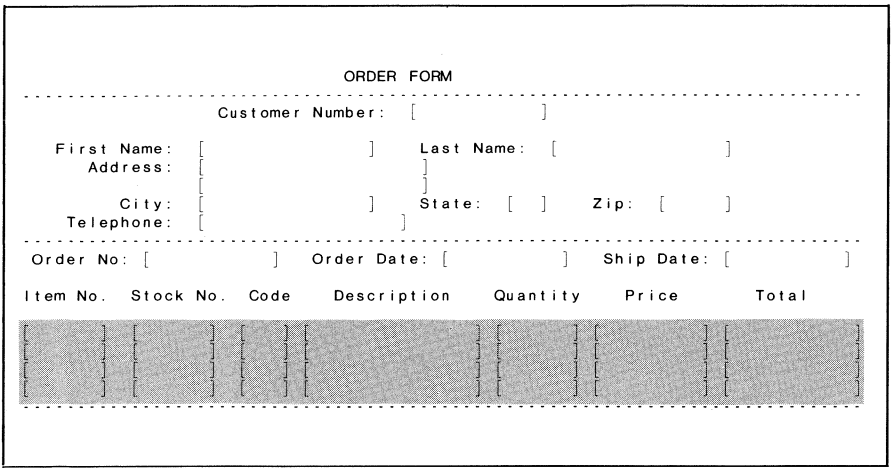


Figure 1-4. An **order** Form That Contains a Screen Array (shaded)

The **order** screen form in Figure 1-4 allows the user to enter and update several items for a particular order at a time through a *screen array*. Since this screen array is actually a *window* on a larger program array, the user can enter as many rows as the program array can hold.

For example, the user can enter up to ten rows of information through the screen array in the **order** form since the underlying program array can store ten rows of information (see Figure 1-5). (Your program array might store 30, 50, or 100 rows of information, depending on the dimensions you specify.)

Item No.	Stock No.	Code	Description	Quantity	Price	Total
1	8	ANZ	volleyball	2	840.00	1680.00
2	5	SMT	tennis racquet	15	25.00	375.00
[3]	[2]	[HRO]	[baseball]	[1]	[126.00]	[126.00]
[4]	[5]	[NRG]	[tennis racquet]	[10]	[28.00]	[280.00]
[5]	[6]	[SMT]	[tennis ball]	[2]	[36.00]	[72.00]
[6]	[3]	[HSK]	[baseball bat]	[3]	[240.00]	[720.00]
7	1	HRO	baseball gloves	5	250.00	1250.00
8	9	ANZ	volleyball net	12	20.00	240.00
9	4	HSK	football	3	960.00	2880.00
10	1	SMT	baseball gloves	2	450.00	900.00

Figure 1-5. Screen Array as a Window on a Program Array

When the screen array is full, the data scrolls automatically so that the user can continue entering items. The user can also use editing keys to change date, insert or delete rows, and scan through the data a screenfull at a time.

Query by Example

With the powerful CONSTRUCT statement, you can let the user perform a *query by example.* By filling in one or more fields on a screen form, the user specifies what information should be retrieved from the database. What makes this feature so powerful are the relational, range, alternation, and wildcard symbols that the user can include in a query.

For example, the user can search for all customers with last names that begin with *B* who live in Palo Alto by filling in the Last Name and City fields of the **customer** form as follows:

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                          CUSTOMER  FORM

       Number:  [            ]

   First Name:  [                ]     Last Name:  [B*               ]

      Company:  [                     ]

      Address:  [                     ]
                [                     ]

         City:  [Palo Alto       ]

        State:  [  ]   Zipcode:  [      ]

    Telephone:  [                  ]
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

Figure 1-6. Query by Example on the **customer** Form

Or, the user can search for all orders placed by customers with the last name "Grant" or "Miller" after June 5, 1986, which include a particular item:

```
                              ORDER FORM
--------------------------------------------------------------------------------
                  Customer Number:  [            ]

   First Name:  [                ]     Last Name:  [Grant|Miller    ]
      Address:  [                     ]
                [                     ]
         City:  [                ]     State:  [  ]    Zip:  [      ]
     Telephone:  [                  ]
--------------------------------------------------------------------------------
 Order No: [            ]   Order Date: [>06/05/86 ]   Ship Date: [            ]

Item No.   Stock No.   Code    Description    Quantity     Price        Total

[        ] [        ] [    ] [baseball glove ] [        ] [            ] [            ]
[        ] [        ] [    ] [               ] [        ] [            ] [            ]
[        ] [        ] [    ] [               ] [        ] [            ] [            ]
[        ] [        ] [    ] [               ] [        ] [            ] [            ]
--------------------------------------------------------------------------------
```

Figure 1-7. Query by Example on the **order** Form

When the user completes a query by example, INFORMIX-4GL automatically creates a string that can be used to retrieve information from the database. For example, the search criteria entered on the **order** form produce the following string that you can use to select data from a database:

```
lname in ("Grant","Miller")
and order_date>"06/05/86"
and description="baseball glove"
```

A Menu-Building Utility

As mentioned previously, INFORMIX-4GL provides the powerful MENU statement that simplifies the process of creating menus. The MENU statement enables you to build a menu such as the one in Figure 1-8 with a minimum amount of code. You can even nest menus within menus to create a menu interface for your application. By using the MENU statement, you can ensure that the menus in your applications are consistent.

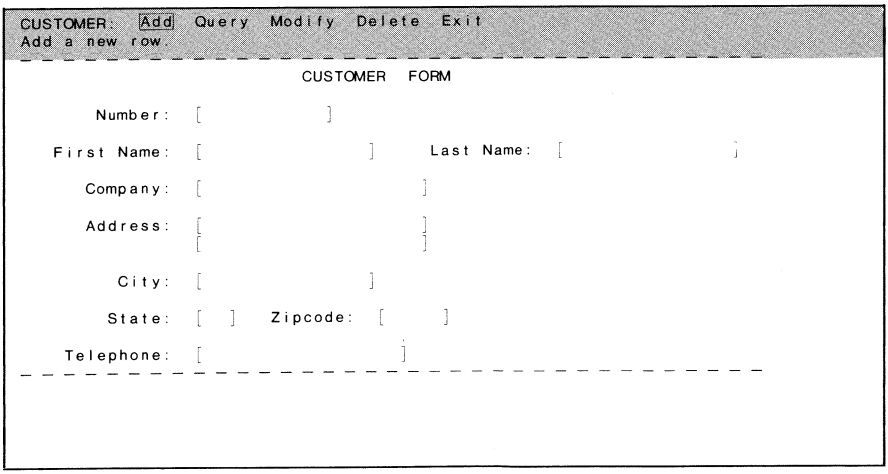


Figure 1-8. A Sample Menu (shaded)

With little extra work, you can provide online help for the menu by creating a help file and specifying help messages at appropriate places in your application. For example, you can add online help to the CUSTOMER Menu in Figure 1-8 so that INFORMIX-4GL displays the following help message when the user requests help while the **Add** option is highlighted:

```
HELP:    Screen  Resume
Ends this session.

To add a row to the database, type the information
about the customer into the spaces between the brackets,
which are called "fields."  The label to the
left of a field tells what type of information
the field should contain.

After you have filled in one field, press RETURN to move the
cursor to the next field.  When you have filled the last
field, press ESCAPE or RETURN to add this customer to the
database.  If you want to add the customer without filling
in all fields, press ESCAPE.

When the program prompts you, type "y" to add another
customer, or type "n" to return to the CUSTOMER Menu.




You have reached the end of this help text.  Press RETURN to continue.
```

Figure 1-9. Help Screen for the **Add** Option of the CUSTOMER Menu

A Report Writer

A primary function of most database applications is to provide reports based on the information in the database. INFORMIX-4GL offers statements that allow you to create a variety of reports, from simple default reports to custom-formatted reports. With INFORMIX-4GL, you can retrieve data from the database and then order, group, and format the data to your exact specifications. INFORMIX-4GL also has built-in functions that allow you to determine minimum, maximum, and average values, as well as calculate percentages and totals for the data in your report. Advanced formatting capabilities such as adjustable page lengths and margins make it easy to produce columnar reports, payroll checks, invoices, and more.

In Chapter 9 of this guide, you will learn how to create reports such as the one shown in Figure 1-10.

Stock Number	Manufacturer	Description	Unit	Quantity
1	HRO	baseball gloves	case	1
1	HRO	baseball gloves	case	1
1	HRO	baseball gloves	case	1
1	HSK	baseball gloves	case	1
1	SMT	baseball gloves	case	1
1	SMT	baseball gloves	case	1
		Total Quantity on Order:		6
2	HRO	baseball	case	1
2	HRO	baseball	case	1
		Total Quantity on Order:		2
3	HSK	baseball bat	case	1
3	HSK	baseball bat	case	1
3	HSK	baseball bat	case	1
		Total Quantity on Order:		3

Figure 1-10. A Sample Report

A Window Manager

With INFORMIX-4GL, you can create applications that devote different rectangular parts of the screen to different activities. Each rectangular portion of the screen is called a *window.*

INFORMIX-4GL includes powerful window management statements that allow you to open, clear, close, and change windows within your programs. In Chapter 13, for example, you will learn how to create an order entry program that opens a customer selection window when the user presses a designated help key:

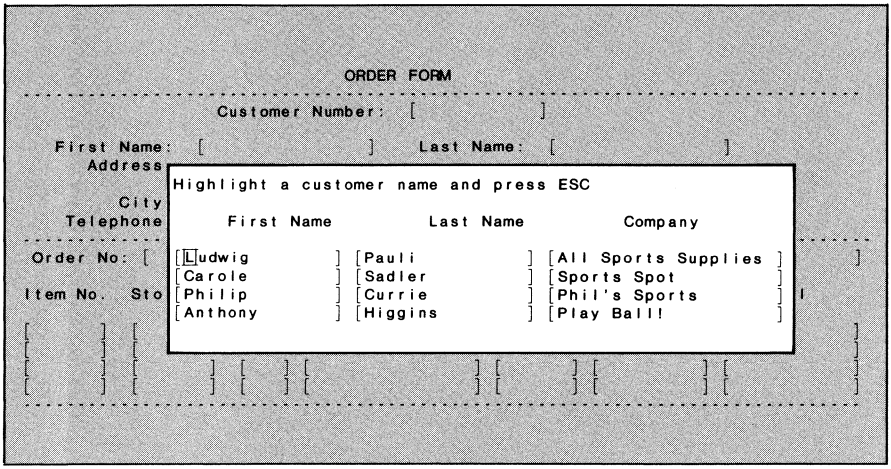


Figure 1-11. A Customer Selection Window

When the user selects a customer by positioning the cursor on a row and pressing the **ESCAPE** key, the order entry program automatically displays the customer information in the customer information fields of the underlying form:

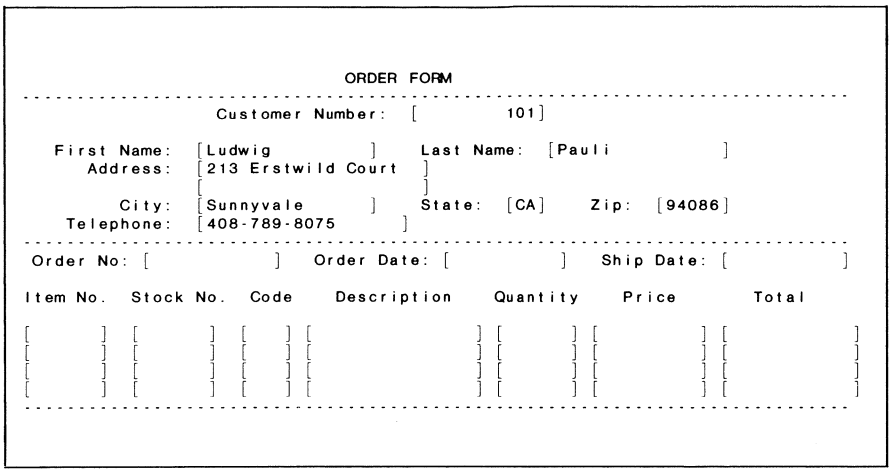


Figure 1-12. Automatic Display of Customer Information

Chapter 13 also shows you how to create an application that allows the user to select customers in one window and retrieve the orders they have placed in another window:

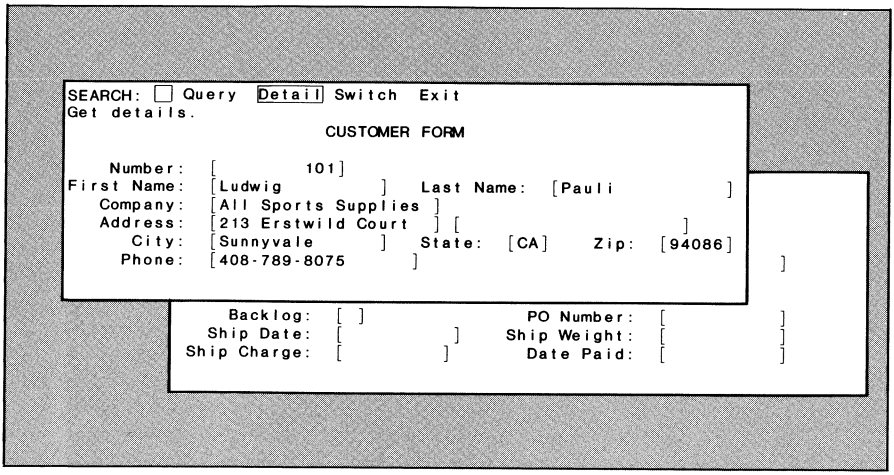


Figure 1-13. The Customer Information Window

When the user selects the **Detail** option in the customer information window, the program automatically retrieves the orders for the current customer and displays the first order in the order information window:

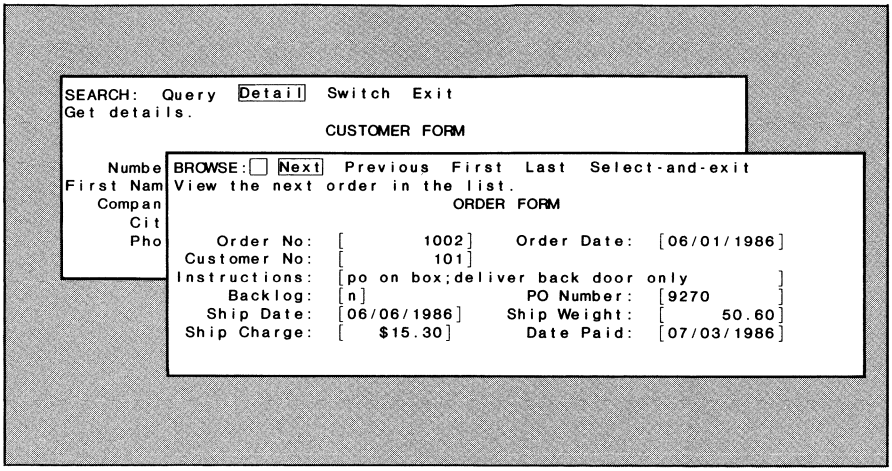


Figure 1-14. The Order Information Window

With the window management statements provided by INFORMIX-4GL, you can create an effective user interface for your application programs.

The INFORMIX-4GL Programmer's Environment

INFORMIX-4GL provides a Programmer's Environment consisting of a series of menus that guide you through all the steps involved in developing an application. In the INFORMIX-4GL Programmer's Environment, you can work on program modules, create and compile screen forms, compile and link modules together to create multi-module programs, and use the RDSQL query language (if you have INFORMIX-SQL).

In addition, INFORMIX-4GL includes library functions and several utility programs that check and restore the integrity of your index files, load data from different sources, create default attributes for fields on your screen forms, and more.

A Creative Solution for Database Applications

INFORMIX-4GL represents a creative solution to the need for flexibility and simplicity in a fourth-generation language. Its basic statements are simple, and yet the optional extensions provide the flexibility you need for complex applications. The non-procedural statements are very compact and handle the bulk of the application. The procedural statements enable you to do things that the designers of INFORMIX-4GL could not predict. You may not need to use all the features of INFORMIX-4GL, but when you want to do something special in your database application, the chances are that INFORMIX-4GL has supplied you with the proper tools.

Chapter 2

Creating a Database

Chapter 2 Table of Contents

Chapter Overview 5

Basic Database Concepts 6
What is a Database? 6
Database Tables 7
Rows and Columns 8
Columns 8
Rows 9
Joining Tables 10
Join Columns in the *stores* Database 11
Indexes 17

Creating a Database 18
The MAIN Statement 18
The CREATE DATABASE Statement 19
Guidelines for Naming a Database 20
The CREATE TABLE Statement 20
Column Names 21
Assigning Data Types 22
CHAR Columns 22
Numeric Columns 23
SERIAL Columns 25
DATE Columns 26
MONEY Columns 27
Examining a Sample Table 28
Determining Column and Row Lengths 29
Creating an Index 31
Types of Indexes 31
When to Index a Table 32
The CREATE INDEX Statement 33
Guidelines for Naming the Index 34
The UNIQUE and DISTINCT Keywords 35
Ascending and Descending Indexes 35

The Programmer's Environment 37
Entering the Programmer's Environment 38
A Program That Creates a Database 39
Formatting Programs 40

Using Comments 40
A Brief Tutorial 41

Chapter Summary 48

Chapter Overview

This chapter introduces the basic database concepts that are used throughout this guide. It also presents the **INFORMIX-4GL** Programmer's Environment, a series of menus that you can use to create and compile **INFORMIX-4GL** programs. Topics discussed in this chapter include

- What is a database
- What are tables
- What are data types
- What is an index
- How to write an **INFORMIX-4GL** program that creates a database, table, and index
- How to compile and run programs in the **INFORMIX-4GL** Programmer's Environment

For complete information about the statements used to create a database, refer to Chapter 2 and Chapter 7 of the *INFORMIX-4GL Reference Manual.*

Basic Database Concepts

This section explains database concepts that include

- What is a database
- What is a table
- What are rows and columns
- What are indexes

What is a Database?

A *database* is a collection of information that is useful to an organization or that is used for a particular purpose.

The telephone directory is a database; it includes names, addresses, and telephone numbers of residents and businesses organized alphabetically by last name. Computerized databases, such as those you create and manipulate with **INFORMIX-4GL** programs, take advantage of the storage capacity and quick accessing ability of computer systems. Some common automated database applications are

- Airline-ticketing systems
- Mail-order catalog mailing lists
- Retail credit-card billing systems

Database Tables

A database is made up of tables. A table is a collection of information organized in rows and columns. Figure 2-1 shows the rows and columns of a typical database table.

Data in the *customer* Table

customer num	fname	lname	name	address1
101	Ludwig	Pauli	All Sports Supplies	213 Erstwild Court
102	Carole	Sadler	Sports Sport	785 Geary St.
103	Philip	Currie	Phil's Sports	654 Poplar
104	Anthony	Higgins	Play Ball!	East Shopping Cntr.
105	Raymond	Vector	Los Altos Sports	1899 La Loma Drive

(rows) (rows) (columns)

Figure 2-1. Rows and Columns in a Database Table

Every database must contain at least one table and may have as many tables as you need. The number of tables is limited only by the amount of disk space available on your computer.

Each table in a database normally contains a different kind of information. For example, you would probably maintain separate tables for the products you sell, the orders you take, and the customers you serve.

Throughout this guide you will see references to the **stores** database, which contains information used by a fictitious distributor who sells equipment to sporting-goods stores. The distributor maintains different kinds of data, which are grouped into the following six tables:

customer Contains information about customers, including name, company, address, and phone number

orders	Contains information about orders, including order number, order date, shipment date, and shipping instructions
items	Contains information about the items ordered, including stock number, manufacturer code, quantity, and total price
stock	Contains a description of each item offered by the distributor and its unit price
manufact	Contains the name and an identification code for each equipment manufacturer whose products the distributor offers
state	Contains the name and abbreviation for each state

The **stores** database is included with your INFORMIX-4GL software as a demonstration database. See Appendix A of this guide for more information about the demonstration database.

Rows and Columns

Figure 2-1 shows you how a table is organized into *rows* and *columns*. The information in the figure includes data from the **customer** table. The actual table has more columns and many more rows. The following sections explain the significance of these rows and columns.

Columns

Each column contains a specific type of information. In Figure 2-1, the first column contains the first name of a customer, the second column contains the customer's last name, and so on.

When you learn how to create a table in the next section, you will assign a name to each column. These *column names* are

similar to the headings in a printed report. **fname**, **lname**, **company**, and **phone**, shown in Figure 2-1, are some of the column names in the **customer** table. In your INFORMIX-4GL programs, you will use these column names to refer to the columns in the table.

Figure 2-2 lists all columns in the **customer** table by column name and explains the information in each column.

This column:	Contains this information:
customer_num	customer identification number
fname	first name
lname	last name
company	company name
address1	company address
address2	company address
city	city
state	state
zipcode	zip code
phone	phone number

Figure 2-2. Columns of the **customer** Table

Rows

A row contains all the data about one of the objects the table describes. Each row contains an entry for some or all of the columns in the table. For example, the first row in Figure 2-1 contains information about the customer Philip Currie. Philip Currie's row in the **customer** table contains ten entries, including his customer number, name, company, address, and phone number.

When you first create a table, it has no rows. In subsequent chapters, you will learn how to create INFORMIX-4GL programs that add rows to a table.

Joining Tables

With INFORMIX-4GL, you can connect data from different tables using a technique called *joining*. Joining eliminates the need to duplicate the same information in different tables because it enables you to look at data stored in several tables as if it were all part of the same table. When tables are joined, you can retrieve information from both tables with one INFORMIX-4GL statement.

If you are going to join two tables, you must make sure that one column in each table has the same data. These columns are called *join columns*. You use them to create a temporary link between the tables. Though the columns contain identical information, they need not have the same name.

Join Columns in the *stores* Database

The tables in the **stores** database have several join columns. For example, the **customer** table and the **orders** table are joined by the **customer_num** column, as shown in the following example:

customer table (detail)

customer_num	fname	lname
101	Ludwig	Pauli
102	Carole	Sadler
103	Philip	Currie
104	Anthony	Higgins

orders table (detail)

order_num	order_date	customer_num
1001	01/20/1986	104
1002	06/01/1986	101
1003	10/12/1986	104
1004	04/12/1986	106

Figure 2-3. Tables Joined by the **customer_num** Column

For example, the **customer** table contains a **customer_num** column that holds a number identifying the customer, along with columns for name, company, address, and telephone number. The row with information about Anthony Higgins contains the number 104 in the **customer_num** column. The **orders** table also contains a **customer_num** column that stores the number of the customer who placed a particular order. According to Figure 2-3, customer 104 (Anthony Higgins) has placed two orders, since his customer number appears in two rows of the **orders** table.

The join relationship lets you select information from both tables. This means you can retrieve Anthony Higgins's name and address and information about his orders at the same time.

The six tables of the **stores** database are linked together by join columns. The following figures show the relationship between the tables, and how information stored in one table supplements information in other tables.

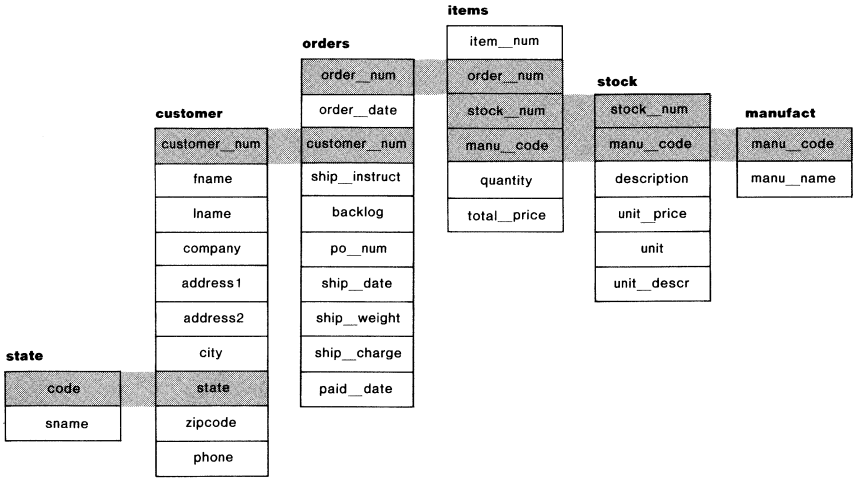


Figure 2-4. Join Relationships in the **stores** Database (potential join columns are shaded)

Join Columns in the **orders** *and* **items** *Tables.* As you can see in Figure 2-4, the **orders** and **items** tables are linked by an **order_num** column that contains an identification number for each order. If a particular order includes several items, the same order number will appear in several rows of the **items** table, as shown in the following example:

orders table (detail)

order_num	order_date	customer_num
1001	01/20/1986	104
1002	06/01/1986	101
1003	10/12/1988	104

items table (detail)

item_num	order_num	stock_num	manu_code
1	1001	1	HRO
1	1002	4	HSK
2	1002	3	HSK
1	1003	9	ANZ
2	1003	8	ANZ
3	1003	5	ANZ

Figure 2-5. Tables Joined by the **order_num** Column

Join Columns in the **items** *and* **stock** *Tables.* The **items** table and the **stock** table are joined by two columns: the **stock_num** column stores a stock number for an item, and the **manu_code** column stores a code that identifies the manufacturer. You need both the stock number and the manufacturer code to uniquely identify an item. For example, the item with the stock number 1 and the manufacturer code HRO is a Hero baseball glove, while the item with the stock number 1 and the manufacturer code HSK is a Husky baseball glove.

The same stock number and manufacturer code may appear in more than one row of the **items** table if the same item belongs to separate orders, as shown in the following example:

items table (detail)

item_num	order_num	stock_num	manu_code
1	1001	1	HRO
1	1002	4	HSK
2	1002	3	HSK
1	1003	9	ANZ
2	1003	8	ANZ
3	1003	5	ANZ
1	1004	1	HRO

stock table (detail)

stock_num	manu_code	description
1	HRO	baseball glove
1	HSK	baseball glove
1	SMT	baseball glove

Figure 2-6. Tables Joined by Two Columns

Join Columns in the **stock** *and* **manufact** *Tables.* The **stock** table and the **manufact** table are joined by the **manu_code** column. The same manufacturer code may appear in more than one piece of equipment, as shown in Figure 2-7.

stock table (detail)

stock_num	manu_code	description
1	HRO	baseball glove
1	HSK	baseball glove
1	SMT	baseball glove
2	HRO	baseball

manufact table (detail)

manu_code	manu_name
NRG	Norge
HSK	Husky
HRO	Hero

Figure 2-7. Tables Joined by the **manu_code** Column

Join Columns in the **state** *and* **customer** *Tables.* The **state** table and the **customer** table are joined by a column that contains the state code. This column is called **code** in the **state** table and **state** in the **customer** table. If several customers live in the same state, the same state code will appear in several rows of the **customer** table, as shown in the following example.

customer tabl
(detail)

customer_num	fname	lname	...	state
101	Ludwig	Pauli	...	CA
102	Carole	Sadler	...	CA
103	Philip	Currie	...	CA

state table
(detail)

code	sname
AK	Alaska
AL	Alabama
AR	Arkansas
AZ	Arizona
CA	California

Figure 2-8. Tables Joined by the State-Code Column

Joining lets you rearrange your view of a database whenever you want. It provides flexibility that lets you create new relationships between the tables without redesigning the database. You can easily expand the scope of a database by adding new tables that join the tables you already have created. As you read through this guide, you will see programs that set up the join relationships between tables of the **stores** database. You can refer to the preceding figures whenever you need to review the relationships between tables.

Indexes

An *index* helps INFORMIX-4GL to find rows in a table more quickly. If INFORMIX-4GL needs to find a row in an indexed table, it uses the index to go directly to the row. When INFORMIX-4GL searches for a row in a table *without* an index, it must start with the first row and continue searching sequentially until it finds the row.

If you create an index on a column in a table, INFORMIX-4GL sets up an index file containing the values found in the indexed column. These values are listed in ascending ASCII order for CHAR columns or in numeric order for numeric columns, unless you specify otherwise, as explained later in this chapter. Along with each value, the index file lists the number of the row containing the same value in the indexed column.

For example, suppose you have a table of names and addresses for which you create an index on the last-name column. The corresponding index file will contain each last name and a record number telling INFORMIX-4GL where to locate the row with the same value in the last-name column.

Indexes are very useful for tables with many rows. The section "Creating an Index," later in this chapter, provides guidelines for determining when to index columns.

Creating a Database

Unless you have another SQL product from Relational Database Systems, you must create a database and its tables with an INFORMIX-4GL program. This section describes the INFORMIX-4GL program statements that are used to create a database:

- MAIN
- CREATE DATABASE
- CREATE TABLE
- CREATE INDEX

In the section "The Programmer's Environment," later in the chapter, you will create a small program that uses these statements.

Note: Throughout this text, keywords such as MAIN always appear in uppercase letters to make them easier for you to identify. However, INFORMIX-4GL does not distinguish between uppercase and lowercase letters in a program. Therefore, you can use either style, or a mixture of both, in your programs.

The MAIN Statement

Every INFORMIX-4GL program must contain a MAIN statement. The format of the MAIN statement is

```
MAIN
  statement
  ...
END MAIN
```

where MAIN is a keyword that indicates the beginning of the main program, *statement* is any INFORMIX-4GL statement, and END MAIN are keywords that terminate the main program.

The following example shows a simple program that sends a message to the screen:

```
MAIN
  DISPLAY "Welcome to INFORMIX-4GL"
END MAIN
```

The DISPLAY statement is described in Chapter 3.

The CREATE DATABASE Statement

To create a database, you must use the CREATE DATABASE statement. Its simplest form is

CREATE DATABASE *database-name*

where CREATE DATABASE are keywords and *database-name* is the name you want to assign to the database.

For example, to create a database named **mydb**, you would write an **INFORMIX-4GL** program like the following:

```
MAIN
    CREATE DATABASE mydb
END MAIN
```

When **INFORMIX-4GL** encounters this statement, it creates the **mydb.dbs** directory in your current directory. The **mydb.dbs** directory contains files called system catalogs that store information about tables, columns, indexes, and authorizations.

Guidelines for Naming a Database

You can assign any name to a database, as long as you follow these guidelines:

- The database name cannot exceed ten characters if you are using a UNIX-based system, or eight characters if you are using a DOS-based system.

- The database name must begin with a letter. The rest of the name may contain any combination of letters, numbers, and underscores (_). You can use any combination of uppercase and lowercase letters, but be aware that INFORMIX-4GL treats them as though they were the same.

- If you store more than one database in a single directory, each database must have a unique name.

- Do not use INFORMIX-4GL *reserved words* for the database name. If you use a reserved word for a database name, INFORMIX-4GL will not compile your program. (See Appendix C for a list of INFORMIX-4GL reserved words.)

Note: If other users are going to work with the database, you must use the GRANT statement to give them CONNECT access to the database. Refer to Chapter 14 of this guide or Chapter 7 of the *INFORMIX-4GL Reference Manual* for complete information about the GRANT statement.

The CREATE TABLE Statement

The CREATE TABLE statement contains the instructions that INFORMIX-4GL uses to create a table in the database. Its simplest form is

```
CREATE TABLE table-name
  (column-name  data-type [,...] )
```

where CREATE TABLE are keywords, *table-name* is the name of the table, *column-name* is the name of a column in the table, and *data-type* indicates the type of information that the

column contains. You must use a separate CREATE TABLE statement for every table in the database.

Table names can have up to 18 characters. Otherwise, they have the same restrictions as database names. Refer to the section "Guidelines for Naming a Database," earlier in this chapter, for more information.

Column Names

Follow the table name with a list of column names and corresponding data types. Make sure you include a comma after each data type except the last and enclose the list of column names and data types in parentheses:

```
CREATE TABLE table-name
(column1  data-type, column-2  data-type2 [,...] )
```

Follow these guidelines when naming columns:

- A column name cannot exceed 18 characters.

  Note: If you plan to define a program record LIKE *table.**, make sure that the first eight characters of each column name are unique. (See Chapter 4 for more information about defining program records using the LIKE keyword.)

- Column names in the same table must be unique. However, the same column name may appear in different tables in the same database. For example, both the **customer** and **orders** tables in the **stores** database have a column called **customer_num**.

- A column name must begin with a letter. The rest of the name may contain any combination of letters, numbers, or underscores(_). INFORMIX-4GL does not distinguish between uppercase and lowercase letters.

- Do not use an INFORMIX-4GL reserved word for a column name.

Assigning Data Types

You must assign a *data type* to each column in a table. The data type indicates what kind of information you intend to store in the column:

Character stores any combination of letters, numbers, and symbols.

Numeric stores numeric data. There are six different numeric data types.

Date stores calendar dates.

Money stores currency amounts.

The following sections describe these data types in detail.

CHAR Columns

CHAR (character) columns store any combination of letters, numbers, and symbols. You normally use CHAR columns to store names, addresses, phone numbers, and so on. Use the following format to specify the length of each CHAR column

```
CHAR(n)
```

where *n* is the total number of characters you want to reserve for the column. An example follows:

```
CREATE TABLE customer (lname char(15))
```

The statement creates a table named **customer** with one column named **lname**. This column can hold a name up to 15 characters long.

Note: You can store numbers in CHAR columns, but you may not be able to use them in arithmetic operations. If you want to perform calculations on numbers stored in a column, you should assign a numeric data type to that column.

Numeric Columns

INFORMIX-4GL has six data types for storing numbers; each is designed for a different type of number. You cannot store characters or symbols in any type of numeric column, although you can use plus signs (+) and minus signs (−) to show whether numbers are positive or negative.

The numeric data types are summarized as follows:

Data Type	Type of Number
DECIMAL	Numbers with definable scale and precision
SMALLINT	Whole numbers from −32,767 to +32,767
INTEGER	Whole numbers from −2,147,483,647 to +2,147,483,647
SMALLFLOAT	Single-precision, floating-point numbers corresponding to the float data type in the C language
FLOAT	Double-precision, floating-point numbers corresponding to the double data type in the C language
SERIAL	Sequential integers assigned by INFORMIX-4GL

DECIMAL Columns. These columns store numbers with the number of digits that you specify. When you create a DECIMAL column, you can specify the total number of significant digits up to a maximum of 32. Optionally, you may also specify the number of digits after the decimal point. The general form is

decimal (*precision* [,*scale*])

where *precision* is the total number of digits you want, and *scale* is the number of digits after the decimal point.

For example,

```
CREATE TABLE mytable (mycol DECIMAL(6,2))
```

creates a table named **mytable** containing a DECIMAL column named **mycol**. The **mycol** column stores six-digit numbers, with four digits before and two digits after the decimal point.

If you do not specify the scale for a DECIMAL column, **INFORMIX-4GL** treats the column as having a floating decimal point. This means that DECIMAL (*m*) columns have a precision of *m* and a range in absolute value from 10^{-128} to 10^{126}. If you do not specify a precision for a DECIMAL column, **INFORMIX-4GL** treats the column as DECIMAL(16) with a floating decimal point.

Note: Whenever possible, store floating-point numbers in DECIMAL columns instead of SMALLFLOAT or FLOAT columns. When you enter a numeric value into a SMALLFLOAT or FLOAT column, **INFORMIX-4GL** has to convert the value from decimal to binary format so that it can be stored. Likewise, when a SMALLFLOAT or FLOAT number is displayed, **INFORMIX-4GL** has to convert it from binary to decimal format, which may lead to inaccuracy. For example, if you enter the value 10.7 into a FLOAT column, it may actually be stored as 10.6999.

SMALLINT and INTEGER Columns. These columns store whole numbers—numbers that have no fractional part. SMALLINT columns store whole numbers from −32,767 to +32,767. INTEGER columns store whole numbers from −2,147,483,647 to +2,147,483,647. To define these data types, enter SMALLINT or INTEGER after the column name in the CREATE TABLE statement.

SMALLFLOAT and FLOAT Columns. These columns store floating-point numbers. SMALLFLOAT columns contain single-precision, floating-point numbers with approximately seven significant digits. FLOAT columns contain double-precision, floating-point numbers with approximately fourteen significant digits.

FLOAT columns do not store larger numbers; they store numbers with greater precision.

As mentioned previously, the number you enter into a floating-point column and the number INFORMIX-4GL stores may differ slightly, depending on how your computer handles floating-point numbers internally. Therefore, you should use SMALLFLOAT or FLOAT columns only if you have done so in other databases to maintain consistency, or if the system for which you are designing the database has limited disk space. SMALLFLOAT and FLOAT columns take up somewhat less disk space than DECIMAL columns.

To define these data types, enter SMALLFLOAT or FLOAT after the column name in the CREATE TABLE statement.

Note: On some small machines, SMALLFLOAT and FLOAT columns are implemented as DECIMAL(8) and DECIMAL(16) respectively. See Chapter 2 of the *INFORMIX-4GL Reference Manual* for details.

SERIAL Columns

INFORMIX-4GL automatically assigns a number to any column you specify as SERIAL. You do not need to enter data in a SERIAL column. Each time a new row is added to a table, INFORMIX-4GL assigns the next number in sequence to the SERIAL column. Normally, the starting number for the sequence is one, but you can select any number greater than zero as the starting number. Once INFORMIX-4GL assigns a SERIAL number, it cannot be changed. Each table in a database may contain only one SERIAL column.

To create a SERIAL column, enter SERIAL after the column name in a CREATE TABLE statement.

For example,

```
CREATE TABLE mytable (mycol SERIAL)
```

creates a table named **mytable** containing a SERIAL column named **mycol**. INFORMIX-4GL will automatically assign serial numbers, starting with the number one, to the **mycol** column as you add rows to the **mytable** table. You can specify any starting number greater than zero by enclosing the number in parentheses after the SERIAL keyword.

For example,

```
CREATE TABLE mytable (mycol SERIAL(10000))
```

designates 10,000 as the starting serial number for the **mycol** column. Do not enter a comma in the number you enclose in parentheses.

The highest serial number INFORMIX-4GL can assign is 2,147,483,647.

DATE Columns

Use the DATE data type for columns in which you want to store calendar dates. For example,

```
CREATE TABLE mytable (mydate DATE)
```

creates a table named **mytable** that contains a DATE column named **mydate**.

By default, a date in a DATE column has the format

month day year

When entering a value in a DATE column, you should separate each component by a non-numeric character, such as a period (.), hyphen (-), or slash (/). For the *month*, INFORMIX-4GL accepts a number value, such as 1 or 01 for January, 2 or 02 for February, and so on. Acceptable values for *day* are the numbers 1-31, corresponding to the days in the month.

Acceptable values for *year* are four-digit numbers between 0000 and 9999. If you enter only two digits for the year, as in 85 or 86, INFORMIX-4GL assumes the year is in the twentieth century and assigns the numbers 19 as the first two digits of the *year*.

Note: This format can be changed using the DBDATE environment variable. See Appendix B for more information.

MONEY Columns

MONEY columns store currency amounts. Values in these columns are displayed with a dollar sign and decimal point. To create a MONEY column, enter MONEY after the column name in the CREATE TABLE statement.

For example,

```
CREATE TABLE mytable (mymoney MONEY)
```

creates a table named **mytable** that contains a MONEY column called **mymoney**. As with DECIMAL columns, you can designate precision (total number of digits) and scale (number of digits to the right of the decimal point) for MONEY columns. For example, you can define the following MONEY column

```
CREATE TABLE mytable (mymoney MONEY(8,3))
```

where the column contains five digits to the left of the decimal point and three to the right, for a total of eight digits.

If you define a MONEY column with only one argument, such as

```
CREATE TABLE mytable (mymoney MONEY(8))
```

INFORMIX-4GL automatically defines the column as MONEY(8,2), meaning the column will have six digits to the left of the decimal point and two digits to the right. If you do not specify a precision or scale for a MONEY column, INFORMIX-4GL automatically defines the column as equal to DECIMAL(16,2). Unlike DECIMAL columns, however, MONEY columns always have fixed decimal points.

Examining a Sample Table

So far, you have seen how to create a database using the CREATE DATABASE and CREATE TABLE statements. Figure 2-8 shows a segment of the program that built the **stores** database and the **customer** table.

```
MAIN

CREATE DATABASE stores

CREATE TABLE customer
(customer_num       SERIAL(101),
 fname               CHAR(15),
 lname               CHAR(15),
 company             CHAR(20),
 address1            CHAR(20),
 address2            CHAR(20),
 city                CHAR(15),
 state               CHAR(2),
 zipcode             CHAR(5),
 phone               CHAR(18)
 )

END MAIN
```

Figure 2-9. The **customer** Table

As you can see, the CREATE DATABASE and CREATE TABLE statements are included within the main program.

All columns in the **customer** table have the CHAR data type except for **customer_num** (which is a SERIAL column). Although the **zipcode** column holds numbers, it is designated as a CHAR column instead of an INTEGER column so that INFORMIX-4GL will not drop leading zeros when it stores zip codes such as 01867.

Columns in other tables in the **stores** database use data types such as SMALLINT, MONEY, and DATE, as you will see in the following sections.

Determining Column and Row Lengths

Although INFORMIX-4GL does not impose a logical limit on the length of a row in a table, your system may have certain space limitations. Therefore, you should know how to compute the length, in *bytes*, of a column and row. A byte is equivalent to a single character.

Column Lengths. INFORMIX-4GL assigns lengths to columns as follows:

Data Type	Length
CHAR	Length specified in the CREATE TABLE statement
SMALLINT	2 bytes
INTEGER	4 bytes
SMALLFLOAT	4 bytes
FLOAT	8 bytes
SERIAL	4 bytes
DATE	4 bytes
DECIMAL	Depends upon scale (see below)
MONEY	Depends upon scale (see below)

Note: On some small machines, SMALLFLOAT and FLOAT are implemented as DECIMAL(8) and DECIMAL(16) respectively. See the *INFORMIX-4GL Reference Manual* for details.

Determining the Lengths of DECIMAL and MONEY Columns. The number of bytes required for a DECIMAL or MONEY column is half the total number of digits specified in the CREATE TABLE statement, plus 1. INFORMIX-4GL rounds fractional numbers up to the next whole number. For example, if you specify DECIMAL(14,4), the total storage required is eight bytes (14/2 + 1). However, if you specify DECIMAL(15,4), the total storage is nine bytes. (Since the result of the expression (15/2 + 1) is the fractional number 8.5, INFORMIX-4GL rounds up to the next whole number.)

Calculating Row Lengths. You can easily calculate the length of each row in a table. To do so, add the lengths for each column, following the guidelines in the previous table.

Figure 2-9 shows the CREATE TABLE statement for the **orders** table in the **stores** database. This table keeps track of the order number, order date, customer number, shipment date, and shipping instructions for each order.

```
CREATE TABLE orders
(order_num            SERIAL(1001),
 order_date           DATE,
 customer_num         INTEGER,
 ship_instruct        CHAR(40)
 backlog              CHAR(1),
 po_num               CHAR(10),
 ship_date            DATE,
 ship_weight          DECIMAL(8,2),
 ship_charge          MONEY(6),
 paid_date            DATE
)
```

Figure 2-10. The **orders** table

The following table shows how to calculate the total number of bytes used by a row:

Qty	Type	Length Each	Length Total
1	SERIAL	4 bytes	4 bytes
3	DATE	4 bytes	12 bytes
1	INTEGER	4 bytes	4 bytes
1	CHAR	40 bytes	40 bytes
1	CHAR	1 byte	1 byte
1	CHAR	10 bytes	10 bytes
1	DECIMAL	5 bytes (1 + 8/2)	5 bytes
1	MONEY	4 bytes (1 + 6/2)	4 bytes
		total length of each row:	80 bytes

Creating an Index

This section explains how to create an index. It describes the two types of indexes and suggests when indexes should be used. The section also explains how to use the CREATE INDEX statement in an INFORMIX-4GL program.

Types of Indexes

You can create an index that applies to a single column or to several columns as a group. These indexes are known as *single-column indexes* and *multiple-column indexes.*

Single-Column Indexes. These indexes make database query and sorting operations that involve the indexed column more efficient.

For example, if you frequently sort rows of customers by last name, you should index the last-name column. With an index, INFORMIX-4GL takes less time to sort names like this:

Last Name	First Name
Bender	George
Bender	Alice
Bender	Charles
Cranston	Philip
Cranston	Meredith
Cranston	Winston

Multiple-Column Indexes. You can also create an index that applies to more than one column. These multiple-column indexes speed up the process of sorting several columns.

For example, to sort the previous list so that first and last names are in alphabetical order, you could create an index that applies to both the last-name and first-name columns. Then INFORMIX-4GL could quickly sort the names as follows:

Last Name	First Name
Bender	Alice
Bender	Charles
Bender	George
Cranston	Meredith
Cranston	Philip
Cranston	Winston

Note: You could sort the list in this way even if the table did not have an index. An index simply speeds up the sorting process for a large database.

When to Index a Table

Before you create an index, determine whether the table requires one and, if so, which columns should be indexed. Following are guidelines for determining when to index columns:

- Do not create an index for a table until it contains several hundred rows.

- Create indexes only for certain columns, such as those frequently used for searching and sorting operations. For example, you would want to create an index on the last-name column in a table of names and addresses, since you usually sort mailing lists by last name.

- In general, create an index for any column that is frequently used to join tables.

- Do not index columns that contain a large number of duplicate values, such as Y (for yes) or N (for no). Here the index can significantly slow down the process of accessing values in the indexed column. Moreover, you cannot have more than 65,536 occurrences of the same value in an

indexed column—an important consideration if you are designing a large database.

- Do not create an index for a group of more than eight columns.

- Do not create an index for a column or group of columns whose total length exceeds 120 bytes.

When you create an index, it will take slightly longer to enter and modify data in a table, since **INFORMIX-4GL** has to update both the table and the index.

The CREATE INDEX Statement

Use the CREATE INDEX statement to create an index for one or more columns in a table. You must specify each index in a separate CREATE INDEX statement.

The simplest form of the CREATE INDEX statement is

```
CREATE INDEX index-name
   ON table-name (column-name [,...] )
```

where *index-name* is the name for the index, *table-name* is the table containing the column(s) to be indexed, and *column-name* is a column to be indexed.

Note: **INFORMIX-4GL** allows only one index on a particular column or group of columns. However, you can create one ascending index and one descending index on the same column or group of columns, since **INFORMIX-4GL** makes a distinction between them. See the section "Ascending and Descending Indexes" for more information.

For example, to speed up sorting operations on the **customer** table, you could create an index on the **lname** column using the following CREATE INDEX statement:

```
CREATE INDEX i_lname ON customer(lname)
```

In this example, INFORMIX-4GL creates an index called **i_lname** on the **lname** column of the **customer** table.

To create a multiple-column index, you could enter a series of column names separated by commas. For example,

```
CREATE INDEX i_name ON customer (lname, fname)
```

Here, INFORMIX-4GL creates the index **i_name** for the **lname** and **fname** columns in the **customer** table. In this case, the indexed values are sorted alphabetically by last name and then by first name within each last-name group.

Guidelines for Naming the Index

You must assign a name to every index you create. To make it easier to remember an index name, you may want to specify a name that is similar to the name of the indexed column.

Here are some guidelines to follow when naming an index:

- Give the index a name that does not exceed 18 characters.
- Begin the index name with a letter. The remaining characters may be letters, numbers, and underscores (_). INFORMIX-4GL does not distinguish between uppercase and lowercase letters in an index name.
- Make sure that index names are unique throughout the database. You cannot assign the same name to two indexes, even if they apply to different tables.
- Do not use an INFORMIX-4GL reserved word as an index name. (See Appendix C for a list of reserved words.)

The UNIQUE and DISTINCT Keywords

INFORMIX-4GL normally permits users to enter the same value in different rows of an indexed column. If necessary, you can prevent users from entering duplicate information into a column by using the UNIQUE or the DISTINCT keyword in a CREATE INDEX statement.

Use either of the following formats

```
CREATE UNIQUE INDEX index-name
  ON table-name (column-name [,...] )
```

or

```
CREATE DISTINCT INDEX index-name
  ON table-name (column-name [,...] )
```

where *index-name* is the name of the index, *table-name* is the table containing the column(s) to be indexed, and *column-name* is an indexed column that you want to be unique. You can use either the UNIQUE or DISTINCT keyword to prevent duplicate entries; both have the same effect.

If you want to prevent duplicate entries in the **social_sec** column of a table called **personnel**, you could use an CREATE INDEX statement like the following:

```
CREATE UNIQUE INDEX i_social_sec
       ON personnel(social_sec)
```

When you create this index, duplicate social security numbers cannot be entered into the **personnel** table.

Ascending and Descending Indexes

When INFORMIX-4GL creates an index, it sorts the values in the indexed column in ascending order, by default. For example, values in indexed CHAR columns are sorted from *A* to *Z*; values in numeric and MONEY columns are sorted from low to high; and values in DATE columns are sorted in chronological order.

However, you can also create an index with values sorted in descending order by using the following form of the CREATE INDEX statement

```
CREATE INDEX index-name
   ON table-name (column-name DESC [,...])
```

where DESC is a keyword.

Create descending indexes only for columns that will be sorted in descending order on a regular basis.

For example, if you frequently sort orders in reverse chronological order by order date, you might want to create an index for the **order_date** column of the **orders** table as follows:

```
CREATE INDEX i_order_date
        ON orders (order_date DESC)
```

This statement creates a descending index named **i_order_date** that applies to the **order_date** column in the **orders** table.

The Programmer's Environment

This section introduces the INFORMIX-4GL Programmer's Environment. This Programmer's Environment consists of a series of menus, which have been designed to support you through the program development process. In the INFORMIX-4GL Programmer's Environment you can

- Create and compile an INFORMIX-4GL program module.
- Create and compile a screen form.
- Create and link several modules in a multi-module program.
- Use the RDSQL query language if INFORMIX-SQL is installed on your system.

An INFORMIX-4GL program can consist of a single program module, which you create using the **Module** option of the INFORMIX-4GL Menu. Or, a program can be large, consisting of many modules that you link together using the **Program** option of the INFORMIX-4GL Menu.

In the following sections, you will use the **Module** option to create and compile a small program. Chapter 6 explains how to create a screen form using the **Form** option of the INFORMIX-4GL Menu. See the *INFORMIX-4GL Reference Manual* for information about creating multi-module programs and using the RDSQL query language.

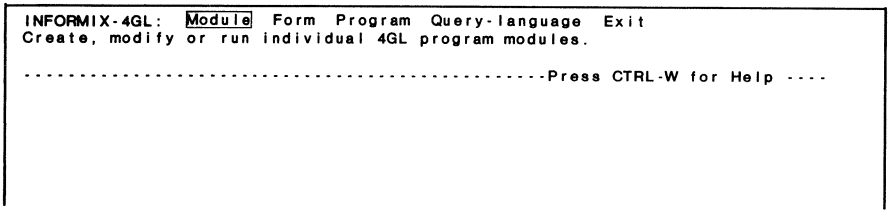
Note: If you are using a DOS system on MS-NET, there are certain procedures you must follow if you want to compile an INFORMIX-4GL program. See the section "Compiling and Executing Programs" in Chapter 1 of the *INFORMIX-4GL Reference Manual* for information on how to do this.

Entering the Programmer's Environment

To access the **INFORMIX-4GL** Programmer's Environment, enter the following from the operating system prompt:

```
i4gl
```

The INFORMIX-4GL Menu appears on the screen:



The INFORMIX-4GL Menu gives you five options:

Module	Work on **INFORMIX-4GL** program modules.
Form	Work on **INFORMIX-4GL** screen forms.
Program	Work on **INFORMIX-4GL** programs consisting of more than one module.
Query-language	Use the **RDSQL** query language if **INFORMIX-SQL** is installed on your system.
Exit	Leave the **INFORMIX-4GL** Programmer's Environment.

You select an option from the menu in either of two ways: by typing the first letter of the option, or by using the **SPACEBAR** or **ARROW** keys to move the cursor to the option you want and pressing **RETURN**.

A Program That Creates a Database

Program creation is essentially a two-step process. First you create a module with a text editor, and then you compile it using **INFORMIX-4GL**. You will want to use the **Module** option on the **INFORMIX-4GL** menu when you initially work on an **INFORMIX-4GL** program.

This section contains a brief tutorial that explains how to use the **Module** option of the INFORMIX-4GL Menu. The steps describe how to create and compile a program that builds a database. Figure 2-10 shows the program you will create in the tutorial.

```
MAIN
      #Creates a database of phone numbers

      CREATE DATABASE phones

      CREATE TABLE home_phones
      (fname CHAR(15),
       lname CHAR(15),
       phone CHAR(12)
      )

      CREATE INDEX i_lname
            ON home_phones(lname)

      GRANT CONNECT TO PUBLIC

END MAIN
```

Figure 2-11. The **phones** Program

Note: You will learn more about the statement GRANT CONNECT TO PUBLIC in Chapter 14.

Formatting Programs

You can use any format for writing INFORMIX-4GL programs. Since INFORMIX-4GL ignores blank lines and extra spaces, you can arrange statements in any way you like. For example, the following program is equivalent to the program in Figure 2-10 although it has a different format:

```
MAIN

  CREATE DATABASE phones  #creates database of phone numbers
  CREATE TABLE home_phones (fname CHAR(15),
    lname CHAR(15),
    phone CHAR(12))
  CREATE INDEX i_lname ON home_phones (lname)
  GRANT CONNECT TO PUBLIC

END MAIN
```

Using Comments

You can use comments at any place within an INFORMIX-4GL program. Following are two formats for comments:

Format 1.

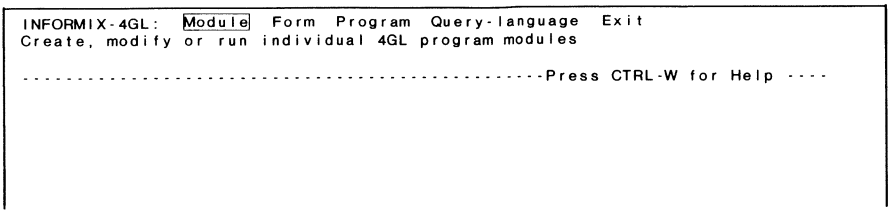
 # text where the pound sign # indicates the beginning of the comment, and *text* is a character string.

When INFORMIX-4GL encounters the pound sign, it ignores all characters after the pound sign to the end of the current line. You do not have to place comments on a separate line within the program.

Format 2.

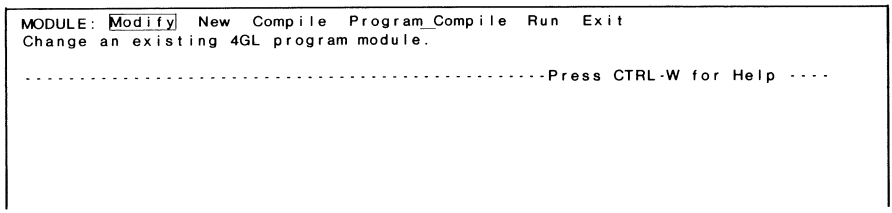
{text} where the left brace { indicates the beginning of the comment, *text* is a character string of any length, and the right brace } indicates the end of the comment. Since INFORMIX-4GL ignores everything between the braces, you can arrange the text in any way you choose.

Before continuing with the tutorial, make sure the INFORMIX-4GL Menu is displayed on the screen:



Now follow these steps.

1. Press **RETURN** to select the **Module** option. **INFORMIX-4GL** displays the MODULE Menu.



2. Type n, or move the cursor to **New** and press **RETURN**. **INFORMIX-4GL** displays the following screen:

```
NEW MODULE>>
Enter the name you want to assign to the module, then press Return.

----------------------------------------------Press CTRL-W for Help ----
```

3. Type phones and press **RETURN**. **INFORMIX-4GL** displays this screen:

```
USE-EDITOR>>vi
Enter editor name. (RETURN only for default editor).

----------------------------------------------Press CTRL-W for Help ----
```

4. Press **RETURN** to accept the default editor or type the name of another editor on your system and press **RETURN**. **INFORMIX-4GL** runs the selected editor.

   Note: If you have set the DBEDIT environment variable, **INFORMIX-4GL** automatically runs the text editor specified by DBEDIT without displaying the USE-EDITOR screen. Refer to Appendix B for more information about DBEDIT.

5. Enter the **phones** program shown in Figure 2-10. When you are finished, leave the editor. **INFORMIX-4GL** displays the following menu:

```
NEW MODULE:  Compile  Save-and-exit  Discard-and-exit
Compile the 4GL module specification.

-------------------------------------------------Press CTRL-W for Help ----
```

6. Press **RETURN** to select the **Compile** option and display the following menu:

```
COMPILE MODULE:  Object  Runable  Exit
Create object file only; no linking to occur.

-------------------------------------------------Press CTRL-W for Help ----
```

7. Type r, or move the cursor to **Runable** and press **RETURN**. **INFORMIX-4GL** compiles the **phones** program.

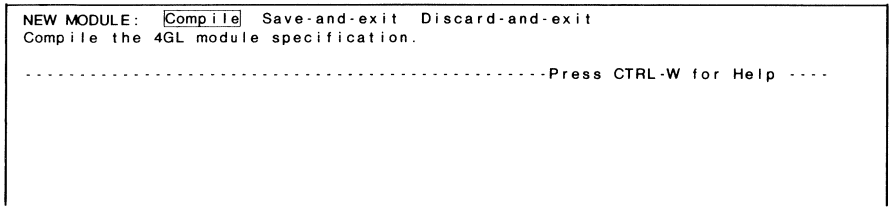
Correcting Errors. If the program has errors, **INFORMIX-4GL** displays the following menu:

```
COMPILE MODULE:  Correct  Exit
Correct errors in the 4GL module.

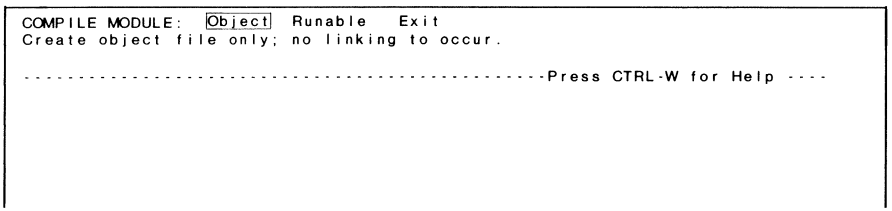
-------------------------------------------------Press CTRL-W for Help ----
```

1. Press **RETURN** to go back to the editor, which automatically displays the file containing your **INFORMIX-4GL** program. **INFORMIX-4GL** indicates errors with arrows and error messages.

2. Correct the errors indicated by the error messages. You do not need to remove the error messages. Then leave the text editor. **INFORMIX-4GL** displays the following menu:



3. Press **RETURN** to select the **Compile** option. **INFORMIX-4GL** displays the following menu:



4. Type r, or move the cursor to **Runable** and press **RETURN**. **INFORMIX-4GL** recompiles your program.

Saving the Program. If compilation is successful, **INFORMIX-4GL** displays the following menu:

```
NEW MODULE:  Compile  Save-and-exit  Discard-and-exit
Save the 4GL module and return to MODULE Menu.

------------------------------------------------Press CTRL-W for Help ----
```

1. Press **RETURN** to save the compiled program and redisplay the MODULE Menu.

```
MODULE: Modify  New  Compile  Program_Compile  Run  Exit
Execute an existing 4GL program module or application program.

------------------------------------------------Press CTRL-W for Help ----
```

2. Press **RETURN** to select the **Run** option. **INFORMIX-4GL** displays the following screen:

```
RUN PROGRAM>>
Choose a 4GL program with Arrow Keys, or enter a name, then press Return.

------------------------------------------------Press CTRL-W for Help ----
phones
```

3. Move the cursor to **phones**, if it is not there already, and press RETURN. INFORMIX-4GL runs the **phones** program, which creates the database, table, and index in the **phones.dbs** directory. After the program has finished running, INFORMIX-4GL displays the following message:

```
Hit Return to Continue
```

4. Press RETURN. INFORMIX-4GL displays the MODULE Menu.

5. Select the **Exit** option to return to the INFORMIX-4GL Menu.

6. Select the **Exit** option to return to the operating-system prompt.

If you now list the contents of your current directory, you will see the following files:

phones.4gl The INFORMIX-4GL source program, as indicated by the **.4gl** extension.

phones.4ge The executable program, as indicated by the **.4ge** extension.

phones.dbs The directory containing the table defined with the CREATE TABLE statement, the index defined with the CREATE INDEX statement, and other tables needed for database operations.

Notes: You should not run a program that creates a database more than once.

Do not create files within a database directory or try to alter any tables in a database directory at this time. Chapter 14 of this guide and the *INFORMIX-4GL Reference Manual* contain more information on modifying tables.

Chapter Summary

- A database is a collection of information that is useful to a particular organization, or that is used for a particular purpose.
- A database is comprised of tables, which organize data into rows and columns.
- A column in a table contains one particular type of information.
- A row in a table contains all the data about one of the objects in the table.
- By joining columns, you can access data in a number of tables as if it were all in one table.
- Indexes help INFORMIX-4GL retrieve and sort data in large databases more quickly.
- You use the MAIN statement to create a main program.
- You use the CREATE DATABASE statement to build a database.
- You use the CREATE TABLE statement to create a table. The CREATE TABLE statement lists the columns in the table as well as the type of data each column will store.
- You use the CREATE INDEX statement to define an index for a table. An index can apply to either a single column or multiple columns.
- You can create program modules and screen forms, link programs, and use the RDSQL query language in the INFORMIX-4GL Programmer's Environment.

Chapter 3

Working with a Database

Chapter 3 Table of Contents

Chapter Overview 5

Selecting a Database: The DATABASE Statement 6

Defining Program Variables 7
- The DEFINE Statement 7
  - Variable Names 7
  - Variable Types 8
- Sample DEFINE Statements 10

Assignment Statements: LET 12
- Expressions 12
  - Numeric Expressions 12
  - String Expressions 15
  - NULL Values 16
  - Boolean Expressions 18
  - Data-Type Conversions 19
  - Sample Assignment Statements 20

Interactive Programs 21
- Getting Input from the User 21
- Displaying Output: the DISPLAY Statement 23
  - Formatting 23

Basic Database Operations 25
- Inserting Rows into a Table 25
  - Adding Complete Rows to a Table 26
  - Adding Partial Rows to a Table 27
  - Using Program Variables in INSERT Statements 27
- Selecting Rows from a Table 28
  - Singleton SELECT Statements 28
  - Displaying the Results of a SELECT Statement 32
  - An Interactive Program for Selecting Data 34
- Updating Rows in a Table 35
- Deleting Rows from a Table 38

Chapter Summary 40

Chapter Overview

This chapter explains how to write INFORMIX-4GL programs to enter, retrieve, update, and delete information in a database.

The first part of this chapter introduces basic statements you will need for writing most INFORMIX-4GL programs. The discussion includes

- How to select a database
- How to define program variables
- How to assign values to variables
- How to write simple interactive programs

The second part of this chapter shows you how to perform basic database operations with INFORMIX-4GL and explains

- How to insert rows into a table
- How to select a row from a table
- How to update rows in a table
- How to delete rows from a table

For complete information about the statements you use to manipulate the information in a database, refer to the *INFORMIX-4GL Reference Manual.*

Selecting a Database: The DATABASE Statement

If you are writing a program that includes references to an existing database, you must use a DATABASE statement to select the existing database as the current database. In its simplest form, the DATABASE statement has the following format

DATABASE *database-name*

where *database-name* is the name of a database in the current directory or in a directory specified by the DBPATH environment variable.

Note: Do not include the extension **.dbs** when you specify the name of a database.

The DATABASE statement must appear before the MAIN statement if the first statement in the main program is a DEFINE statement that contains the LIKE keyword. (See the following section, "Defining Program Variables," for more information about DEFINE.) For example,

```
DATABASE stores

MAIN

  DEFINE last_name LIKE customer.lname
  . . .

END MAIN
```

Otherwise, the DATABASE statement can appear anywhere in a program, as long as it precedes any statement that makes reference to the current database.

Note: You do not need to use the DATABASE statement if your program makes no reference to an existing database.

Defining Program Variables

Variables represent places in the computer's memory for storing data used in a program. You can use variables to store values such as names, numbers, dates, and currency amounts.

Before using a variable in an **INFORMIX-4GL** program, you must *define* it by listing its name and type in a variable-declaration statement. Then you can assign a value to the variable and use it in **INFORMIX-4GL** expressions and statements.

The DEFINE Statement

You must use the DEFINE statement to define the variables in your program. The DEFINE statement has the following syntax

DEFINE *variable-name* [,...] *data-type*[,...]

where *variable-name* is the name of a program variable and *data-type* is the corresponding data type (for example, INTEGER, DATE, or MONEY). If you use a DEFINE statement in your program, it must occur immediately after the MAIN, FUNCTION, REPORT, or GLOBALS keyword. (See Chapter 5 for more information about FUNCTION and GLOBALS and Chapter 9 for more information about REPORT.)

Variable Names

You must follow a few guidelines in assigning variable names.

Number of Characters. Variable names may be from one to 18 characters long. You must make the first eight characters unique since **INFORMIX-4GL** regards only the first eight characters as significant. (For some compilers and linkers, only the first seven characters of global variables are significant. See Chapter 5 for more information about global variables.)

Characters You Can Use. The first character of a variable name must be a letter; the rest of the name may consist of letters, numbers, and underscores (_).

Choose meaningful names that indicate something about the purpose of the variable. Since INFORMIX-4GL makes no distinction between uppercase and lowercase letters, the variable name **acc_balance** is the same as the name **ACC_BALANCE**.

Reserved Words. You cannot use an INFORMIX-4GL reserved word as a variable name. (See Appendix C for a list of reserved words.)

Distinguishing Variable Names from Database, Table, and Column Names. If a program variable has the same name as a database, table, or column, INFORMIX-4GL assumes that the name refers to the program variable when it appears in an INFORMIX-4GL statement. For example, if you refer to **customer_num** in your program, INFORMIX-4GL assumes that it is a program variable instead of the **customer_num** column in the **customer** table. To refer to the database, table, or column name in such cases, put an "@" sign before the name.

For example,

```
variable:   customer_num
 column:    @customer_num

variable:   customer
   table:   @customer

variable:   stores
database:   @stores
```

Variable Types

You must assign a data type to each program variable. The data type indicates what kind of information you intend to store in the variable. You can assign any data type to a program variable that you can assign to a database column except SERIAL.

These data types are described in detail in Chapter 2 and are summarized below:

Data Types	Description
CHAR(*n*)	CHARacter variables store character strings of length *n*.
DECIMAL[(*m*[,*n*])]	Decimal variables store numbers with *m* (<=32) significant digits and *n* digits to the right of the decimal point.
SMALLINT	SMALLINT variables store whole numbers from −32,767 to +32,767.
INTEGER	Integer variables store whole numbers from −2,147,483,647 to +2,147,483,647.
SMALLFLOAT	Smallfloat variables store single-precision, floating-point numbers with approximately seven significant digits.
FLOAT	Float variables store double-precision, floating-point numbers with approximately 14 significant digits.
DATE	Date variables store character strings that contain calendar dates.
MONEY[(*m*[,*n*])]	Money variables store decimal numbers with a fixed number of digits to the right of the decimal point. MONEY(*m*)=DECIMAL(*m*,2). MONEY=DECIMAL(16,2).

In addition, you can assign a data type to a program variable by making reference to the data type of a database column with the LIKE keyword:

DEFINE *variable-name* [, ...] LIKE *table.column*

For example, if the column **fname** in the **customer** table has the data type char(15), you can define the program variable **cust_fname** to have the same data type with the following statement:

```
DEFINE cust_fname LIKE customer.fname
```

If the column has the data type SERIAL, INFORMIX-4GL assigns the variable the data type INTEGER and does not enforce any of the other restrictions associated with the SERIAL data type.

You can also define variables as records or arrays. (See Chapters 4 and 11 for more information about these data types.)

Sample DEFINE Statements

The following examples show how you might define program variables using the DEFINE statement.

Example 1.

```
DEFINE
    cust_name       CHAR(20),
    cust_balance    MONEY
```

This statement defines two program variables: **cust_name** with the data type CHAR(20) and **cust_balance** with the data type MONEY.

Example 2.

```
DEFINE
    cust_fname,
    cust_lname      CHAR(15),
    qty             SMALLINT
```

This statement defines three program variables: **cust_fname** with the data type CHAR(15), **cust_lname** with the data type CHAR(15), and **qty** with the data type SMALLINT.

Example 3.

```
DEFINE
    cust_lname      LIKE customer.lname,
    c_order_date,
    c_ship_date     DATE
```

This statement defines three program variables: **cust_lname** with the same data type as the column **customer.lname**, **c_order_date** with the data type DATE, and **c_ship_date** with the data type DATE.

Assignment Statements: LET

Once you have defined a program variable, you can assign a value to the variable with the LET statement:

LET *variable* = *expression*

When INFORMIX-4GL encounters a LET statement, it evaluates the expression to the right of the equal sign, and then assigns the calculated value to the program variable on the left.

Expressions

You can use numeric, string, and Boolean expressions in a LET statement. These different kinds of expressions are briefly described in the following sections.

Numeric Expressions

A numeric expression may consist of a numeric constant, program variable, column name, or function returning a numeric value, or any combination of these, connected by arithmetic operators.

Numeric Constants. A constant is a specific value. The representation of a numeric constant depends on the data type of the variable to which it is assigned. The following table shows one or more numeric constants you might assign to a variable of the given data type:

Numeric Type	Legal Constants
DECIMAL(6,3)	1.243 −256.28
SMALLINT	67
INTEGER	−133655
SMALLFLOAT	156.34 1.5634E2
FLOAT	2.67E-7

Program Variables. Before you use a program variable in a numeric expression, you must give it a value. In the following example, the variable **old_length** receives the value 3 before it appears in the expression old_length + 5:

```
DEFINE
    old_length,
    new_length          INTEGER

LET old_length = 3
LET new_length = old_length + 5
```

Numeric Columns and Functions. Column names can occur in expressions only in SELECT and UPDATE statements. See Chapter 4 and the section "Updating Rows in a Table," later in this chapter, for more information about using column names in expressions.

You can include a function in a numeric expression if it returns a numeric value. See Chapter 5 for more information about functions.

Arithmetic Operators. You can combine constants, program variables, column names, and functions in numeric expressions with the following operators:

Operator	Operation
**	Exponentiation
*	Multiplication
/	Division
mod	Modulus
+	Addition
−	Subtraction

When you use the division operator on two integers, **INFORMIX-4GL** ignores any remainder. For example, the expression `25/4` results in a value of `6`. When you use the modulus operator on two integers, **INFORMIX-4GL** returns the remainder after division. The expression `25 mod 4` results in a value of `1`.

When you use several operators in an expression, **INFORMIX-4GL** evaluates the expression according to operator precedence. In **INFORMIX-4GL**, exponentiation and parenthetical expressions take precedence over multiplication, division, and modulus, which, in turn, take precedence over addition and subtraction. Operators of equal precedence are evaluated from left to right.

Some examples follow:

Expression	Result
2 * 3 + 4	10
2 mod 5 − 1	1
5 − 3 + 1	3
5 − (3 + 1)	1
2 + 3 * 4	14

String Expressions

A string expression may consist of a string constant, a CHAR variable, a CHAR column, a function returning a CHAR expression, or any combination of these, connected by string operators.

String Constants and Variables. A string constant consists of zero or more characters enclosed in double quotes.

Some examples follow:

String Constants

```
"Enter a customer name: "
"12345"
" "
"David Smith"
", "
```

You can use a string constant in a LET statement to assign a value to a character variable as shown in the following example:

```
DEFINE last_name CHAR(15)
LET last_name = "Smith"
```

String Columns and Functions. Column names can occur in string expressions only in SELECT and UPDATE statements. You can include a function in a string expression if it returns a string value.

String Operators. You can use string constants, variables, columns, or functions in an expression with string operators. The string operators used in this guide are shown below:

Operator	Operation
,	concatenation
USING	formatting
CLIPPED	drop trailing blanks

You can use the concatenation operator to put two or more strings together. The following example shows how to use commas to concatenate three strings:

```
DEFINE cust_name CHAR(15)

LET cust_name = "John", " ", "Smith"
```

When **INFORMIX-4GL** encounters the LET statement in this example, it assigns the value John Smith to the character variable **cust_name**.

You can use the USING operator if you want a character string (or number) to conform to a pattern that you specify. (See Chapter 9 of this guide or the *INFORMIX-4GL Reference Manual* for more information about USING.)

Use the CLIPPED operator if you want to remove trailing blanks from a character string (see the section "Formatting," later in this chapter, for more information).

NULL Values

INFORMIX-4GL uses the NULL value to distinguish an unknown numeric value from zero and an unknown character value from one or more blanks. By default, **INFORMIX-4GL** assigns a NULL value to a column in a particular row of a table if that column does not contain a value.

You can assign a NULL value to a program variable using a LET statement. For example,

```
DEFINE total_price MONEY
LET total_price = NULL
```

If you use a NULL value in an arithmetic expression, the value of the entire expression is NULL. In the following example, the value of the arithmetic expression salary * last_raise is NULL since the value of last_raise is NULL:

```
LET last_raise = NULL

LET salary = 30000

LET new_salary = salary * last_raise
```

If a NULL value appears in a string expression, INFORMIX-4GL substitutes blanks for the value. For example, the value of the variable **full_name** in the following example is

```
Dr.                    Smith
```

since INFORMIX-4GL substitutes 15 blanks for the variable **fname**:

```
DEFINE fname, lname CHAR (15),
               full_name CHAR(35)

LET fname = NULL

LET lname = "Smith"

LET full_name = "Dr. ", fname, lname
```

If you use a NULL value in a Boolean expression that contains the IS NULL keywords, the value of the expression is either TRUE or FALSE. Otherwise, if you use a NULL value in a Boolean expression, the value of the expression is UNKNOWN and is treated as a FALSE condition in INFORMIX-4GL statements like IF, CASE, WHILE, SELECT, UPDATE, and

DELETE. (See the following section and the *INFORMIX-4GL Reference Manual* for more information about Boolean expressions and the IS NULL keywords.)

Note: If you want, you can create **INFORMIX-4GL** applications that do not test for NULL values and do not use the three-value logic described in the following section, "Boolean Expressions." (See Appendix I of the *INFORMIX-4GL Reference Manual* for details.)

Boolean Expressions

A Boolean expression evaluates to TRUE (defined as 1), FALSE (defined as 0), or UNKNOWN. Here are some examples of Boolean expressions that evaluate to TRUE or FALSE:

Expression	Value
`(2+5)*3 = 18`	FALSE
`14 <= 16`	TRUE
`"James" = "Jones"`	FALSE

If one of the values in a Boolean expression is NULL, the entire expression is UNKNOWN. In the following example, the value of the Boolean expression is UNKNOWN if the value of **last_raise** is NULL:

```
salary * last_raise < 25000
```

Combining Boolean expressions using the operators AND, OR, and NOT produces the following results (where T corresponds to TRUE, F corresponds to FALSE, and ? corresponds to UNKNOWN):

AND	T	F	?
T	T	F	?
F	F	F	F
?	?	F	?

OR	T	F	?
T	T	T	T
F	T	F	?
?	T	?	?

NOT	
T	F
F	T
?	?

Figure 3-1. Combining Boolean Expressions

You can use Boolean expressions in **INFORMIX-4GL** statements such as IF, CASE, WHILE, SELECT, UPDATE, and DELETE. See Chapter 4 and the section "Basic Database Operations," later in this chapter, for more information about using Boolean expressions in the IF, SELECT, UPDATE, and DELETE statements. See Chapter 7 for more information about using Boolean expressions in the WHILE statement. Refer to the *INFORMIX-4GL Reference Manual* for complete information about Boolean expressions.

Data-Type Conversions

Whenever possible, **INFORMIX-4GL** automatically converts an expression to the data type of the program variable to which it is assigned. This means that **INFORMIX-4GL** can convert one numeric data type to another, convert a numeric data type to a character data type, or even convert a character data type to a numeric data type if the CHAR expression contains characters that are valid as numbers.

Here are some examples:

```
DEFINE dec_type      DECIMAL(6,2),
       char_type     CHAR(10),
       num_type      INTEGER

LET dec_type = 5

LET char_type = 94301

LET num_type = "12"
```

Sample Assignment Statements

The following examples show how to use different kinds of expressions in a LET statement.

Example 1.

```
DEFINE
    base,
    height,
    tri_area    DECIMAL

LET base = 3
LET height = 4
LET tri_area = (base*height)/2
```

In this example, the expression (base*height)/2 results in a value of 6.

Example 2.

```
DEFINE counter INTEGER

LET counter = 1
LET counter = counter + 1
```

In this example, INFORMIX-4GL initially assigns the value 1 to the program variable **counter.** Then INFORMIX-4GL adds 1 to the current value of **counter** and assigns the resulting value, 2, to the same program variable.

Example 3.

```
DEFINE
    lname        CHAR(15),
    full_name    CHAR(30)

LET lname = "Smith"
LET full_name = "David ", lname
```

In this example, INFORMIX-4GL assigns the value Smith to the variable **lname**. Then INFORMIX-4GL concatenates the value of **lname** to a string constant and assigns the result to the variable **full_name**.

Interactive Programs

Most database applications are *interactive* because they accept input from the user, process the input, and then print the output on the screen, to a file, or on a printer. The following section describes how to prompt for input and display output on the screen using INFORMIX-4GL.

Getting Input from the User

You can have the user supply a value for a variable with the PROMPT statement. The syntax for this statement is

PROMPT *display-list* FOR *variable*

where *display-list* is a list of one or more variables or constants, separated by commas, and *variable* stores the value entered by the user.

When INFORMIX-4GL encounters a PROMPT statement, it prints the values in the *display-list* on the screen, waits for the user to enter a value, and then reads the value into the *variable*. In a simple PROMPT statement, the *display-list* may consist of a single string constant, enclosed in double quotes, as shown below:

```
DEFINE cust_num INTEGER

PROMPT "Enter a customer number: "
   FOR cust_num
```

In this example, INFORMIX-4GL displays the prompt

```
Enter a customer number:
```

and waits for the user to enter an integer, followed by a RETURN, for the program variable **cust_num**.

In a more complex PROMPT statement, the *display-list* might contain both program variables and string constants:

```
DEFINE
       cust_num          INTEGER,
       cust_name         CHAR(15)

LET cust_name = "David Smith"

PROMPT "Enter the customer number for ",
       cust_name, ": " FOR cust_num
```

In this example, INFORMIX-4GL displays the following prompt and waits for input from the user:

```
Enter the customer number for David Smith     :
```

Displaying Output: the DISPLAY Statement

You can use the DISPLAY statement to display information on the screen in a format you specify. The DISPLAY statement has the following format

DISPLAY *display-list*

where *display-list* is a list of one or more variables or constants separated by commas. Each item in the display list may include an optional format clause such as CLIPPED or USING.

When **INFORMIX-4GL** encounters a DISPLAY statement, it prints the values of the expressions on the screen according to the specified format, if any. Each DISPLAY statement begins on a new line. In a simple DISPLAY statement, the *expression* may consist of a string constant, enclosed in double quotes:

```
DISPLAY "Updating customer records"
```

When **INFORMIX-4GL** encounters this statement, it displays the message Updating customer records on the screen.

A more complex DISPLAY statement might include string constants and variables, separated by commas:

```
LET cust_name = "David Smith"

DISPLAY "Deleting information on ", cust_name
```

In this example, **INFORMIX-4GL** displays the message Deleting information on David Smith.

Formatting

You can use the CLIPPED and USING keywords as well as the COLUMN function to control the format of your DISPLAY statements. This section describes the CLIPPED keyword. (See Chapter 9 of this guide and the *INFORMIX-4GL Reference Manual* for information on USING and COLUMN.)

CLIPPED. When you print a CHAR variable, **INFORMIX-4GL** adds trailing blanks (if necessary) to the value of the variable until the total number of characters in the string equals the number of characters you specified in the variable definition.

For example,

```
DEFINE cust_fname, cust_lname CHAR(15)

LET cust_fname = "John"
LET cust_lname = "Smith"

DISPLAY cust_fname, cust_lname, "has placed an order."
```

When **INFORMIX-4GL** encounters the DISPLAY statement in this example, it displays 15 characters for **cust_fname** and 15 characters for **cust_lname:**

```
John            Smith           has placed an order.
```

If you want to print a CHAR variable without the trailing spaces, use the CLIPPED keyword after the variable, as follows:

```
DEFINE cust_fname, cust_lname CHAR(15)

LET cust_fname = "John"

LET cust_lname = "Smith"

DISPLAY cust_fname CLIPPED, " ",
    cust_lname CLIPPED,
    " has placed an order."
```

This example produces the following output:

```
John Smith has placed an order.
```

Basic Database Operations

You now have most of the tools that you will need to write interactive programs that perform basic database operations. The rest of this chapter explains how to use the INSERT, SELECT, UPDATE, and DELETE statements to work with data in the **stores** database.

Inserting Rows into a Table

You can use the INSERT statement to add a row to an existing table. The INSERT statement has the following syntax:

```
INSERT INTO table-name [(column-list)]
   VALUES (value-list)
```

You must follow a few guidelines to insert a row into a table using **INFORMIX-4GL**:

- Follow the INSERT INTO keywords with the name of the table.

- Entering column names is optional. If you enter column names, enclose them in parentheses and separate them with commas. If you omit the column names, you must include a value for every column and list the values in the order that the columns appear in the table.

- Follow the VALUES keyword with constants and/or program variables containing the data you want to add. Include a comma between items and enclose the list in parentheses. Make sure that the items in the *value-list* correspond to the columns in the *column-list*.

- If you use CHAR or DATE constants after the VALUES keyword, make sure you put double quotes around each constant.

Adding Complete Rows to a Table

If you want to add a value to every column in the table, you can use an INSERT statement that lists all the column names or one that omits the column names entirely. For example, suppose you want to add a value for each column in the **customer** table:

customer_num	0 (instructs RDSQL to select the next serial number)
fname	Enid
lname	Smythe
company	Smythe Sports
address1	Bay Road
address2	P. O. Box 111
city	Palo Alto
state	CA
zipcode	94303
phone	415-329-2222

To add this data to the table, you could use either of the INSERT statements that follow:

```
INSERT INTO customer
        (customer_num, fname, lname, company,
         address1, address2,
          city, state, zipcode, phone)
    VALUES
        (0, "Enid", "Smythe", "Smythe Sports",
         "Bay Road", "P. O. Box 111",
         "Palo Alto", "CA", "94303", "415-329-2222")
```

or

```
INSERT INTO customer VALUES
    (0, "Enid", "Smythe", "Smythe Sports",
     "Bay Road", "P. O. Box 111",
     "Palo Alto", "CA", "94303", "415-329-2222")
```

Note: You can specify a non-NULL value for a SERIAL column if the value does not duplicate other numbers in the column. It is easier, however, to enter a zero (0) as a place holder for the SERIAL value and let INFORMIX-4GL add the proper value (as shown above).

Adding Partial Rows to a Table

You can also use the INSERT statement to insert partial rows into a table. Here are some examples:

```
INSERT INTO customer
      (customer_num, fname, lname, company)
   VALUES
      (0, "Enid", "Smythe", "Smythe Sports")

INSERT INTO customer
      (customer_num, company, address1, city, state, zipcode)
   VALUES
      (0, "Pro Sports Shop", "665 Addison", "Palo Alto", "CA", "94303")
```

Note: **INFORMIX-4GL** assigns NULL values to any columns that you do not specify in the INSERT statement.

Using Program Variables in INSERT Statements

Often, you will want to use program variables as well as constants in an INSERT statement. You should make sure the variables have values before using them in the VALUES clause of an INSERT statement. Here is an example:

```
LET cust_fname = "Enid"

LET cust_lname = "Smythe"

LET cust_company = "Smythe Sports"

INSERT INTO customer
     (customer_num, fname, lname, company)
   VALUES
     (0, cust_fname, cust_lname, cust_company)
```

If you want, you can have the user enter values for the variables as in the following example:

```
PROMPT "Enter the customer's first name: " FOR cust_fname

PROMPT "Enter the customer's last name: " FOR cust_lname

PROMPT "Enter the customer's firm: " FOR cust_company

INSERT INTO customer
      (customer_num, fname, lname, company)
   VALUES
      (0, cust_fname, cust_lname, cust_company)
```

Selecting Rows from a Table

You can use the SELECT statement to retrieve information from a database. The SELECT statement is the basis for all queries in **INFORMIX-4GL**, and the results of a SELECT statement may be used in reports, screen displays, and more. The SELECT statement has eight clauses:

- SELECT clause
- INTO clause
- FROM clause
- WHERE clause
- GROUP BY clause
- HAVING clause
- ORDER BY clause
- INTO TEMP clause

This section describes how to use the first four clauses in the list. (See Chapter 4 of this guide and the *INFORMIX-4GL Reference Manual* for information about the other clauses.)

Singleton SELECT Statements

You can use the SELECT statement to retrieve one, several, or all rows in a table. This chapter explains how to write *singleton* SELECT statements that retrieve single rows from a table; the next chapter explains how to select several rows from one or more tables with a SELECT statement and a *cursor*.

The following shows the form of the SELECT statement that retrieves a single row:

SELECT *select-list* INTO *variable-list*
FROM *table-list* WHERE *search-conditions*

Although you can create a valid SELECT statement using only the SELECT and FROM clauses, you cannot create a SELECT statement that retrieves a single row unless you also include the INTO and WHERE clauses. These clauses are described in the following sections.

The SELECT Clause. You can use the SELECT clause to specify the columns containing the values you want to retrieve. It has the following syntax

SELECT *select-list*

where *select-list* is a list of column names (and/or expressions) separated by commas.

If you want to select all the columns in a table, you may use an asterisk (*) instead of listing each column. Here are some examples:

```
SELECT *

SELECT customer_num, fname, lname

SELECT lname, company, city, state
```

Note: Since the SELECT statement requires both a SELECT clause and a FROM clause, the previous examples do not represent valid SELECT statements. Rather, they show various forms that the SELECT clause can take.

The INTO Clause. You must use an INTO clause in a singleton SELECT statement to specify the program variables that will store the data retrieved by the SELECT statement. The INTO clause has the following format

INTO *variable-list*

where *variable-list* is a list of program variables that correspond to the columns in the *select-list.*

You should make sure that the data type of each program variable is the same as (or compatible with) the data type of the corresponding column in the select-list. Often, it is a good idea to define the program variables with the LIKE keyword to ensure type compatibility between variables and database columns. For example,

```
DEFINE
        m_code char(3),
        m_name LIKE manufact.manu_name,
        cust_fname LIKE customer.fname,
        cust_lname LIKE customer.lname,
        cust_company LIKE customer.company

SELECT manu_code, manu_name INTO m_code, m_name

SELECT fname, lname, company
        INTO cust_fname, cust_lname, cust_company
```

Note: Since the SELECT statement requires both a SELECT clause and a FROM clause, the previous examples do not represent valid SELECT statements.

The FROM Clause. You can use the FROM clause to specify the table from which you want to select data. The format follows

FROM *table-list*

where *table-list* is a list of tables separated by commas.

Here are some examples:

```
DEFINE
     m_code LIKE manufact.manu_code,
     m_name LIKE manufact.manu_name,
     cust_fname LIKE customer.fname,
     cust_lname LIKE customer.lname,
     cust_company LIKE customer.company

SELECT manu_code, manu_name
     INTO m_code, m_name
     FROM manufact

SELECT fname, lname, company
     INTO cust_fname, cust_lname, cust_company
     FROM customer
```

Note: The previous examples *do* represent valid SELECT statements, since they contain both the SELECT and FROM clauses. However, since the sample SELECT statements *do not* have WHERE clauses, they extract data from all rows in the named table and will require special handling (see Chapter 4 for more information).

The WHERE Clause. To retrieve one row at a time using a SELECT statement, you must include a WHERE clause that uniquely describes the row containing the data you want to select. The general format is

WHERE *search-condition*

You can add a WHERE clause with a search condition to each of the preceding SELECT statements so that each SELECT statement retrieves a single row:

```
DEFINE
  m_code LIKE manufact.manu_code,
  m_name LIKE manufact.manu_name

SELECT manu_code, manu_name INTO m_code, m_name
  FROM manufact
  WHERE manu_code = "SMT"
```

This statement selects the row in the **manufact** table that has a value of SMT in the **manu_code** column (assuming that the values in the **manu_code** column are unique).

```
DEFINE
  cust_fname LIKE customer.fname,
  cust_lname LIKE customer.lname,
  cust_company LIKE customer.company

SELECT fname, lname, company
  INTO cust_fname, cust_lname, cust_company
  FROM customer
  WHERE lname = "Grant"
```

This statement selects the row in the **customer** table that has the name Grant in the **lname** column (assuming that the **lname** column contains no duplicate names).

Displaying the Results of a SELECT Statement

The SELECT statement only retrieves data from the database and stores it in program variables at your request—it does not display or print results automatically. To view the results of a SELECT statement on screen, you need to print the data in the program variables using the DISPLAY statement described earlier in the section "Displaying Output."

Here is an example:

```
DEFINE
     cust_fname LIKE customer.fname,
     cust_lname LIKE customer.lname,
     cust_company LIKE customer.company,
     cust_add1 LIKE customer.address1,
     cust_add2 LIKE customer.address2,
     cust_city LIKE customer.city,
     cust_state LIKE customer.state,
     cust_zipcode LIKE customer.zipcode

SELECT fname, lname, company,
       address1, address2,
       city, state, zipcode
     INTO cust_fname, cust_lname, cust_company,
       cust_add1, cust_add2,
       cust_city, cust_state, cust_zipcode
     FROM customer
     WHERE lname = "Smythe"

DISPLAY cust_fname CLIPPED, " ", cust_lname
DISPLAY cust_company
DISPLAY cust_add1
DISPLAY cust_add2
DISPLAY cust_city CLIPPED, ", ",
     cust_state CLIPPED, "  ", cust_zipcode
```

These statements produce a display like the following:

```
Enid Smythe
Smythe Sports
Bay Road
P. O. Box 111
Palo Alto, CA  94303
```

An Interactive Program for Selecting Data

You now have enough information to write a simple interactive program that retrieves data from the **customer** table. The following sample program asks the user to enter a customer number, selects the row with the specified customer number, and displays the row on screen:

```
DATABASE stores

MAIN

DEFINE
     cust_num LIKE customer.customer_num,
     cust_fname LIKE customer.fname,
     cust_lname LIKE customer.lname,
     cust_company LIKE customer.company,
     cust_add1 LIKE customer.address1,
     cust_add2 LIKE customer.address2,
     cust_city LIKE customer.city,
     cust_state LIKE customer.state,
     cust_zipcode LIKE customer.zipcode

PROMPT "Enter a customer number: " FOR cust_num

SELECT fname, lname, company,
          address1, address2,
          city, state, zipcode
     INTO cust_fname, cust_lname, cust_company,
          cust_add1, cust_add2,
          cust_city, cust_state, cust_zipcode
     FROM customer
     WHERE customer_num = cust_num

DISPLAY cust_fname CLIPPED, " ", cust_lname
DISPLAY cust_company
DISPLAY cust_add1
DISPLAY cust_add2
DISPLAY cust_city CLIPPED, ", ",
     cust_state CLIPPED, "  ", cust_zipcode

END MAIN
```

If the user enters the customer number 103, INFORMIX-4GL will display the following information:

```
Philip Currie
Phil's Sports
654 Poplar
P.O. Box 3498
Palo Alto, CA  94303
```

Updating Rows in a Table

You can change the data in a table by using the UPDATE statement to set a column equal to a value. The simplest format for an UPDATE statement is

```
UPDATE table-name
   SET column-name = expr [, ...]
   [WHERE clause]
```

where *table-name* is the name of the table that contains the rows you want to update; *column-name* is the name of a column you want to update; and *expr* is a constant, program variable, column name, or any combination of the above.

You specify which row or rows you want to change by including a WHERE clause.

To change data in a table with the UPDATE statement, you must follow a few guidelines:

- Follow the UPDATE keyword with the name of the table.
- Follow the SET keyword with the name of a column, an equal sign, and the value you want to enter for that column. Make sure that the data type of the value to the right of the equal sign is compatible with the data type of the column on the left.
- If you want to change data in more than one column, follow each column name with an equal sign and the value you want to enter. Separate column entries with commas. Type SET for the first column entry only.

- Enclose CHAR and DATE data in quotation marks.

- Use a WHERE clause to select the row or rows you want to update. If you leave out the WHERE clause, **INFORMIX-4GL** assumes you want to change *all* rows.

Here are some examples:

Example 1.

```
UPDATE stock
  SET unit_price = unit_price * 1.05
```

This statement increases the value of **unit_price** by five percent in every row of the **stock** table.

Example 2.

```
UPDATE stock
  SET unit_price = unit_price * 1.05
  WHERE unit_price < 20
```

This statement adds five percent to the value of **unit_price** only in rows where the current unit price is less than 20 dollars.

Example 3.

```
UPDATE customer
  SET company = "Phil's Sports Shop",
    zipcode = "94301"
  WHERE customer_num = 103
```

This statement changes the **company** to Phil's Sports Shop and the **zipcode** to 94301 in the row where **customer_num** is equal to 103.

The UPDATE statement has several formats. The following form of the UPDATE statement lists all the column names to the left of the equal sign and all the expressions to the right of the equal sign:

```
UPDATE table-name
   SET (column-list) = (expression-list)
   [WHERE clause]
```

If you use this format, separate list items with commas and make sure that the expressions in *expression-list* correspond to the column names in *column-list*. For example, the following statement produces the same result as the UPDATE statement in Example 3.

```
UPDATE customer
   SET (company, zipcode) =
                     ("Phil's Sports Shop", "94301")
   WHERE customer_num = 103
```

Note: To update all the columns in one or more rows of a table, you can use the following form of the UPDATE statement:

```
UPDATE table-name
   SET table-name.* = record-name.*
   [WHERE clause]
```

where *table-name.** refers to all columns in the named table and *record-name.** refers to all the components in a program record. See Chapter 7 of the *INFORMIX-4GL Reference Manual* for more information about this form of the UPDATE statement. See Chapter 4 of this guide for more information about variables of type RECORD.

Deleting Rows from a Table

You can use the DELETE statement to delete one or more rows from a table. The general format for a DELETE statement is

DELETE FROM *table-name* [WHERE clause]

where *table-name* is the name of the table containing the row(s) you want to remove.

Here are some guidelines you should follow to delete one or more rows:

- Follow the DELETE FROM keywords with the name of the table from which you want to delete row(s).
- Use a WHERE clause to specify the row or rows you want to delete. If you leave out the WHERE clause, **INFORMIX-4GL** assumes you want to delete *all* rows in the table.

Note: You cannot delete part of a row; if you want to delete the data in some of the columns in a row, you should use the UPDATE statement.

Here are some examples:

Example 1.

```
DELETE FROM customer
  WHERE lname = "Smythe"
```

This statement deletes the row for Enid Smythe in the **customer** table.

Example 2.

```
DELETE FROM orders
  WHERE order_date < "6/1/1986"
```

This statement deletes the rows from the **orders** table that have an order date earlier than 6/1/1986.

Example 3.

```
DELETE FROM stock
  WHERE manu_code = "SMT"
```

This statement deletes the rows from the **stock** table that have the value SMT in the **manu_code** column.

Chapter Summary

- You can use INFORMIX-4GL to write interactive programs to enter, retrieve, update, and delete information in a database.
- You use the DEFINE statement to define the variables used in your program.
- You can assign any data type to a program variable that you can assign to a database column except SERIAL. You can also assign a data type to a variable by making reference to the data type of a database column with the LIKE keyword.
- You use the LET statement to assign the value of an expression to a variable.
- You can write interactive programs using the PROMPT and DISPLAY statements. The PROMPT statement lets you obtain a value for a variable from the user. The DISPLAY statement displays information on the terminal screen.
- You can add rows to a table with the INSERT statement.
- You can extract a row from a table with a SELECT statement. A SELECT statement that retrieves one row at a time must have a SELECT, INTO, FROM, and WHERE clause. You can use the DISPLAY statement to print the results of a SELECT statement.
- You can change the information in a table with the UPDATE statement.
- You can remove rows from a table with the DELETE statement.

Chapter 4

The SELECT Statement

Chapter 4 Table of Contents

Chapter Overview 5

Defining Records 6
  Using a Record in a SELECT Statement 7

SELECT Statements That Retrieve More than One Row 10
  Assigning a Cursor to a SELECT Statement 10
  Retrieving and Processing Rows 11
    The FOREACH Statement 11
    The OPEN, FETCH, and CLOSE Statements 16
    The WHILE Statement 21
    More About the FETCH Statement 24

Complex SELECT Statements 27
  More About the SELECT Clause 27
    Eliminating Duplicate Rows 27
    Arithmetic Operators 28
    Math Functions 28
  More About the WHERE Clause 29
    Relational Operators 30
    Using Several Search Conditions: the AND and OR Keywords 31
    Ranges: the BETWEEN and AND Keywords 31
    Sets: the IN Keyword 32
    Pattern Matching: the MATCHES Keyword 33
  The ORDER BY Clause 34
    Sorting by More than One Column 35
  Selecting Data from Several Tables 36

Chapter Summary 39

Chapter Overview

This chapter explains how to use the SELECT statement with a cursor to retrieve groups of rows from a database containing one or more tables. The discussion includes

- How to define records
- How to declare a cursor for a SELECT statement
- How to retrieve rows with the FOREACH statement
- How to retrieve rows with the OPEN, FETCH, and CLOSE statements
- How to write complex SELECT statements

For complete information about the commands and statements you use to select rows from a database, refer to the *INFORMIX-4GL Reference Manual.*

Defining Records

In most database applications, you frequently need to transfer data between program variables and your database with the SELECT statement and other **INFORMIX-4GL** statements. Often, you will find it more convenient to perform such transfers using *records* rather than the simple variables described earlier in Chapter 3.

A record is a collection of variables of possibly differing data types. You can work with the variables in the record one by one or as a group depending on program requirements. Before you use a record, you must define it by using the DEFINE statement with the RECORD keyword. The format is

```
DEFINE variable-list
   RECORD
      variable-list data-type [, ...]
   END RECORD
```

where *variable-list* is a list of one or more program variables separated by commas and *data-type* is one of the data types described in Chapter 3.

You refer to the entire set of variables in the order in which they were defined by using the following notation:

```
record–name . *
```

You refer to an individual variable in the record by using this notation:

```
record-name.variable-name
```

If you want to refer to a consecutive group of variables in a record, you can use the following notation:

record_name.variable_name1 THRU *record_name.variable_name2*

The following example shows how to declare a record using the DEFINE statement:

```
DEFINE cust_address
       RECORD
            company,
            addr1,
            addr2    CHAR(20),
            city     CHAR(15),
            state    CHAR(2),
            zipcode  CHAR(5)
       END RECORD
```

This statement defines a record called **cust_address** that has six components: **company**, **addr1**, **addr2**, **city**, **state**, and **zipcode**. In an INFORMIX-4GL statement, you would use the notation **cust_address.*** to refer to all the variables in the **cust_address** record, and notations like **cust_address.company** and **cust_address.city** to refer to individual variables in the record. If you want to refer to the subset **cust_address.city**, **cust_address.state**, and **cust_address.zipcode**, you could use a notation like the following:

```
cust_address.city THRU cust_address.zipcode
```

Using a Record in a SELECT Statement

You can use a record after the INTO keyword of a SELECT statement to conveniently copy information from database columns to program variables. You must make sure that the columns in the SELECT clause appear in the same order as the variables in the record. You should also make sure that the data type of the column and the data type of the corresponding record variable are identical or compatible.

In the following example, the **cust_address** record stores data selected from the **customer** table:

```
DEFINE cust_address
    RECORD
        company,
        addr1,
        addr2    CHAR(20),
        city     CHAR(15),
        state    CHAR(2),
        zipcode  CHAR(5)
    END RECORD

SELECT company, address1, address2, city, state, zipcode
    INTO cust_address.*
    FROM customer
    WHERE customer_num = 115
```

This SELECT statement retrieves information from the **company**, **address1**, **address2**, **city**, **state**, and **zipcode** columns and stores it in the variables **cust_address.company**, **cust_address.addr1**, **cust_address.addr2**, **cust_address.city**, **cust_address.state**, and **cust_address.zipcode**.

Similarly, the following statement retrieves information from the **city**, **state**, and **zipcode** columns and stores it in the variables **cust_address.city**, **cust_address.state**, and **cust_address.zipcode**:

```
SELECT city, state, zipcode
    INTO cust_address.city THRU cust_address.zipcode
    FROM customer
    WHERE customer_num = 115
```

You can store an entire row in a record that contains a variable for each column in a table. For example, the following record contains one variable for each column in the **customer** table:

```
DEFINE p_customer
  RECORD
    customer_num    integer,
    fname,
    lname           CHAR(15),
    company,
    address1,
    address2        CHAR(20),
    city            CHAR(15),
    state           CHAR(2),
    zipcode         CHAR(5),
    phone           CHAR(18)
  END RECORD
```

You can define the same record more easily, however, by using the DEFINE statement with the RECORD LIKE keywords:

```
DEFINE p_customer RECORD LIKE customer.*
```

This statement defines a record **p_customer** that has variables of the same name and data type as the columns in the **customer** table.

Note: If you define a record LIKE a table, make sure that the first eight characters of each column name in the table are unique.

After you define the record **p_customer** using either method shown above, you can use it after the INTO keyword in a SELECT statement to retrieve and store an entire row. For example,

```
SELECT * INTO p_customer.* FROM customer
  WHERE customer_num = 112
```

SELECT Statements That Retrieve More than One Row

Up to this point you have worked with singleton SELECT statements that retrieve one row at a time. If you want to write a SELECT statement that retrieves several rows at once, you will need to declare a *cursor* for the SELECT statement. You may think of a cursor as a marker that points to a row retrieved by a SELECT statement. With a cursor and cursor-control statements, you can work sequentially or non-sequentially through the set of rows specified by a SELECT statement. This set of rows is called the *active set.*

Assigning a Cursor to a SELECT Statement

You use the DECLARE statement to assign a cursor to a SELECT statement. You may declare a *non-scrolling* cursor that allows rows to be retrieved from the active set in sequential order, or declare a *scrolling* cursor that allows rows to be retrieved in any order you choose. The DECLARE statement has the following format

```
DECLARE cursor-name [SCROLL] CURSOR FOR
  SELECT-statement
```

where *cursor-name* is the name of a cursor for the given SELECT statement and SCROLL is an optional keyword that indicates a scrolling cursor. *cursor-name* must be unique to the source file in which it is declared.

For example, if you want to select all the rows from the **customer** table and process them in consecutive order, you can define a non-scrolling cursor using a DECLARE statement like the following:

```
DECLARE q_curs CURSOR FOR
  SELECT * FROM customer
```

If you want to retrieve rows for customers in California and process them in random order, use the following DECLARE statement:

```
DECLARE my_curs SCROLL CURSOR FOR
    SELECT * FROM customer
    WHERE state = "CA"
```

Note: Do not declare a scrolling cursor unless you need to process rows in random order, since it requires additional "overhead." See the section "More About the FETCH Statement" later in this chapter, for information about scrolling cursors.

Retrieving and Processing Rows

Once you have declared a cursor for a SELECT statement, you can use either the FOREACH statement or the OPEN, FETCH, and CLOSE statements to retrieve and process rows. The following sections describe both methods.

The FOREACH Statement

Once you have declared a cursor for a SELECT statement, you can select the rows and execute a sequence of statements for each row returned by the query using the FOREACH statement. Although the FOREACH statement works with either a non-scrolling or scrolling cursor, it is designed to process rows only in sequential order. The FOREACH statement has the following general form:

```
FOREACH cursor-name [INTO variable-list]
    statement
    ...
    [CONTINUE FOREACH]
    ...
    [EXIT FOREACH]
END FOREACH
```

Here are the guidelines you must follow to use the FOREACH statement:

- Follow the FOREACH keyword with the name of a cursor you have previously declared.
- Include an INTO clause if there is no INTO clause in the SELECT statement associated with the named cursor. The INTO clause lists the variables that will store the column values returned by the query. Make sure the variables in the list correspond in order and in type to the columns in the SELECT clause of the SELECT statement associated with the named cursor.
- List the statements you want **INFORMIX-4GL** to execute for each row returned by the query.
- Use CONTINUE FOREACH or EXIT FOREACH with the IF statement if you want to interrupt the sequence of statements in a FOREACH loop under certain conditions. (See the section "The IF Statement," later in this chapter, for details.)
- Include the END FOREACH keywords after the last statement to be executed for a given row. When **INFORMIX-4GL** encounters the END FOREACH keywords, it starts the FOREACH loop again, unless there are no more rows to be processed.

The following example shows how to use the FOREACH loop to display all the rows in the **customer** table:

```
DEFINE p_customer RECORD LIKE customer.*

DECLARE q_curs CURSOR FOR
   SELECT * FROM customer

FOREACH q_curs INTO p_customer.*
   DISPLAY p_customer.customer_num
   DISPLAY p_customer.fname CLIPPED, " ", p_customer.lname
   DISPLAY p_customer.company
   DISPLAY p_customer.address1
   DISPLAY p_customer.address2
   DISPLAY p_customer.city CLIPPED, ", ", p_customer.state,
      "  ", p_customer.zipcode
   DISPLAY p_customer.phone
   DISPLAY ""
END FOREACH
```

As you can see, this example includes the following INFORMIX-4GL statements:

- The DEFINE statement defines a record with the same structure as the **customer** table. This record will be used to store the rows retrieved by the SELECT statement.

- The DECLARE statement associates a non-scrolling cursor named **q_curs** with the SELECT statement SELECT * FROM customer.

- The FOREACH statement repeatedly retrieves a row from the customer table and displays it on screen until all rows in the **customer** table have been processed.

These statements produce a display like the following for each row in the **customer** table:

```
120
Enid Smythe
Smythe Sports
Bay Road
P.O. Box 111
Palo Alto, CA  94303
415-329-2222
```

The IF Statement. One way you can interrupt the sequence of statements in a FOREACH loop is by using the IF statement with the CONTINUE FOREACH or EXIT FOREACH statement. The IF statement has the following format:

```
IF Boolean-expr
   THEN
        statement
        ...
   [ELSE
        statement
        ... ]
END IF
```

where *Boolean-expr* is a Boolean expression that evaluates to TRUE or FALSE, and *statement* is any **INFORMIX-4GL** statement.

If the given expression is TRUE, **INFORMIX-4GL** executes the statements between THEN and ELSE (if present) or between THEN and END IF (if ELSE is not present). If the given expression is FALSE, **INFORMIX-4GL** executes only the statements between ELSE (if present) and IF END.

The CONTINUE FOREACH Statement. You can use IF with the CONTINUE FOREACH statement to interrupt the sequence of statements in a FOREACH loop and retrieve the next row. For example, you can use conditional statements to display data for those customers living outside of Palo Alto:

```
DECLARE q_curs CURSOR FOR
   SELECT * FROM customer

FOREACH q_curs INTO p_customer.*
   IF p_customer.city = "Palo Alto" THEN
         CONTINUE FOREACH
   END IF

   DISPLAY p_customer.customer_num
   DISPLAY p_customer.fname CLIPPED, " ", p_customer.lname
   DISPLAY p_customer.company
   DISPLAY p_customer.address1
   DISPLAY p_customer.address2
   DISPLAY p_customer.city CLIPPED, ", ", p_customer.state,
         "  ", p_customer.zipcode
   DISPLAY p_customer.phone
   DISPLAY ""
END FOREACH
```

When **INFORMIX-4GL** encounters the IF statement in this example, it determines whether the value in the **city** column for the current row is Palo Alto. If the current row represents a customer who lives in Palo Alto, then **INFORMIX-4GL** executes the CONTINUE FOREACH statement and returns to the top of the FOREACH loop to retrieve the next row. If the current row represents a customer who does not live in Palo Alto, then **INFORMIX-4GL** ignores the CONTINUE FOREACH statement and displays the data in the current row. In this way, **INFORMIX-4GL** displays only the data for customers who live outside of Palo Alto.

Note: You can also display data for customers living outside of Palo Alto by using a WHERE clause in the SELECT statement instead of using IF and CONTINUE FOREACH in the FOREACH statement.

The EXIT FOREACH Statement. You can use IF with the EXIT FOREACH statement to interrupt the sequence of statements and leave the FOREACH loop entirely. For example, you can stop the FOREACH loop if the user enters a specific value in response to a prompt:

```
DECLARE q_curs CURSOR FOR
   SELECT * FROM customer

FOREACH q_curs INTO p_customer.*

   DISPLAY p_customer.customer_num
   DISPLAY p_customer.fname CLIPPED, " ", p_customer.lname
   DISPLAY p_customer.company
   DISPLAY p_customer.address1
   DISPLAY p_customer.address2
   DISPLAY p_customer.city CLIPPED, ", ", p_customer.state,
         "  ", p_customer.zipcode
   DISPLAY p_customer.phone
   DISPLAY ""

   PROMPT "Do you want to see the next customer (y/n): "
         FOR answer

   IF answer = "n" THEN
         EXIT FOREACH
   END IF

END FOREACH
```

When INFORMIX-4GL encounters the IF statement in this example, it evaluates the Boolean expression. If the value of **answer** is n, INFORMIX-4GL executes the EXIT FOREACH statement and leaves the loop. If **answer** has any other value, INFORMIX-4GL ignores the EXIT FOREACH statement and attempts to retrieve another row from the active set.

The OPEN, FETCH, and CLOSE Statements

FOREACH combines several functions in a single statement:

- It *opens* the cursor associated with a SELECT statement.
- It repeatedly *fetches* a row from the active set and executes a series of statements for that row.
- It *closes* the cursor after all the rows in the active set have been processed.

The following sections show how to achieve the same result by using the OPEN, FETCH, and CLOSE statements with a looping statement such as WHILE. As you will see, the FETCH statement is more versatile than the FOREACH statement because it allows you to process rows in random as well as sequential order. (See the section "More About the FETCH Statement" and Chapter 8 for details.)

The OPEN Statement. You can use the OPEN statement to determine the active set for the SELECT statement associated with a cursor. The simplest form of the OPEN statement is

```
OPEN cursor-name
```

where *cursor-name* is the name of a cursor that you have previously declared. After **INFORMIX-4GL** executes an OPEN statement, it leaves the cursor pointing just before the first row of the active set.

For example, if you declare and open the following cursor

```
DECLARE q_curs CURSOR FOR
    SELECT * FROM customer WHERE city = "Sunnyvale"

OPEN q_curs
```

INFORMIX-4GL determines the rows that satisfy the SELECT statement and leaves the cursor before the first row:

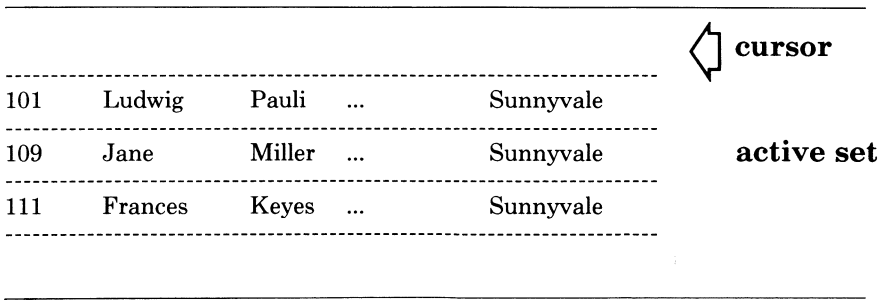


Figure 4-1. Position of Cursor After the OPEN Statement

Note: The OPEN statement determines the logical set of rows that satisfies the WHERE clause of the SELECT statement but does not actually retrieve them from the database. This function is performed by the FETCH statement described in the next section.

The FETCH Statement. You can use a simple form of the FETCH statement to move the cursor to the next row in the active set and to retrieve values from that row. A simple FETCH statement has the following format:

FETCH *cursor-name* [INTO *variable-list*]

Here are the guidelines you must follow to use this form of the FETCH statement:

- Follow the FETCH keyword with the name of a cursor that you have previously declared and opened.

- Include an INTO clause if there is no INTO clause in the SELECT statement associated with the named cursor. The INTO clause lists the variables that store the column values returned by the FETCH statement. Make sure the variables in the list correspond in order and in type to the columns in the SELECT clause of the SELECT statement associated with the named cursor.

For example, if you execute the following FETCH statement

```
FETCH q_curs INTO p_customer.*
```

INFORMIX-4GL advances the cursor to the first row and retrieves values from that row:

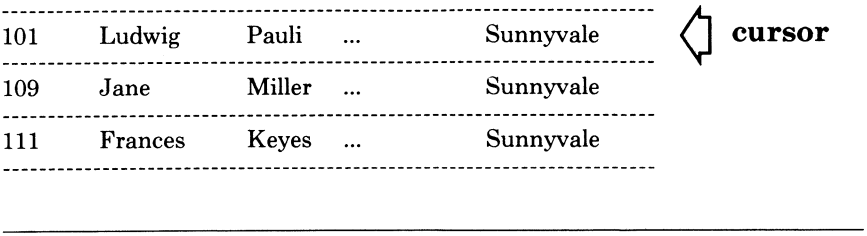


Figure 4-2. Position of Cursor After the First FETCH

If you then execute another FETCH statement, INFORMIX-4GL advances the cursor to the second row and retrieves values from that row:

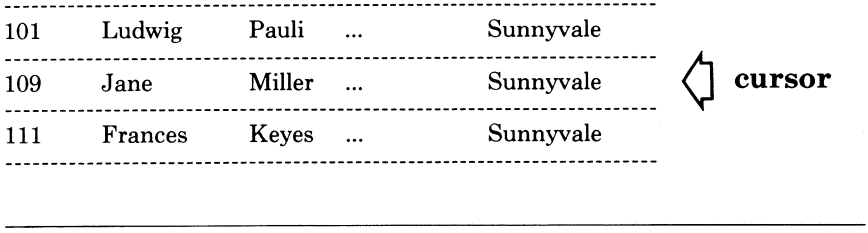


Figure 4-3. Position of Cursor After the Second FETCH

If the cursor is pointing to the last row of the active set and you execute a FETCH statement, INFORMIX-4GL moves the cursor beyond the last row in the active set but does not retrieve any values. Instead, INFORMIX-4GL sets a **status** variable to NOTFOUND to indicate that the cursor is "out of bounds."

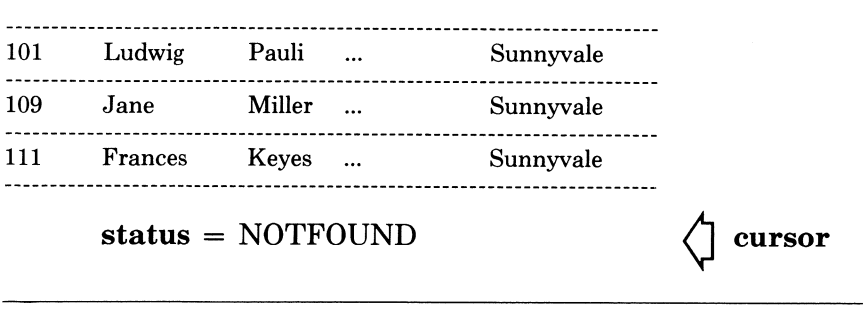


Figure 4-4. Position of Cursor After the Fourth FETCH

Since you usually cannot predict the number of rows in an active set, you should test the **status** variable immediately after every FETCH statement to determine whether the end of the active set has been reached. For example, the following IF statement displays the message Row not found if the cursor has moved beyond the end of the active set; otherwise it displays the customer row in the **p_customer** record on the screen:

```
FETCH q_curs INTO p_customer.*

IF status = NOTFOUND THEN
    MESSAGE "Row not found"
ELSE
    DISPLAY p_customer.customer_num
    DISPLAY p_customer.fname CLIPPED, " ", p_customer.lname
    DISPLAY p_customer.company
    DISPLAY p_customer.address1
    DISPLAY p_customer.address2
    DISPLAY p_customer.city CLIPPED, ", ", p_customer.state,
            "  ", p_customer.zipcode
    DISPLAY p_customer.phone
END IF
```

Note: Although you can test the **status** variable, you cannot define it or assign values to it. The **status** variable is defined by INFORMIX-4GL and indicates whether an error has occurred during the execution of an INFORMIX-4GL statement. If INFORMIX-4GL executes a statement successfully, the value of

status will be zero. If an error occurs during the execution of a statement, **status** will be a negative number. After a FETCH statement that moves the cursor beyond the end of the active set, the value of **status** will be NOTFOUND. See Chapter 10, "Handling Errors and Interrupts from the User," for more information on the **status** variable.

The CLOSE Statement. When you no longer need to refer to the rows specified by a SELECT statement, you can *close* the cursor associated with that statement. The CLOSE statement has the following format:

CLOSE *cursor-name*

where *cursor-name* is the name of the cursor you want to close.

For example, you can close the **q_curs** cursor by executing the following statement:

```
CLOSE q_curs
```

The WHILE Statement

You can use a WHILE loop to repeatedly FETCH a row and test the **status** variable until all the rows in the active set have been processed. The WHILE statement has the following format:

WHILE *Boolean-expression*

statement
...
[CONTINUE WHILE]
...
[EXIT WHILE]

END WHILE

Here are the guidelines you must follow when using the WHILE loop:

- Follow the WHILE keyword with a Boolean expression that can be either TRUE or FALSE.

- List the statements you want **INFORMIX-4GL** to execute while the given expression is TRUE.

- Use CONTINUE WHILE or EXIT WHILE if you want to interrupt the sequence of statements in a WHILE loop. The CONTINUE WHILE statement causes **INFORMIX-4GL** to evaluate the Boolean expression again and execute the statements in the loop if the expression is still TRUE. The EXIT WHILE statement causes **INFORMIX-4GL** to execute the first statement that follows the END WHILE keywords.

- Include the END WHILE keywords after the last statement you want to execute in the loop. When **INFORMIX-4GL** encounters the END WHILE keywords, it executes the statements in the WHILE loop again if the Boolean expression is still TRUE.

The following example shows how to use the WHILE loop with FETCH and IF statements to process the rows in an active set:

```
DEFINE p_customer RECORD LIKE customer.*

DECLARE q_curs CURSOR FOR
   SELECT * FROM customer WHERE city = "Sunnyvale"

OPEN q_curs

LET answer = "y"

WHILE answer = "y"

   FETCH q_curs INTO p_customer.*

   IF status = NOTFOUND THEN
      MESSAGE "No row found."
      EXIT WHILE
   END IF

   DISPLAY p_customer.customer_num
   DISPLAY p_customer.fname CLIPPED, " ", p_customer.lname
   DISPLAY p_customer.company
   DISPLAY p_customer.address1
   DISPLAY p_customer.address2
   DISPLAY p_customer.city CLIPPED, ", ", p_customer.state,
         "  ", p_customer.zipcode
   DISPLAY p_customer.phone
   DISPLAY ""

   PROMPT "Do you want to see the next row (y/n): " FOR answer

END WHILE

CLOSE q_curs
```

This example includes the following statements:

- The DECLARE statement associates a cursor called **q_curs** with the SELECT statement that retrieves the customers who live in Sunnyvale.

- The OPEN statement determines the rows that satisfy the SELECT statement and leaves the cursor pointing before the first row of the active set.

- The WHILE loop includes statements that repeatedly fetch and display a row until the active set is exhausted:

  1. The FETCH statement advances the cursor and retrieves a row (if the cursor points to one).

  2. The IF statement displays a message and exits the WHILE loop if the value of the **status** variable is NOTFOUND, indicating that the cursor has moved beyond the last row in the active set.

  3. The DISPLAY statements print the current values in the **p_customer** record on screen.

  4. The PROMPT statement asks the user to supply a value for **answer**. If the user enters y in response to the prompt, INFORMIX-4GL repeats the statements in the WHILE loop again, since the Boolean expression is TRUE. If the user enters any other character, INFORMIX-4GL ends the WHILE loop since the Boolean expression is FALSE.

- The CLOSE statement closes the cursor after the rows in the active set have been processed.

As you can see, the statements in this example produce nearly the same result as the FOREACH example in a previous section, "The EXIT FOREACH Statement."

More About the FETCH Statement

You can use the FETCH statement with different options to retrieve rows in any order you choose. The complete syntax for the FETCH statement follows:

```
FETCH [NEXT | {PREVIOUS | PRIOR} | FIRST | LAST |
  CURRENT | RELATIVE m | ABSOLUTE n ] cursor-name
  [INTO variable-list]
```

The options have the following effect:

NEXT	indicates the next row in the active set. If you use the FETCH statement without any options, **INFORMIX-4GL** executes a FETCH NEXT statement.
PREVIOUS	indicates the previous row in the active set.
PRIOR	is a synonym for PREVIOUS.
FIRST	indicates the first row of the active set.
LAST	indicates the last row of the active set.
CURRENT	indicates the current row of the active set.
RELATIVE *m*	indicates the *m*th row relative to the current cursor position in the active set. *m* may evaluate to a positive or negative integer.
ABSOLUTE *n*	indicates the *n*th row in the active set.

You must first declare a scrolling cursor before issuing any FETCH statement except FETCH or FETCH NEXT. You must also test the **status** variable after the FETCH statement to determine whether either end of the active set has been reached. For example, if the cursor is pointing to the first row in the following illustration and you execute a FETCH PREVIOUS statement, **INFORMIX-4GL** moves the cursor before the first row and sets the **status** variable to NOTFOUND.

status = NOTFOUND



101	Ludwig	Pauli	...	Sunnyvale
109	Jane	Miller	...	Sunnyvale
111	Frances	Keyes	...	Sunnyvale

status = NOTFOUND

Figure 4-5. Position of Cursor After FETCH PREVIOUS

Similarly, if the cursor is pointing to the first row and you execute the statement FETCH ABSOLUTE 4, INFORMIX-4GL moves the cursor after the last row and sets **status** to NOTFOUND.

status = NOTFOUND

101	Ludwig	Pauli	...	Sunnyvale
109	Jane	Miller	...	Sunnyvale
111	Frances	Keyes	...	Sunnyvale

status = NOTFOUND



Figure 4-6. Position of Cursor After FETCH ABSOLUTE 4

The following example shows how to determine whether the active set is empty:

```
DECLARE q_curs SCROLL CURSOR FOR
   SELECT * FROM customer WHERE city = "Sunnyvale"

OPEN q_curs

FETCH FIRST q_curs INTO p_customer.*

IF status = NOTFOUND THEN
   MESSAGE "No rows in the active set."
ELSE
   MESSAGE "The active set contains at least one row."
END IF
```

The example includes a DECLARE statement that declares a scrolling cursor and a FETCH FIRST statement that attempts to retrieve the first row in the active set. The IF statement tests the **status** variable and displays an appropriate message, depending on its value.

Note: When you initially fetch a row from the database with a scrolling cursor, all the rows in the active set up to and including the fetched row are placed in a temporary file and remain there until you close the cursor. If you then fetch the same row or any row prior to it, **INFORMIX-4GL** makes the retrieval from the temporary file instead of from the database. This means that subsequent changes to the database may not be reflected in the active set used by a scrolling cursor.

With the flexibility provided by the FETCH statement, you can create applications that allow users to specify which rows in the active set they want to see. Chapter 8 shows you how to use FETCH statements with the MENU statement to produce such an application.

Complex SELECT Statements

Until now, you have used simple SELECT statements that retrieve information from one table at a time. This section explains additional features of the SELECT statement and shows you how to query several tables at once.

More About the SELECT Clause

The following sections explain how to eliminate duplicate rows retrieved by a SELECT statement and how to perform calculations on numeric columns listed in the SELECT clause.

Eliminating Duplicate Rows

You can use either the DISTINCT or UNIQUE keyword in a SELECT statement to eliminate duplicate rows retrieved by the query. Following are two equivalent SELECT clauses that eliminate duplicates:

SELECT DISTINCT *select-list*

SELECT UNIQUE *select-list*

For example, the following SELECT statement eliminates duplicate values if more than one customer lives in a particular city:

```
DECLARE q_curs CURSOR FOR
   SELECT UNIQUE city FROM customer
```

Similarly, the following SELECT statement eliminates duplicate values if a particular customer has placed more than one order:

```
DECLARE q_curs CURSOR FOR
   SELECT UNIQUE customer_num FROM orders
```

Arithmetic Operators

The following arithmetic operators can be used with numeric columns in a SELECT clause:

Symbol	Function
+	Addition
–	Subtraction
*	Multiplication
/	Division

You could use an arithmetic operator, for example, to add five percent to the unit price of each item in the **stock** table and store the result in the program variable **new_price**:

```
DEFINE new_price LIKE stock.unit_price

DECLARE q_curs CURSOR FOR
   SELECT unit_price * 1.05 INTO new_price FROM stock
```

Similarly, you could retrieve the shipping charge for each row in the **orders** table and reduce it by $0.50:

```
DECLARE q_curs CURSOR FOR
   SELECT ship_charge – .5 FROM orders
```

Math Functions

You can also perform calculations on columns in the SELECT clause using the math or *aggregate* functions that follow:

COUNT(*)	Counts the number of rows satisfying the WHERE criteria
SUM(*column-name*)	Adds all the values in the column you specify
AVG(*column-name*)	Finds the average value in the column you specify
MAX(*column-name*)	Finds the maximum (highest) value in the column you specify

MIN(*column-name*)	Finds the minimum (lowest) value in the column you specify

Each aggregate function finds one result for the whole table (or for a group of rows) instead of one result for each row.

When you use an aggregate function with a column, you have to enclose the column name in parentheses. If you use more than one function in a SELECT clause, separate the functions with commas. The following statement, for example, finds the average, minimum, and maximum prices of items listed in the **stock** table:

```
DEFINE avg_price, min_price, max_price  MONEY(6)

SELECT AVG(unit_price), MIN(unit_price), MAX(unit_price)
   INTO avg_price, min_price, max_price
   FROM stock
```

Aggregates are very useful in a SELECT statement that includes the GROUP BY clause. (See the *INFORMIX-4GL Reference Manual* for details.)

More About the WHERE Clause

Many of the SELECT statements in this guide contain WHERE clauses that compare column values to constants. These comparisons are made using *relational operators* such as = and <. This section describes how to use other relational operators as well as ranges, sets, and pattern-matching statements to specify search conditions.

Relational Operators

You can use the following relational operators to describe the relationship between a column and a value in a WHERE clause:

Symbol	Meaning
=	Equal to
< > or !=	Not equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to

With dates, greater than (>) means later and less than (<) means earlier. With characters, greater than (>) means later in the alphabet (closer to *z*) and less than (<) means earlier in the alphabet (closer to *a*). Lowercase letters are greater than uppercase letters, and uppercase letters are greater than numbers since **INFORMIX-4GL** compares characters based on ASCII values. Make sure you put double quotes around DATE and CHAR values when you use them with relational operators.

You can use the >= operator, for example, to retrieve all orders that were placed on or after June 4, 1986:

```
DECLARE q_curs CURSOR FOR
   SELECT * FROM orders
   WHERE order_date >= "06/04/86"
```

Similarly, you can select data for all customers who have last names that start with *A* through *K* by using the < operator:

```
DECLARE q_curs CURSOR FOR
   SELECT customer_num, lname, company, city, state
   FROM customer
   WHERE lname < "L"
```

Using Several Search Conditions: the AND and OR Keywords

If you want to use more than one search condition in a WHERE clause, you can connect the search conditions with the AND keyword. You can also include alternative search conditions with the OR keyword. This means you can write SELECT statements as follows:

```
DECLARE q_curs CURSOR FOR
        SELECT * FROM orders
        WHERE customer_num = 104
        AND order_date < "6/6/1986"
```

This statement selects orders placed before June 6, 1986, by the customer with the number 104.

```
DECLARE q_curs CURSOR FOR
        SELECT * FROM orders
        WHERE order_date = "06/05/1986"
        OR ship_date = "06/05/1986"
```

This statement selects orders where the order date or the shipment date is June 5, 1986.

Ranges: the BETWEEN and AND Keywords

You can use the BETWEEN and AND keywords in a WHERE clause to specify the range for a column value in a search condition. The general format for using ranges in a WHERE clause is

WHERE *column-name*
  [NOT] BETWEEN *expression* AND *expression*

The range is inclusive.

The following example shows how you might use a range to search for customers who live in a given region:

```
DECLARE q_curs CURSOR FOR
        SELECT fname, lname, zipcode FROM customer
        WHERE zipcode BETWEEN "94300" AND "94310"
```

If the boundaries of a range change frequently, you may want to use program variables instead of constants after BETWEEN and AND as in the following example:

```
LET low_price = 100.00
LET high_price = 400.00

DECLARE q_curs CURSOR FOR
        SELECT * FROM stock
        WHERE unit_price BETWEEN low_price AND high_price
```

Sets: the IN Keyword

You can use the IN keyword in a WHERE clause to list the desired values for a given column. The general format is

column-name [NOT] IN (*value1, value2, ... valueN*)

As with other types of comparisons, you should enclose DATE and CHAR values in double quotes, and make sure that the values have data types that are compatible with the given column.

For example, you can select customers who live in Sunnyvale or Palo Alto with the following statement:

```
DECLARE q_curs CURSOR FOR
        SELECT * FROM customer
        WHERE city IN ("Sunnyvale", "Palo Alto")
```

Pattern Matching: the MATCHES Keyword

You can use the MATCHES keyword in a WHERE clause to find row(s) containing a particular word or phrase. The general form is

WHERE *char-column* MATCHES "*value*"

where *char-column* is a character column, and *value* is a character value enclosed in double quotes.

The strength of the MATCHES keyword lies in the special characters you can use with it. The special characters described in this chapter are shown below:

*	Replaces zero or more characters.
?	Replaces any single character.
[...]	Replaces any characters within the square brackets, including character ranges such as [a-z]. If the first character within the brackets is a caret (^), it matches any character that is not listed.

When you use a special character, make sure it appears within the quotes that surround the value you are searching for. Here are some examples:

MATCHES "S*"	Matches any word that begins with an uppercase *S*.
MATCHES "*ere"	Matches any word that ends with the letters *ere*.
MATCHES "*an*"	Matches any word that includes the letters *an*.
MATCHES "an?"	Matches any three-letter word that begins with *an*.
MATCHES "r?t"	Matches any three-letter word that begins with *r* and ends with *t*.

MATCHES "[yY]" Matches *y* or *Y*.

MATCHES "[^ab]" Matches any character except *a* or *b*.

To select data for customers whose last names end in *son*, for example, you could enter

```
DECLARE q_curs CURSOR FOR
        SELECT * FROM customer
        WHERE lname MATCHES "*son"
```

This statement would retrieve data for customers with last names like Watson, Lawson, and Albertson.

Note: You can also use the LIKE keyword to determine whether a column value matches a particular pattern. See Chapter 7 of the *INFORMIX-4GL Reference Manual* for details.

The ORDER BY Clause

The previous sections described additional features of the SELECT and WHERE clauses that were introduced in Chapter 3. This section introduces a new clause, ORDER BY, that enables you to sort the rows retrieved by a SELECT statement. You can use the ORDER BY clause in a SELECT statement to sort query results by any column specified in the SELECT clause. The general form is as follows:

ORDER BY *column1* [, *column2, ...columnn*]

Here are the guidelines for using the ORDER BY clause in a SELECT statement:

- Follow the ORDER BY keywords with the names of the columns by which you want to sort.
- Use only columns named in the SELECT clause in an ORDER BY clause.

For example, you can use the ORDER BY clause in a SELECT statement to retrieve customer rows and arrange them in alphabetical order by last name:

```
DECLARE q_curs CURSOR FOR
SELECT customer_num, fname, lname, company
        FROM customer
        ORDER BY lname
```

You could also order the rows by any other column named in this SELECT clause, including **customer_num**, **fname**, or **company**.

Sorting by More than One Column

The ORDER BY clause also handles *nested sorts*, or sorts within sorts. When you use more than one sort column, separate the column names following the ORDER BY keywords with commas.

The following statement, for example, sorts customer rows by last name, and then by first name within each last-name category:

```
DECLARE q_curs CURSOR FOR
SELECT customer_num, fname, lname, company FROM customer
       ORDER BY lname, fname
```

Selecting Data from Several Tables

You can use a SELECT statement to retrieve data from more than one table, as long as all the tables you are working with belong to the same database. You can follow the format below to select data from two tables:

```
SELECT column-list
   FROM table1, table2
   WHERE [table1.]column = [table2.]column
```

Here are the guidelines for using more than one table in a SELECT statement:

- Follow the SELECT keyword with the names of the columns you want to find, separated with commas.
- When a column name appears in more than one table, preface the column name with the table name. Use a period (.) to separate the table name and the column name.
- Follow the FROM keyword with the names of the tables you want to use, arranged in any order and separated with commas.
- Use the WHERE clause to join columns from different tables that contain the same type of data. When join columns have the same name, preface the column name with the table name, and separate the two parts of the name with a period.

For example, the **customer** table contains information about customers and the **orders** table contains information about customer orders:

Customer Table	**Orders Table**
customer_num	order_num
fname	order_date
lname	customer_num
company	ship_instruct
address1	backlog
address2	po_num
city	ship_date
state	ship_weight
zipcode	ship_charge
phone	paid_date

Both tables contain **customer_num** columns. By joining the customer number columns, you can select information from both tables at the same time. For example, the following SELECT statement searches the database to find customer and order information for each customer who placed an order after June 5, 1986:

```
DECLARE q_curs CURSOR FOR
SELECT customer.customer_num, lname, fname, order_date
       FROM customer, orders
       WHERE customer.customer_num = orders.customer_num
            AND order_date > "6/5/1986"
       ORDER BY customer.customer_num
```

The SELECT clause in this example contains three columns from the **customer** table (**customer.customer_num**, **lname**, and **fname**), and one column from the **orders** table (**order_date**). It selects information from both tables by joining the **customer_num** column in the **customer** table to the **customer_num** column in the **orders** table.

Note: You can list either the **customer.customer_num** or the **orders.customer_num** column in the SELECT clause. However, if you are going to sort by the **customer_num** column, you must use the **customer_num** column listed in the SELECT clause in the ORDER BY clause. You cannot list **customer.customer_num** in the SELECT clause and **orders.customer_num** in the ORDER BY clause.

You can follow the same rules to select information from three or more tables in the **stores** database.

For example, the following statement retrieves information from the **orders**, **items**, and **stock** tables, by using **order_num** to join the **orders** table to the **items** table and by using **stock_num** and **manu_code** to join the **items** table to the **stock** table:

```
SELECT orders.order_num, order_date, items.stock_num,
       items.manu_code, description, quantity
   FROM orders, items, stock
   WHERE orders.order_num = items.order_num
       AND items.stock_num = stock.stock_num
       AND items.manu_code = stock.manu_code
   ORDER BY orders.order_num
```

This statement finds the order number, order date, stock number, manufacturer code, description, and quantity for all items on order. It sorts the results by order number.

Chapter Summary

- You can retrieve groups of rows from a database by using a SELECT statement with a cursor.
- Declare a cursor for a query using the DECLARE statement.
- Select the rows and execute a series of statements for each row returned by a query using the FOREACH statement. Alternatively, you can use the OPEN, FETCH, and CLOSE statements with a looping statement such as WHILE.
- Use the IF statement to execute a series of statements conditionally.
- You can execute a group of statements as long as a condition is true by using the WHILE statement.
- For convenience, use records instead of simple program variables to store the rows retrieved by a SELECT statement. You can define a record by using a DEFINE statement with the RECORD or RECORD LIKE keywords.
- You can perform calculations on any numeric column in the SELECT clause of a SELECT statement by using arithmetic operators or math functions.
- You can use relational operators, ranges, sets, and pattern-matching statements when you specify search conditions in the WHERE clause of a SELECT statement.
- Include an ORDER BY clause in a SELECT statement to sort query results by a given column.
- You can select information from several tables at once by listing the tables in the FROM clause and specifying the *join* conditions in the WHERE clause of your SELECT statement.

Chapter 5

Functions

Chapter 5 Table of Contents

Chapter Overview 5

Writing Functions 6
- The FUNCTION Statement 6
- Calling Functions 7
- Passing Values Between the Main Program and a Function 8
- Global and Local Variables 9
  - The GLOBALS Statement 9
  - Scope Rules 11
  - Global and Local Variables with the Same Name 13
  - A Sample Program That Uses Global Variables 15
- Parameters 16
- Using Parameters in the FUNCTION Statement 16
  - A Sample Program that Uses Parameters 20
- Returning Values from a Function to the Main Program 21
  - A Sample Program That Returns Values 23
  - Using a Function in an Expression 24
- Library Functions 25

Chapter Summary 26

Chapter Overview

This chapter explains how to use functions in INFORMIX-4GL programs. The chapter includes a discussion of

- How to call functions
- How to use global and local variables
- How to pass parameters to functions
- How to write functions that return values
- How to use functions in expressions

For complete information about the statements you use to handle functions in INFORMIX-4GL, refer to the *INFORMIX-4GL Reference Manual.*

Writing Functions

A function is a collection of statements that performs a particular task. By using functions to represent groups of statements, you can divide a large program into smaller modules that are easy to write, test, and change. Functions also allow you to repeat a series of operations without having to specify the steps for each repetition.

The FUNCTION Statement

You can define a function by using the FUNCTION statement outside the main program. A simple FUNCTION statement has the following format:

```
FUNCTION function-name ()
  statement
  ...
END FUNCTION
```

where *function-name* is the name of the function you want to define and *statement* is any **INFORMIX-4GL** statement. Functions may contain variable definition statements, assignment statements, database manipulation statements, SQL statements, program flow statements, and input/output statements. The statements you include depend only on the task you want the function to perform.

For example, the following function counts the orders placed on a given day and displays the number of orders on the screen:

```
FUNCTION count_orders ()

        DEFINE ord_date      DATE,
               row_count     INTEGER

        PROMPT "Enter an order date: "
               FOR ord_date

        SELECT COUNT(*)
               INTO row_count
               FROM orders
               WHERE order_date = ord_date

        DISPLAY row_count, " orders were placed on ",
               ord_date

END FUNCTION
```

As you can see, the function contains a DEFINE statement, SELECT statement, and statements for input and output.

Calling Functions

To use a function in an **INFORMIX-4GL** application, you invoke, or call, it from the main program (or another function) with the CALL statement. A simple CALL statement has the following format:

CALL *function*()

When **INFORMIX-4GL** encounters a CALL statement, it locates the named function and executes the function statements in sequence. Upon reaching the END FUNCTION keywords, **INFORMIX-4GL** resumes execution from the statement that follows the function call.

For example, the following CALL statement invokes the **count_orders** function from the main program:

```
DATABASE stores

MAIN
        ...
        CALL count_orders ()
        ...
END MAIN

FUNCTION count_orders ()

        DEFINE ord_date      DATE,
               row_count     INTEGER

        PROMPT "Enter an order date: "
            FOR ord_date

        SELECT COUNT(*)
            INTO row_count
            FROM orders
            WHERE order_date = ord_date

        DISPLAY row_count, " orders were placed on ",
            ord_date

END FUNCTION
```

Passing Values Between the Main Program and a Function

Many functions receive information from the calling routine, process the data, and return one or more values. Since functions reside outside the calling routine, they have access to program information only through global variables or parameters. This section explains how to use global variables to pass information between the main program and a function. Later sections explain how to use parameters.

A global variable retains its value anywhere in a program, while a local variable has meaning only in the main program or function in which it is defined. In **INFORMIX-4GL**, all variables you define in MAIN and FUNCTION statements are local to those routines. If you want both the main program and a function to be able to use a variable, you must declare it in a GLOBALS statement outside the main program. Once declared, a global variable can be used to transfer data between the main program and any function.

The GLOBALS Statement

A simple GLOBALS statement has the following format:

```
GLOBALS
  DEFINE-statement
  [...]
END GLOBALS
```

To use this form of the GLOBALS statement, follow these guidelines:

- The GLOBALS statement must occur at the beginning of the program before the MAIN and FUNCTION statements.
- Follow the GLOBALS keyword with one or more DEFINE statements for the global variables you want to declare.
- Follow the DEFINE statements with the END GLOBALS keywords.
- If you plan to define a global variable LIKE a database column or table, make sure you name the database by including a DATABASE statement before the GLOBALS statement.

The following examples show you how to use the GLOBALS statement in **INFORMIX-4GL**.

Example 1.

The following statement defines two global variables called **cust_balance** and **cust_name**:

```
GLOBALS
    DEFINE cust_balance      MONEY,
           cust_name         CHAR(15)
END GLOBALS
```

Example 2.

If you want to define the global variable **cust_name** with the LIKE keyword, you could insert a DATABASE statement before the GLOBALS statement as follows:

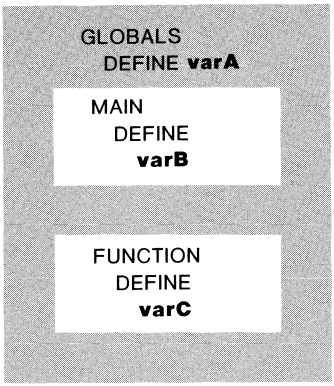
```
DATABASE stores

GLOBALS
    DEFINE cust_balance      MONEY,
           cust_name LIKE customer.lname
END GLOBALS
```

Note: You can also use the GLOBALS statement to refer to a file where global variables are defined. See the *INFORMIX-4GL Reference Manual* for more information.

Scope Rules

Scope refers to the part of a program where a variable has meaning—that is, where the variable can represent a value. Figure 5-1 shows the scope of global and local variables in INFORMIX-4GL programs:



Statement	Variable	Type	Scope
GLOBALS	varA	global	MAIN
			FUNCTION
MAIN	varB	local	MAIN
FUNCTION	varC	local	FUNCTION

Figure 5-1. Scope of Global and Local Variables

As shown, the global variable **varA** has meaning throughout the program, while the local variables **varB** and **varC** only exist in the blocks in which they are defined. This means that if the main program assigns a value to a global variable, and then calls a function that assigns a different value to the global variable, the value of the global variable changes in the main program as well.

This is shown in the following example:

```
GLOBALS

        DEFINE account_bal      MONEY

END GLOBALS


MAIN

        LET account_bal = 0
        CALL change_bal()
        DISPLAY account_bal

END MAIN

FUNCTION change_bal()

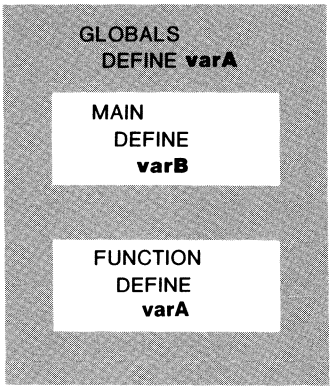
        LET account_bal = 99

END FUNCTION
```

In this example, the main program assigns the value 0 to the global variable **account_bal**, and then calls the function **change_bal**. After **change_bal** sets **account_bal** to 99, the main program displays 99.00 as the value of the global variable.

Global and Local Variables with the Same Name

If a local variable has the same name as a global variable, the local variable takes precedence inside the block in which it is defined. This feature is illustrated in Figure 5-2.



Statement	Variable	Type	Scope
GLOBALS	varA	global	MAIN
MAIN	varB	local	MAIN
FUNCTION	varA	local	FUNCTION

Figure 5-2. Scope of Variables with the Same Name

In this case, if the main program assigns a value to a global variable, and then calls a function that assigns a different value to a local variable of the same name, the value of the global variable remains unchanged.

This is shown in the following example:

```
GLOBALS

        DEFINE account_bal      MONEY

END GLOBALS


MAIN

        LET account_bal = 0
        CALL change_bal()
        DISPLAY account_bal

END MAIN


FUNCTION change_bal()

        DEFINE account_bal      MONEY
        LET account_bal = 99

END FUNCTION
```

In this example, the main program assigns the value 0 to the global variable **account_bal**, and calls a function, **change_bal**, which contains a local variable of the same name. After **change_bal** sets the local variable to 99, it returns control to the main program. Since the function did not change the value of the global variable, **account_bal**, the main program displays 0.00 as its value.

A Sample Program That Uses Global Variables

The following example shows how you can pass information from one routine to another by using global variables:

```
DATABASE stores

GLOBALS

        DEFINE p_customer RECORD LIKE customer.*

END GLOBALS


MAIN
        DEFINE cust_num          INTEGER

        PROMPT "Enter a customer number: " FOR cust_num

        SELECT * INTO p_customer.* FROM customer
             WHERE customer_num = cust_num

        CALL print_cust()

END MAIN


FUNCTION print_cust()
        DISPLAY p_customer.lname CLIPPED, ", ",
             p_customer.fname
        DISPLAY p_customer.address1
        DISPLAY p_customer.address2
        DISPLAY p_customer.city CLIPPED, ", ",
             p_customer.state CLIPPED, "  ",
             p_customer.zipcode

END FUNCTION
```

This program contains a global record called **p_customer**, which is available to both the main program and the function **print_cust**. The main program selects a row from **customer** table, stores it in **p_customer**, and calls the **print_cust** function. **print_cust** then displays a customer name and address on screen, using the values of the global record **p_customer**. (If the record **p_customer** had been defined in the main program instead of in a GLOBALS statement, the **print_cust** function would not have access to information stored in it.)

Parameters

Parameters are variables or values that may appear between the parentheses of a FUNCTION and its associated CALL statement. Usually you will want to use parameters instead of global variables to pass information between the main program and a function.

With parameters, you can pass information from the main program to a function and change the information in the function without affecting the values of variables in the rest of the program. With global variables, on the other hand, you have to consider whether you want the value of a global variable to change in the rest of the program when you change its value in a function.

This section explains how to use parameters with the FUNCTION and CALL statements to pass information between the main program and a function.

Using Parameters in the FUNCTION Statement

To use parameters in a FUNCTION statement, include simple or record variables between the parentheses that follow the function name. (When the variables receive specific values from the calling routine, they are called *arguments*.) Since parameters are variables, you must define them in the body of the function using a DEFINE statement. For example,

```
FUNCTION print_name (lname, fname)
    DEFINE lname, fname      CHAR(15)
    ...
END FUNCTION
```

This function contains two parameters, **lname** and **fname**, which are separated by a comma. Similarly,

```
FUNCTION print_cust (f_customer)

    DEFINE f_customer RECORD LIKE customer.*
    ...

END FUNCTION
```

This function contains one parameter called **f_customer**, which is defined as a record. (If you list a record as a parameter, make sure it does not contain array components, and do not use the .* notation after the record name.)

Parameters differ from local variables since they receive initial values from the calling statement. This means that the data type of a parameter should be compatible with or identical to the type of value it receives from the calling statement.

To call the **print_name** function from the main program, you can use a CALL statement that contains variables or constants between the parentheses following the function name. A variable must be initialized before it appears in a CALL statement and may not be an array or a record that contains an array component.

This is shown in the following example:

```
MAIN

    DEFINE last_name,
           first_name          CHAR(15)

    LET last_name = "Smith"

    LET first_name = "John"

    CALL print_name (last_name, first_name)

END MAIN


FUNCTION print_name (lname, fname)

    DEFINE lname, fname     CHAR(15)

    DISPLAY lname CLIPPED, ", ", fname

END FUNCTION
```

In this example the main program contains two variables, **last_name** and **first_name**, which receive the values Smith and John, respectively. These variables appear between the parentheses of the CALL statement as parameters. When INFORMIX-4GL encounters the CALL statement, it evaluates the variables **last_name** and **first_name** and assigns the resulting values to the function parameters **lname** and **fname**.

Notes: You cannot use arrays or records containing array components as parameters in a CALL statement. However, you may use records without array components in a CALL statement along with the **.*** or THRU notation to pass values to a function. When **INFORMIX-4GL** encounters either notation in a CALL statement, it substitutes the equivalent list of record variables for *record-name.** or *record-name.variable-name1* THRU *record-name.variable-name2*.

It does not matter whether a parameter in a CALL statement has the same name as the corresponding parameter in a FUNCTION statement, since all arguments in a FUNCTION statement are local to that function. For example,

Program **Screen Display**

```
MAIN

      DEFINE lname, fname CHAR(15)
      LET lname = "Smith"
      LET fname = "John"
      CALL print_name (lname, fname)
      DISPLAY lname CLIPPED, ", ", fname              Smith, John

END MAIN


FUNCTION print_name (lname, fname)

      DEFINE lname, fname CHAR(15)
      DISPLAY lname CLIPPED, ", ", fname              Smith, John
      LET lname = "Jones"
      LET fname = "David"
      DISPLAY lname CLIPPED, ", ", fname              Jones, David

END FUNCTION
```

A Sample Program that Uses Parameters

The following program retrieves customer rows from the **stores** database and displays a name, company, and address for each customer row retrieved. As you can see, the main program passes information to a function through the use of parameters:

```
DATABASE stores

MAIN

        DEFINE p_customer RECORD LIKE customer.*

        DECLARE q_curs CURSOR FOR
                SELECT * FROM customer

        FOREACH q_curs INTO p_customer.*

                CALL print_cust (p_customer.fname,
                p_customer.lname,
                p_customer.company,
                p_customer.address1,
                p_customer.address2,
                p_customer.city,
                p_customer.state,
                p_customer.zipcode)

        END FOREACH

END MAIN


FUNCTION print_cust (fname, lname, company,
                addr1, addr2, city, state, zip)

        DEFINE fname,
                lname CHAR(15),
                company,
                addr1,
                addr2 CHAR(20),
                city CHAR(15),
                state CHAR(2),
                zip CHAR(5)

        DISPLAY lname CLIPPED, ", ", fname
        DISPLAY company
        DISPLAY addr1
        DISPLAY addr2
        DISPLAY city CLIPPED, ", ", state CLIPPED, "  ", zip
        DISPLAY ""
        DISPLAY ""

END FUNCTION
```

The CALL statement in this example contains eight arguments that correspond to the eight arguments in the **print_cust** function. When **INFORMIX-4GL** encounters the CALL statement, it evaluates the expressions in the argument list and passes the resulting values to the named function. **print_cust** then displays the customer name, company, and address on screen, using the given arguments.

Note: Since the the parameters in the CALL statement are record elements and have been previously defined, you can simplify this statement by using the following notation:

```
CALL print_cust (p_customer.fname
     THRU p_customer.zipcode)
```

Returning Values from a Function to the Main Program

The previous section explained how to pass information from the main program to a function with parameters. This section shows you how to pass information from a function back to the main program by including a RETURN statement in the function and a RETURNING clause in the associated CALL statement. This is shown in the following example:

```
MAIN

        DEFINE price      MONEY

        CALL get_price() RETURNING price

        DISPLAY "The total price is: ", price

END MAIN


FUNCTION get_price()

        DEFINE qty        SMALLINT,
           unit_price     MONEY

        LET qty = 3
        LET unit_price = 4.00
        RETURN qty * unit_price

END FUNCTION
```

In this example, the **get_price** function assigns values to the local variables, **qty** and **unit_price**, and returns the result of the expression qty * unit_price to the main program where it is stored in the variable **price**.

If a function returns more than one value, list the expressions after the RETURN keyword and separate them with commas. You should also make sure that the RETURNING clause of the associated CALL statement contains a variable for each value returned by the function. This is shown in the following example:

```
MAIN

        DEFINE wholesale,
               retail          MONEY

        CALL get_price()
               RETURNING wholesale, retail

        DISPLAY "Wholesale price:  ", wholesale

        DISPLAY "Retail price:  ", retail

END MAIN


FUNCTION get_price()

        DEFINE qty              SMALLINT,
               whole_price,
               ret_price        MONEY

        LET qty = 3
        LET whole_price = 2.75
        LET ret_price = 3.99

        RETURN (qty * whole_price), (qty * ret_price)

END FUNCTION
```

In this example, the function returns the values of the expressions (qty * whole_price) and (qty * ret_price) to the main program where they are assigned to the variables **wholesale** and **retail**, respectively.

Notes: You do not have to enclose an expression in parentheses when you include it in a list. Parentheses simply help to make a list of expressions more readable.

You cannot use arrays or records containing array components as parameters in a RETURN statement or RETURNING clause. However, you may use records without array components along with the .* or THRU notation to return values from a function to the calling routine.

A Sample Program That Returns Values

The following program invokes a function that returns the largest and smallest orders for a given item:

```
DATABASE stores

MAIN

        DEFINE stock_number,
            high_order,
            low_order           SMALLINT,
            manu_code           CHAR(3)

        PROMPT "Enter a stock number:  "
            FOR stock_number

        PROMPT "Enter a manufacturer code:  "
            FOR manu_code

        CALL order_range(stock_number, manu_code)
            RETURNING high_order, low_order

        DISPLAY "High order:  ", high_order

        DISPLAY "Low order:  ", low_order

END MAIN


FUNCTION order_range (stock_number, m_code)

        DEFINE stock_number,
            max_qty,
            min_qty             SMALLINT,
            m_code              CHAR(3)

        SELECT MAX(quantity), MIN(quantity)
            INTO max_qty, min_qty
            FROM items
            WHERE stock_num = stock_number
            AND manu_code = m_code

        RETURN max_qty, min_qty

END FUNCTION
```

The CALL statement and the FUNCTION statement in this program have matching argument lists and RETURN clauses. When INFORMIX-4GL encounters the CALL statement, it passes the values of **stock_number** and **manu_code** to the **order_range** function. **order_range** then selects the largest and smallest orders for the given item and returns these values to the main program, where they are assigned to the program variables **high_order** and **low_order**.

Using a Function in an Expression

If a function returns a single value, you can use it in an expression in place of a variable. For example, the following function finds the total quantity on order for a given item and returns that value to the main program:

```
FUNCTION total_qty (stock_number, m_code)

        DEFINE stock_number,
               tot_qty      SMALLINT,
               m_code       CHAR(3)

        SELECT SUM(quantity) INTO tot_qty
               FROM items
               WHERE stock_num = stock_number
               AND manu_code = m_code

        RETURN tot_qty

END FUNCTION
```

Since the **total_qty** function returns only one value, you can use the function in an expression like the following:

```
DATABASE stores

MAIN
        DEFINE stock_number,
               on_hand          SMALLINT,
               manu_code        CHAR(3)

        PROMPT "Enter a stock number:  "
           FOR stock_number

        PROMPT "Enter a manufacturer code:  "
           FOR manu_code

        LET on_hand = 1000 - total_qty(stock_number, manu_code)

        DISPLAY "Quantity on hand:  ", on_hand

END MAIN
```

In this example, **INFORMIX-4GL** prompts the user for a stock number and manufacturer code and passes the information to **total_qty**. The function finds the total quantity for the given item and returns the value to the main program. **INFORMIX-4GL** then subtracts the value from 1000 and assigns the result to the variable **on_hand**.

Note: You cannot use functions in **RDSQL** statements.

Library Functions

INFORMIX-4GL includes a library of "built-in" functions that you can use in your application programs. These functions are fully explained in Chapter 5 of the *INFORMIX-4GL Reference Manual.*

Chapter Summary

- You can use functions to create structured programs that are easy to write, read, and modify.
- You use the FUNCTION statement outside the main program to define a function.
- Use the CALL statement in the main program to invoke a function.
- If a function returns a single value, you can use it in an expression in place of a variable instead of invoking it with the CALL statement.
- All variables you define in the MAIN and FUNCTION statements are local to those routines. If you want to use a global variable, you must declare it in a GLOBALS statement outside the main program.
- If a local variable has the same name as a global variable, the local variable takes precedence inside the block in which it is defined.
- To pass information from the main program to a function, you can use global variables or parameters.
- To pass information from a function to the main program, you can use global variables or include a return clause in the function and the associated CALL statement.
- INFORMIX-4GL provides built-in library functions that are precompiled and linked to your programs automatically.

Chapter 6

Designing Screen Forms

Chapter 6 Table of Contents

Chapter Overview 5

Using Screen Forms 6

The Form Specification File 9
  Sections of a Form Specification File 12

Parts of the Form Specification File 13
  The DATABASE Section 13
  The SCREEN Section 14
    Display Fields 15
  TABLES Section 17
  ATTRIBUTES Section 18
    Assigning Field Names to Field Tags 19
    Assigning Attributes 20
    The NOENTRY Attribute 23
    The REQUIRED Attribute 23
    The REVERSE Attribute 24
    The INCLUDE Attribute 25
    The DEFAULT Attribute 26
    The UPSHIFT and DOWNSHIFT Attributes 27
    The AUTONEXT Attribute 28
    The COMMENTS Attribute 28
    The PICTURE Attribute 29
  The INSTRUCTIONS Section 31
    Defining Screen Records 32
    Changing Delimiters 34

The Programmer's Environment 35
  Creating a Default Form Specification File 36
  Modifying a Form Specification File 39
    Correcting Errors 41
    Saving the Form Specification 42
  Creating a Form Specification File with a Text Editor 43
    Correcting Errors 44
    Saving the Form Specification 45

Chapter Summary 46

Chapter Overview

This chapter explains how to create screen forms for use in data entry and retrieval. Topics covered include

- What a screen form is
- What a form specification file is
- What the parts of a form specification are
- How to assign attributes to display fields
- How to define screen records and change delimiters
- How to create and compile a form specification file
- How to modify a form specification file

Chapter 6 discusses basic features useful for both simple and complex screen forms. Once you have created and compiled a screen form, you can use it in an **INFORMIX-4GL** program, as explained in Chapter 7.

For more information about screen forms, refer to Chapter 11 and the *INFORMIX-4GL Reference Manual.*

Using Screen Forms

Screen forms provide the user with a tool for entering, retrieving, modifying, and deleting data in a database.

Figure 6-1 shows a screen form designed for processing data in the **customer** table of the **stores** database. This sample form is designed to be used by data entry operators in a sporting-goods distribution center. It is part of the demonstration database.

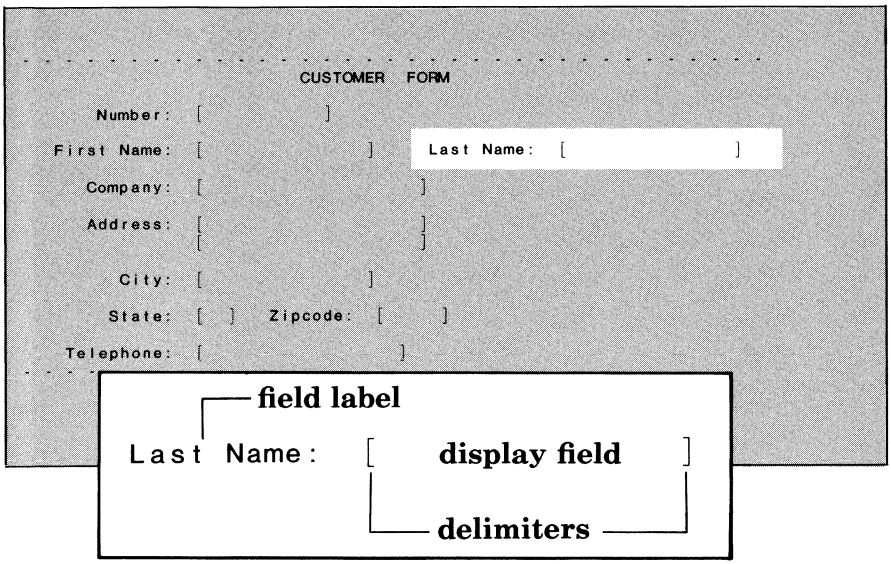


Figure 6-1. The **customer** Form

The screen form contains *display fields* bounded by *delimiters* such as the square brackets [ ] shown here. Each display field corresponds to a column in the **customer** table of the **stores** database. Labels such as "Company" and "Zipcode" make the fields easy to identify.

You can write INFORMIX-4GL programs that allow the user to enter, retrieve, update, and delete customer information through the **customer** form. If you want the user to enter customer information, you can use the INPUT statement described in Chapter 7 to allow the user to fill in the fields on the screen form. The user can fill in one field at a time, pressing RETURN to move to the next field. (The order in which the cursor moves from field to field on a screen form is determined by the order in which you list fields in the INPUT statement.) Any time before pressing RETURN in the last field, the user can use the ARROW keys to move back through the fields and make corrections. The user can indicate that data entry is complete by pressing ESCAPE in any field or by pressing RETURN in the last field.

Note: Information that the user enters through a screen form does not go directly into a table in the database. Instead, you use the INPUT statement described in Chapter 7 to retrieve the data from one or more display fields and transfer it to program variables. Then you can use RDSQL statements such as INSERT to add the information to the appropriate table(s).

The following screen shows how the user might fill in the **customer** form:

```
                        CUSTOMER  FORM

       Number:  [         119]

   First Name:  [Enid            ]     Last Name:  [Smythe          ]

      Company:  [Smythe Sports       ]

      Address:  [Bay Road            ]
                [P. O. Box 111       ]

         City:  [Palo Alto       ]

        State:  [CA]   Zipcode:  [94303]

    Telephone:  [415-329-2222       ]
```

Figure 6-2. Data Entered on the **customer** Form

You can also display information in the database on a screen form. If you use the SELECT statement to retrieve a row from the **customer** table, for example, you can display it on the **customer** form with the DISPLAY statement described in Chapter 7. Then you can let the user update or delete the customer row if you choose.

The Form Specification File

To create a screen form like the one in Figure 6-1, you must create a form specification file and compile it using the **FORM4GL** program. The form specification file is a text file containing the layout of the screen form and instructions for displaying the data. There are several ways to create a form specification file in the **INFORMIX-4GL** Programmer's Environment:

- Use **FORM4GL** to create a *default* form specification.
- Modify the default form specification using the system text editor.
- Use a text editor to create a new form specification from scratch.

Figure 6-3 shows a default form specification file.

```
database stores
screen
{
customer_num          [f000          ]
fname                 [f001               ]
lname                 [f002               ]
company               [f003                    ]
address1              [f004                    ]
address2              [f005                    ]
city                  [f006               ]
state                 [a0]
zipcode               [f007 ]
phone                 [f008                ]
}
end
tables
customer
attributes
f000 = customer.customer_num;
f001 = customer.fname;
f002 = customer.lname;
f003 = customer.company;
f004 = customer.address1;
f005 = customer.address2;
f006 = customer.city;
a0 = customer.state;
f007 = customer.zipcode;
f008 = customer.phone;
end
```

Figure 6-3. A Default Form Specification File

Figure 6-4 shows a modified version of the same form. It is the **customer** form specification used in the examples in this chapter.

```
DATABASE STORES

SCREEN
{
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                          CUSTOMER  FORM

       Number:  [f000       ]

   First Name:  [f001           ]     Last Name:  [f002           ]

      Company:  [f003                ]

      Address:  [f004                ]
                [f005                ]

         City:  [f006           ]

        State:  [a0]   Zipcode:  [f007 ]

    Telephone:  [f008              ]
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
}
END

TABLES
customer

ATTRIBUTES
f000 = customer.customer_num;
f001 = customer.fname;
f002 = customer.lname;
f003 = customer.company;
f004 = customer.address1;
f005 = customer.address2;
f006 = customer.city;
a0 = customer.state;
f007 = customer.zipcode;
f008 = customer.phone;
END

INSTRUCTIONS
SCREEN RECORD sc_cust (customer.fname THRU customer.phone)
END
```

Figure 6-4. The **customer** Form Specification File

Compiling this form specification file with **FORM4GL** created the **customer** screen form shown in Figure 6-1.

Note: **FORM4GL** requires the extension **.per** for files containing form specifications and the extension **.frm** for files containing compiled forms. For example, the form specification file in Figure 6-4 has the filename **customer.per** and the compiled form in Figure 6-1 has the filename **customer.frm**. If you create and compile a form specification through the

INFORMIX-4GL Programmer's Environment, FORM4GL adds these extensions automatically. See the section "The Programmer's Environment," later in this chapter, for more information about creating and compiling form specifications.

Sections of a Form Specification File

A form specification file consists of three required sections (DATABASE, SCREEN, and ATTRIBUTES) and two optional sections (TABLES and INSTRUCTIONS) in the following order:

Database	Identifies the database (if any) on which the form is based
Screen	Determines the layout of the screen form
Tables	Identifies the table or tables (if any) that contain the columns that correspond to the display fields on the screen form
Attributes	Links field tags (identifying display fields) to field names and optionally assigns attributes after the field names
Instructions	Defines screen records and alternate delimiters

Parts of the Form Specification File

This section explains the parts of the form specification file in detail.

Note: Keywords appear in uppercase letters. In practice, however, lowercase letters can be used for keywords, because **FORM4GL** does not distinguish between uppercase and lowercase letters.

The DATABASE Section

Begin the form specification with the DATABASE section.

```
DATABASE STORES

SCREEN
{
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                          CUSTOMER  FORM

        Number:  [f000       ]

   First Name:  [f001            ]     Last Name:  [f002            ]
```

Figure 6-5. The DATABASE Section (shaded)

The format of the DATABASE section is

DATABASE *database-name*

where *database-name* identifies the database on which the form is based.

Note: You can specify FORMONLY as the database if no display fields in your form specification file are associated with database columns. See Chapter 3 of the *INFORMIX-4GL Reference Manual* for details.

Follow the DATABASE section with the SCREEN section. This section contains the layout of the screen form, as shown in the following example:

```
SCREEN
{
                              CUSTOMER  FORM

        Number:  [f000        ]

   First Name:  [f001              ]     Last Name:  [f002              ]

      Company:  [f003                          ]

      Address:  [f004                          ]
                [f005                          ]

         City:  [f006              ]

        State:  [a0]   Zipcode:  [f007 ]

    Telephone:  [f008                 ]

}
END
```

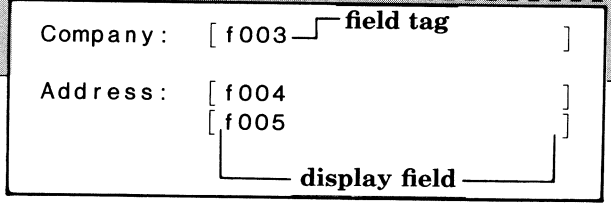


Figure 6-6. The SCREEN Section

The SCREEN section begins with the SCREEN keyword followed by a left brace {. It ends with a right brace } followed by the END keyword (optional).

The screen layout between the braces is identical to the screen form in Figure 6-1 except for the *field tags* (f000, f001, etc.), which appear between the square brackets. These field tags link the display fields to field names that are usually derived from database columns.

Note: The screen layout (what appears between the braces) must not exceed 20 lines since **INFORMIX-4GL** reserves four lines of the screen for prompts, messages, comments, and error messages.

Display Fields

Display fields are the areas where the user may enter data or where the **INFORMIX-4GL** program will display data.

Use the following format to define display fields in the SCREEN section:

```
[field-tag     ]
```

where *field-tag* identifies the field within the form specification file. Field tags appear again in the ATTRIBUTES section.

Optionally, you may label display fields to make it easier for the user to determine what information to enter in each field.

Field Delimiters. You must use square brackets [ ] to indicate display fields in the SCREEN section of the form specification file. However, you can have **INFORMIX-4GL** substitute other characters for the square brackets when it displays the compiled screen form by including a DELIMITERS statement in the INSTRUCTIONS section of the form specification file. See "The INSTRUCTIONS Section," later in this chapter, for more information.

Note: If you want to include many display fields in the SCREEN section, you can save space by using the vertical bar | to mark the end of one display field and the beginning of another. This is shown in the following example:

```
SCREEN
{
      Number        First Name       Last Name

  [f000         |f001             |f002            ]
}
```

However, you must include an INSTRUCTIONS section in the form specification that contains a DELIMITERS statement defining two identical symbols as the beginning and ending delimiters for display fields. See the section entitled "The INSTRUCTIONS Section," later in this chapter, for more information.

Field Width. The width of a display field is the number of characters that can be placed between the delimiters. To prevent **INFORMIX-4GL** from truncating displayed data,

- Make character fields the same width as the corresponding database column.

- Make numeric fields wide enough to accommodate the largest displayed number.

The following table shows the length **FORM4GL** assigns to display fields of each type when it generates a default form specification file.

Data Type	**Default Field Length**
DECIMAL	Total length specified in the CREATE TABLE statement, plus 2
MONEY	Total length specified in the CREATE TABLE statement, plus 3
CHAR	Length specified in the CREATE TABLE statement
SMALLINT	6
INTEGER	11
SERIAL	11
SMALLFLOAT	14
FLOAT	14
DATE	10

If you have **FORM4GL** generate a default form specification, **FORM4GL** refers to the database table(s) you specify for information about data types and display field widths.

Field Tags. Field tags are used in the SCREEN and ATTRIBUTES sections to link display fields with *field names*. Unless you use screen arrays (see Chapter 11), each display field must have a unique tag.

If you have **FORM4GL** generate a default form specification, **FORM4GL** supplies field tags like f000 and f001 to correspond to the first display field, the second display field, and so on. **FORM4GL** assigns a field tag like a0 to any field that cannot accommodate its standard, four-character tag.

A field tag can be from one to 50 characters long—provided it does not exceed the field width. The first character of a field tag must be a letter. The remaining characters may be any combination of letters, numbers, and underscores. Since **FORM4GL** is not sensitive to case, **a1** and **A1** represent the same tag.

TABLES Section

Following the SCREEN section is the TABLES section. The TABLES section lists the tables containing the columns that appear in the ATTRIBUTES section. The TABLES section and part of the ATTRIBUTES section of the **customer** form are shown below:

```
TABLES
customer

ATTRIBUTES
f000 = customer.customer_num;
f001 = customer.fname;
f002 = customer.lname;
```

Figure 6-7. The TABLES Section (shaded)

Although this example lists only one table, you can list up to 14 tables in the TABLES section. Each table must be part of the

database specified in the DATABASE section. (See Chapter 11 and the *INFORMIX-4GL Reference Manual* for more information about multiple-table forms.)

The format of the TABLES section is

```
TABLES table-name [...]
```

where *table-name* identifies a table in the database. A list of tables may be separated by spaces or commas.

Note: You do not need to include a TABLES section if you have specified FORMONLY in the DATABASE section of a form specification file. See the *INFORMIX-4GL Reference Manual* for details.

ATTRIBUTES Section

The ATTRIBUTES section serves two purposes. First, the ATTRIBUTES section assigns a field name to every field tag that appears in the SCREEN section. Second, it optionally lists one or more attributes after a field name. (Attributes can, for example, cause **INFORMIX-4GL** to display a default value in a particular field or protect a field against data entry.)

Assigning Field Names to Field Tags

The ATTRIBUTES section links field names to field tags. Following is the ATTRIBUTES section of the **customer** form specification file:

```
TABLES
customer

ATTRIBUTES
f000 = customer.customer_num;
f001 = customer.fname;
f002 = customer.lname;
f003 = customer.company;
f004 = customer.address1;
f005 = customer.address2;
f006 = customer.city;
a0 = customer.state;
f007 = customer.zipcode;
f008 = customer.phone;
END
```

Figure 6-8. The ATTRIBUTES Section (shaded)

The ATTRIBUTES section includes one assignment statement for each field tag in the SCREEN section. The section begins with the ATTRIBUTES keyword and ends with the END keyword (optional).

In most cases, you use the following format to assign field names to field tags:

field-tag = [*table-name.*]*column-name;*

where *field-tag* is a field tag in the SCREEN section, the optional *table-name* is a table in the TABLES section, and *column-name* is a column in the table that corresponds to a particular display field.

The *table-name* is required only if the *column-name* appears in more than one table in the specified database. (From now on, [*table-name.*]*column-name* is simply referred to as *field-name.*) End each assignment statement with a semicolon (;).

For example,

```
f000 = customer.customer_num;
```

Note: The ATTRIBUTES section may also link field tags to field names that are not associated with any database columns with the format *FORMONLY.name.* For more information about FORMONLY fields, see the *INFORMIX-4GL Reference Manual.*

Assigning Attributes

Optionally, you may assign one or more attributes to a display field by listing the attributes after the corresponding field name in the ATTRIBUTES section. For example, you can use attributes to

- Describe how data should be displayed in a particular field
- Define valid entries for a particular field
- Prevent the user from entering data in a particular field
- Assign a default to a particular field

Use the following format when defining attributes:

field-tag = *field-name*[, *attribute-list*];

where *attribute-list* is one or more attributes, separated by commas.

Note: An attribute takes effect when you list the corresponding field name in an INPUT or DISPLAY statement in an **INFORMIX-4GL** program.

The following table summarizes all attributes recognized by INFORMIX-4GL:

Attribute	Description
AUTONEXT	Causes the cursor to automatically advance to the next field when the current field is full. (The "next" field is determined by the order of field names in the INPUT statement.)
COMMENTS	Displays a message on the comment line (by default, line 23 of the screen) when the cursor moves to the associated field.
DEFAULT	Assigns a default value to a display field.
DISPLAY LIKE	Indicates that the field is to have the display characteristics of another field, or is to have the display characteristics of a column listed in the **syscolatt** table. (Refer to Chapter 14 for more information about **syscolatt**.)
DOWNSHIFT	Converts uppercase letters in CHAR fields to lowercase.
FORMAT	Controls the format of data in DECIMAL, SMALLFLOAT, FLOAT, or DATE display fields.
INCLUDE	Lists acceptable values for a field.
NOENTRY	Protects a field from data entry.
PICTURE	Imposes a specific format for data entered in a CHAR field.
REQUIRED	Requires data to be entered into a certain field.

REVERSE	Causes a specified field to be displayed in reverse video.
UPSHIFT	Converts lowercase characters entered in CHAR fields to uppercase.
VALIDATE LIKE	Validates the data entered in a field by using the same restrictions specified for a column in the **syscolval** table. (See Chapter 14 for more information about **syscolval**.)
VERIFY	Requires users to enter data twice in a particular field to ensure that the entered value is correct.

For details on the DISPLAY LIKE, FORMAT, VALIDATE LIKE, and VERIFY attributes, refer to the *INFORMIX-4GL Reference Manual.* The other commonly used attributes are described in the following sections in this order:

- NOENTRY
- REQUIRED
- REVERSE
- INCLUDE
- DEFAULT
- UPSHIFT
- DOWNSHIFT
- AUTONEXT
- COMMENTS
- PICTURE

These attributes are listed in the order you might add them to the **customer** form specification in Figure 6-4.

The NOENTRY Attribute

The NOENTRY attribute protects a field from data entry. Use it to prevent the user from entering data into fields that contain information supplied by the **INFORMIX-4GL** program.

The format of NOENTRY is

field-tag = *field-name*, NOENTRY;

where *field-tag* identifies a display field in the SCREEN section, *field-name* is the name of a field, and NOENTRY is a keyword.

Example. Use the NOENTRY attribute if you do not want the user to be able to move the cursor into the **customer.customer_num** field:

```
f000 = customer.customer_num, NOENTRY;
```

The REQUIRED Attribute

If you include the REQUIRED attribute after a field name, the user *must* enter data in the corresponding display field. In this way, the user cannot end the INPUT statement until a value has been entered into the required field.

Note: When the user presses **ESCAPE** after entering data without filling in a required field, **INFORMIX-4GL** displays an error message and refuses to accept the data. **INFORMIX-4GL** ignores the REQUIRED attribute during updates.

Use the following format to define a field as REQUIRED

field-tag = *field-name*, REQUIRED;

where REQUIRED is a keyword.

Example. To ensure that the user fills in the **customer.fname** and **customer.lname** fields, you can assign them the REQUIRED attribute:

```
f001 = customer.fname, REQUIRED;
f002 = customer.lname, REQUIRED;
```

The REVERSE Attribute

The REVERSE attribute causes **INFORMIX-4GL** to display a field in reverse video. Its format is

field-tag = *field-name*, REVERSE;

where REVERSE is a keyword. Reverse video is useful for highlighting significant fields.

Note: Some terminals do not support reverse video. On these terminals, fields assigned the REVERSE attribute are enclosed in angle brackets < >.

Example. To emphasize the importance of the **customer.fname** and **customer.lname** fields, you can have **INFORMIX-4GL** display them in reverse video by adding the REVERSE attribute:

```
f001 = customer.fname,
       REQUIRED, REVERSE;
f002 = customer.lname,
       REQUIRED, REVERSE;
```

The INCLUDE Attribute

Use the INCLUDE attribute to specify all acceptable values for a field. The format of the INCLUDE attribute is

field-tag = *field-name*, INCLUDE = (*value-list*);

where INCLUDE is a keyword and *value-list* is a list of acceptable values enclosed in parentheses.

Specifying Values. You can specify acceptable values by listing the values, specifying a range of values, or both. For example,

List of values:	INCLUDE = (10, 15, 20)
Range of values:	INCLUDE = ("A" TO "K")
Combination:	INCLUDE = (1, 2, 10 TO 20)

Follow these guidelines when you specify a value list for the INCLUDE attribute:

- Enclose DATE values or CHAR values that contain spaces or special characters in double quotes. (Generally, it is recommended that you enclose *all* CHAR values in quotation marks.)
- When specifying a range of numeric values, list the lower value first.
- When specifying a range of character values, use ASCII order.

Example. The following INCLUDE attribute defines CA, WA, and OR as acceptable values for the **customer.state** field:

```
a0 = customer.state,
     INCLUDE = ("CA", "WA", "OR");
```

This statement restricts entries to CA(lifornia), WA(shington), and OR(egon).

The DEFAULT Attribute

If you expect the user to enter the same value repeatedly in a display field, you can use the DEFAULT attribute to assign a default value to the field. The default value is displayed whenever an INPUT statement lists that particular field.

The format of the DEFAULT attribute is

field-tag = *field-name*, DEFAULT = *value*;

where DEFAULT is a keyword and *value* is the selected default.

Here are the guidelines for using the DEFAULT attribute:

- Enclose DATE values or CHAR values that include spaces or special characters in double quotes. (Generally, it is recommended that you enclose all CHAR values in quotation marks.)

- Use the TODAY keyword when you want **INFORMIX-4GL** to supply the current date as the default value for a DATE field.

- Do not assign the DEFAULT and REQUIRED attributes to the same field, since **INFORMIX-4GL** will ignore the REQUIRED attribute.

If you do not use the DEFAULT attribute with a display field, **INFORMIX-4GL** displays blanks in that field unless you have assigned a default to the corresponding column in the **syscolval** table. See Chapter 14 and the *INFORMIX-4GL Reference Manual* for more information about **syscolval**.

Example. You can assign CA as the default value for the **customer.state** field as follows:

```
a0 = customer.state,
     INCLUDE = ("CA", "WA", "OR"),
     DEFAULT = "CA";
```

The DEFAULT attribute in this example produces the following result when you use the corresponding field name in an INPUT statement:

```
City:  [                ]

State: [CA]   Zipcode:  [       ]
```

Figure 6-9. The **customer.state** Field with a Default Value

The user can either press **RETURN** to accept the default value or enter another acceptable value in the **customer.state** field.

The UPSHIFT and DOWNSHIFT Attributes

The UPSHIFT attribute converts lowercase characters to uppercase characters; DOWNSHIFT converts uppercase characters to lowercase characters. The format of these attributes is

field-tag = *field-name*, UPSHIFT;

and

field-tag = *field-name*, DOWNSHIFT;

where UPSHIFT and DOWNSHIFT are keywords.

Example. You can use the UPSHIFT attribute to have INFORMIX-4GL convert values like ca or Wa to uppercase letters so that they will be recognized as acceptable values for the **customer.state** field:

```
a0 = customer.state,
     INCLUDE = ("CA", "WA", "OR"),
     DEFAULT = "CA",
     UPSHIFT;
```

The AUTONEXT Attribute

AUTONEXT causes the cursor to move to the next field when the current field is full. (The "next" field is determined by the order of field names in the INPUT statement.) The format is

field-tag = *field-name*, AUTONEXT;

where AUTONEXT is a keyword.

AUTONEXT is useful for CHAR fields into which the user always types the same number of characters.

Example. Use the AUTONEXT attribute if you want the cursor to move to the next field when the user fills the **customer.zipcode** field:

```
f007 = customer.zipcode, AUTONEXT;
```

The COMMENTS Attribute

You can use the COMMENTS attribute to have INFORMIX-4GL display a message on the comment line (by default, line 23 of the screen) when the cursor moves to the associated display field. The format of COMMENTS is

field-tag = *field-name*, COMMENTS = "message";

where COMMENTS is a keyword and *message* is a character string enclosed in double quotes.

Example. You can use the COMMENTS attribute to tell the user to enter an area code, phone number, and extension in the **customer.phone** field.

```
f008 = customer.phone,
       COMMENTS = "Enter an area code, phone number, and extension.";
```

The COMMENTS attribute produces the following result when the user moves the cursor to the **customer.phone** field during an INPUT statement:

```
        State: [CA]   Zipcode: [94301]

    Telephone: [□                 ]
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

Enter an area code, phone number, and extension.
```

Figure 6-10. Using COMMENTS with the **customer.phone** Field

Note: You can also display other messages on the screen form using the MESSAGE and DISPLAY AT statements described in Chapter 7.

The PICTURE Attribute

The PICTURE attribute assigns a specific character pattern to a CHAR field. Its format is

field-tag = *field-name*, PICTURE = "*format*";

where PICTURE is a keyword and *format* is a character string enclosed in double quotes. The number of characters in *format* must equal the number of characters in the corresponding display field.

format may be composed of the following:

Symbol	Meaning
A	A blank that the user can replace with any letter
#	A blank that the user can replace with any digit
X	A blank that the user can replace with any *alphanumeric* character

INFORMIX-4GL displays any other character in the format literally. When the user moves the cursor to a field that has the PICTURE attribute, **INFORMIX-4GL** displays the literal characters and replaces the **A**, **#**, and **X** characters with blanks.

If the user enters data into the field that does not conform to the *format*, **INFORMIX-4GL** will not accept it.

Example. You can specify a format for phone numbers entered in the **customer.phone** field by using a picture like the following:

```
f008 = customer.phone,
       COMMENTS = "Enter an area code, phone number, and extension.",
       PICTURE = "(###)###-####-####";
```

When the user moves the cursor into the **customer.phone** field, INFORMIX-4GL displays the parentheses and hyphens and allows the user to enter only digits in the field:

```
        State:  [  ]    Zipcode:  [      ]
    Telephone:  [(   )   -    -    ]
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

Enter an area code, phone number, and extension.
```

Figure 6-11. Using PICTURE with the **customer.phone** Field

The INSTRUCTIONS Section

When used, the optional INSTRUCTIONS section follows the ATTRIBUTES section. It is used to

- Define screen records
- Change the default delimiters for display fields

Following is the INSTRUCTIONS section of the **customer** form specification file:

```
f007 = customer.zipcode;
f008 = customer.phone;
END

INSTRUCTIONS
SCREEN RECORD sc_cust (customer.fname THRU customer.phone)
END
```

Figure 6-12. The INSTRUCTIONS Section (shaded)

As you can see, the INSTRUCTIONS section begins with the INSTRUCTIONS keyword and ends with the END keyword (optional).

Defining Screen Records

You can use a SCREEN RECORD statement in the INSTRUCTIONS section to group fields into screen records that can be used in INPUT and DISPLAY statements in INFORMIX-4GL programs.

Default Screen Records. FORM4GL generates a default screen record for each table listed in the form specification file. A default screen record has the same name as the table on which it is based, and includes all the fields in the form specification that are associated with that particular table. For example, the **customer** form specification has one default screen record called **customer** that includes all the fields with names that begin with customer. The order of fields in the default screen record is the same as the order of fields in the ATTRIBUTES section.

The SCREEN RECORD Statement. You can use the SCREEN RECORD statement in the INSTRUCTIONS section to define a screen record that includes some or all of the fields in a form specification file. Following are three formats for defining screen records.

Format 1.

SCREEN RECORD *rec-name* (*field-name1* THRU *field-name2*)

where SCREEN RECORD are keywords, *rec-name* is the name of the screen record, *field-name1* is the first field in a range, THRU is a keyword, and *field-name2* is the last field in a range.

For example, the INSTRUCTIONS section of the **customer** form contains a SCREEN RECORD statement that defines a screen record called **sc_cust**. This screen record includes the fields **customer.fname** through **customer.phone** on the **customer** form. The order of fields in the **sc_cust** record is the same as the order of the fields in the ATTRIBUTES section. Chapter 7 includes sample programs that use the **sc_cust** record.

Format 2.

SCREEN RECORD *rec-name* (*field-list*)

where *field-list* is one or more field names, separated by commas.

For example, the following SCREEN RECORD statement defines a screen record called **names** that lists only the **customer.fname**, **customer.lname**, and **customer.company** fields:

```
SCREEN RECORD names (customer.fname, customer.lname,
                     customer.company)
```

Format 3.

SCREEN RECORD *rec-name* (*table.**)

where *table.** is the name of a table in the TABLES section. The asterisk after the table name indicates that the screen record includes all the fields with names that begin with *table.* The order of fields in the screen record is the same as the order of fields in the ATTRIBUTES section of the form specification file.

Notes:

- If you want, you can use more than one format in a SCREEN RECORD statement. For example,

```
SCREEN RECORD sc_cust (customer_num, lname THRU phone)
```

- Do not use a table name as a record name in a SCREEN RECORD statement since **FORM4GL** uses each table name as the name of a default screen record.

- If you reorder the statements in the ATTRIBUTES section of a form specification, make sure that screen records defined using the THRU or the .* notation are not affected.

- If you define a screen record in the INSTRUCTIONS section and use it in an INPUT statement, the cursor moves from field to field according to the order in which fields are listed in the screen record.

Changing Delimiters

By default, **INFORMIX-4GL** uses square brackets to mark fields when it displays a screen form. You can change these delimiters to any other printable characters (including blanks) by using a DELIMITERS statement in the INSTRUCTIONS section.

The format of the DELIMITERS instruction is

```
DELIMITERS "ab"
```

where DELIMITERS is a keyword, *a* is the opening delimiter, and *b* is the closing delimiter.

Example.

```
DELIMITERS "< >"
```

This instruction causes **INFORMIX-4GL** to use angle brackets (< >) instead of square brackets for screen fields when it displays the compiled form.

Note: In the SCREEN section of a form specification file, you can use only square brackets or the vertical bar to mark display fields. If you use a vertical bar, you *must* include a DELIMITERS statement in the INSTRUCTIONS section that defines two identical symbols as the beginning and ending delimiters for display fields. For example,

```
DELIMITERS "||"
```

or

```
DELIMITERS "  "
```

The Programmer's Environment

There are several ways to create a form specification file from the FORM Menu in the INFORMIX-4GL Programmer's Environment:

- Use the **Generate** option to create a default form specification file.
- Use the **Modify** option to modify a form specification file using a text editor.
- Use the **New** option to create a form specification file from scratch using a text editor.

After you create and compile a form specification using one of these methods, you will see two new files in your current directory. The file with the name you specify and the **.per** extension contains the form specification, while the file with the same name and the **.frm** extension contains the compiled form.

Creating a Default Form Specification File

You can create a default form specification like the one shown in Figure 6-3 by simply supplying **FORM4GL** with the name of a form, a database, and one or more tables. **FORM4GL** uses the database and table information to create a form specification that includes a display field for every column in the specified table or tables. **FORM4GL** labels each display field with the name of the corresponding database column. A default form specification file does not include attributes such as REQUIRED or an INSTRUCTIONS section.

If you create a default form specification, you can save typing time and avoid having to calculate the proper width for display fields. However, the default form specification only contains information for a very simple screen form. You will probably want to customize the default form specification by reorganizing the SCREEN section, adding attributes in the ATTRIBUTES section, and including an INSTRUCTIONS section.

To create a default form specification file, follow these steps:

1. Enter i4gl at the operating-system prompt to enter the **INFORMIX-4GL** Programmer's Environment. The INFORMIX-4GL Menu appears on the screen.

```
INFORMIX-4GL:  Module  Form  Program  Query-language  Exit
Create, modify or run individual 4GL program modules.

----------------------------------------------Press CTRL-W for Help--------
```

2. Select the **Form** option. **INFORMIX-4GL** displays the FORM Menu.

```
FORM:  Modify  Generate  New  Compile  Exit
Change an existing form specification.

----------------------------------------------Press CTRL-W for Help--------
```

3. Select **Generate** from the FORM Menu. **INFORMIX-4GL** displays the CHOOSE DATABASE screen.

```
CHOOSE DATABASE>>
Choose a database with the Arrow Keys, or enter a name, then press Return.

----------------------------------------------Press CTRL-W for Help--------
stores
```

4. Select the appropriate database, either by using the **ARROW** keys to move the cursor or by typing the database name. Then press **RETURN**. **INFORMIX-4GL** displays the following screen:

```
NEW FORM>>
Enter the name you want to assign to the form, then press Return.

-----------------------stores-----------------Press CTRL-W for Help--------
```

5. Type a name for the form specification file and press **RETURN**. The file can have any name you choose. Do not use the **.per** extension when typing the filename, since **FORM4GL** adds the extension automatically. **INFORMIX-4GL** displays the CHOOSE TABLE screen.

```
CHOOSE TABLE>>
Choose the table that the default form is to be based on.

-----------------------stores-----------------Press CTRL-W for Help--------
customer

items

manufact
```

6. Move the cursor to the table that contains the columns you want to include in the form specification file. Press **RETURN** to display the following screen:

```
NEW FORM:  Table-selection-complete  Select-more-tables  Exit
Continue creating a default form with the selected tables.

-----------------------stores-----------------Press CTRL-W for Help--------
```

7. Select **Table-selection-complete** to compile the form or choose **Select-more-tables** to add another table to the form.

   Note: FORM4GL generates a separate SCREEN section for each table you select for a default form specification. If a default form specification is based on more than one table, you will have to modify the form so that it contains only one SCREEN section.

   When you select **Table-selection-complete**, FORM4GL creates a default form specification file and automatically compiles it.

8. Select the **Exit** option to leave the FORM Menu and return to the INFORMIX-4GL Menu.

9. Select the **Exit** option to leave the INFORMIX-4GL Menu and return to the operating-system prompt.

Modifying a Form Specification File

The following instructions explain how to modify a form specification file in the INFORMIX-4GL Programmer's Environment.

1. Enter i4gl at the operating-system prompt to run INFORMIX-4GL. The INFORMIX-4GL Menu appears on the screen.

```
INFORMIX-4GL:  Module  Form  Program  Query-language  Exit
Create, modify or run individual 4GL program modules.

----------------------------------------------Press CTRL-W for Help--------
```

2. Select the **Form** option to display the FORM Menu.

```
FORM:  Modify  Generate  New  Compile  Exit
Change an existing form specification.

-------------------------------------------------Press CTRL-W for Help--------
```

3. Select the **Modify** option from the FORM Menu. **INFORMIX-4GL** displays the following screen:

```
MODIFY FORM>>
Choose a form with Arrow Keys, or enter a name, then press Return.

-------------------------------------------------Press CTRL-W for Help--------

customer

order
```

4. Use the **ARROW** keys to select the form specification file to be modified and press **RETURN**. **INFORMIX-4GL** displays the following screen:

```
USE-EDITOR>>vi
Enter editor name. (RETURN only for default editor).

-------------------------------------------------Press CTRL-W for Help--------
```

5. Enter the name of the editor you want to use or press **RETURN** to accept the default editor. **INFORMIX-4GL** runs the text editor and opens the form specification file that you have specified.

Note: If you have set the DBEDIT environment variable, **INFORMIX-4GL** automatically runs the text editor specified by DBEDIT without displaying the USE-EDITOR screen. Refer to Appendix B for more information about DBEDIT.

6. Change the file as necessary and then leave the editor. **INFORMIX-4GL** displays the following menu:

```
MODIFY FORM:  Compile  Save-and-exit  Discard-and-exit
Compile the form specification.

-----------------------------------------------Press CTRL-W for Help--------
```

7. Select the **Compile** option to compile the form specification file.

Correcting Errors

If **FORM4GL** finds errors in the form specification file, it displays the COMPILE FORM Menu.

```
COMPILE FORM:  Correct  Exit
Correct errors in the form specification.

-----------------------------------------------Press CTRL-W for Help--------
```

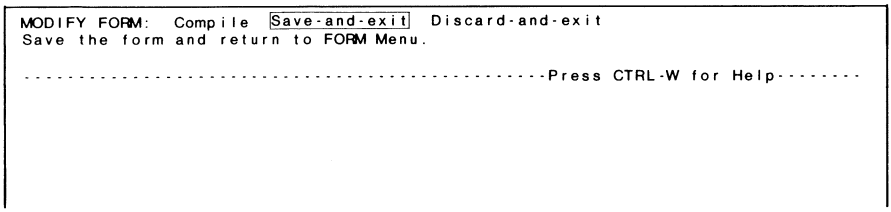
1. Select **Correct** to edit the form specification file.

2. Correct the errors indicated by the error messages. You do not have to delete the error messages. Then leave the editor.

3. Select the **Compile** option from the MODIFY FORM Menu.

Saving the Form Specification

If there are no syntax errors, **FORM4GL** builds the form and displays the MODIFY FORM Menu.



1. Select **Save-and-exit** to save the form specification file. **INFORMIX-4GL** displays the FORM Menu.

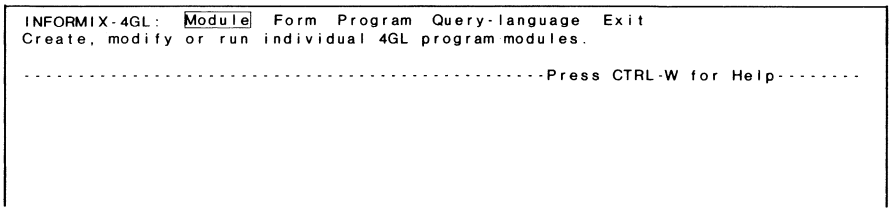
2. Select the **Exit** option to return to the INFORMIX-4GL Menu.

3. Select the **Exit** option to return to the operating-system prompt.

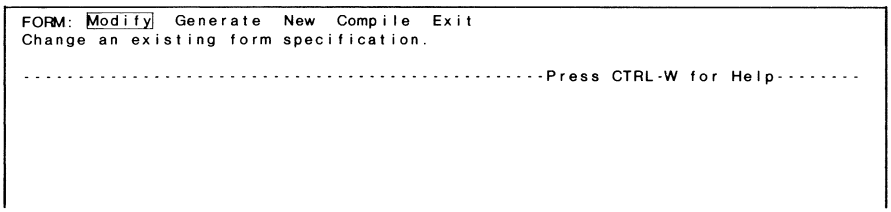
Creating a Form Specification File with a Text Editor

To create a new form specification file with a text editor, follow these steps:

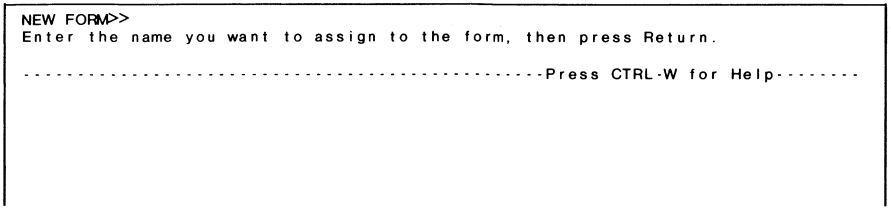
1. Enter i4gl at the operating system prompt to run **INFORMIX-4GL**. The INFORMIX-4GL Menu appears on the screen.



2. Select the **Form** option. **INFORMIX-4GL** displays the FORM Menu.



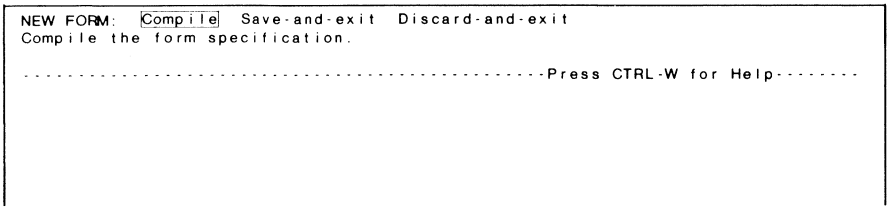
3. Select **New** from the FORM Menu to display the following screen:



4. Type the name of the form specification file and press **RETURN**.

5. If you did not select an editor previously in this session, **INFORMIX-4GL** prompts you to do so. **INFORMIX-4GL** runs the text editor and creates a file with the name you specify.

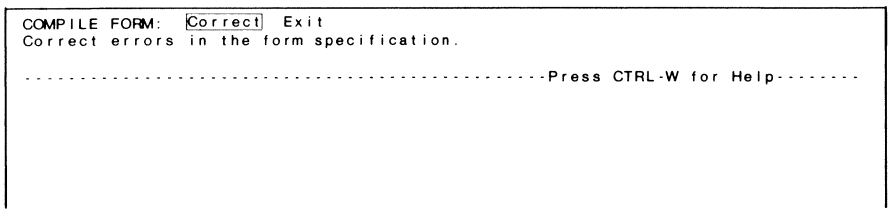
6. Enter the text for the form specification file. (See "Parts of the Form Specification File," earlier in this chapter, for information about form specifications.) Leave the editor with the appropriate exit procedure. **INFORMIX-4GL** displays the NEW FORM Menu.



7. Select the **Compile** option to compile the form specification file.

Correcting Errors

If **FORM4GL** finds errors in the file, it displays the COMPILE FORM Menu.



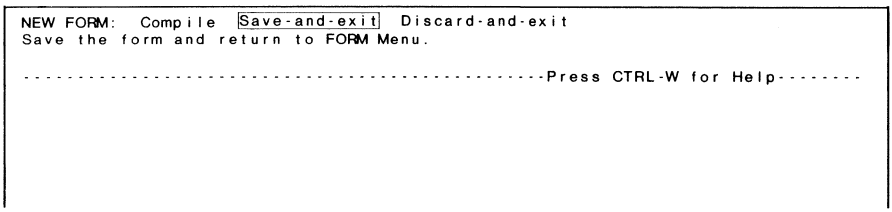
1. Select **Correct** to display the file.

2. Edit the file to remove the errors indicated by the error messages. You do not have to delete the error messages. Then leave the editor.

3. Select the **Compile** option from the NEW FORM Menu.

Saving the Form Specification

If there are no syntax errors, **FORM4GL** compiles the form and returns to the NEW FORM Menu.



1. Select **Save-and-exit** to save the form specification file. **INFORMIX-4GL** displays the FORM Menu.

2. Select **Exit** to return to the INFORMIX-4GL Menu.

3. Select **Exit** to return to the operating-system prompt.

Chapter Summary

- Screen forms are used for entering and displaying data.
- Before you can display a screen form, you must create a form specification file and compile it.
- The form specification file contains the following sections: DATABASE, SCREEN, TABLES, ATTRIBUTES, and INSTRUCTIONS (optional).
- The DATABASE section of the form specification file specifies the database (if any) on which the form is based.
- The SCREEN section of the form specification file describes the layout of the screen form. Display fields are areas on the screen where the user enters data or where a program displays data. Field tags associate display fields in the SCREEN section with field names in the ATTRIBUTES section.
- The TABLES section of the form specification file lists the tables (if any) that contain the columns corresponding to display fields on the form.
- The ATTRIBUTES section of the form specification assigns field names to the field tags that identify display fields. The ATTRIBUTES section optionally assigns one or more attributes to a display field.
- The REQUIRED attribute makes it mandatory for the user to enter data in a field.
- The REVERSE attribute causes a field to be displayed in reverse video.
- The INCLUDE attribute allows you to specify all acceptable values for a field.
- The DEFAULT attribute enables you to specify a default value for a field.

- The UPSHIFT attribute converts all characters entered in a field to uppercase.

- The DOWNSHIFT attribute converts all characters entered in a field to lowercase.

- The AUTONEXT attribute causes the cursor to move to the next field when the current field is full.

- The COMMENTS attribute allows you to display a message when the cursor moves to a field.

- The PICTURE attribute enables you to assign a particular pattern to a CHAR field.

- The NOENTRY attribute prevents data entry in a field.

- The INSTRUCTIONS section of the form specification file is used for defining screen records and for changing delimiters on the screen.

- You can have **FORM4GL** generate a default form specification file by supplying the name of a form, a database, and one or more tables.

- Alternatively, you can modify a form specification or create a form specification file from scratch by using a text editor.

- After you successfully compile a form specification file, you can use the compiled form in **INFORMIX-4GL** programs.

Chapter 7

Working with Screen Forms

Chapter 7 Table of Contents

Chapter Overview 5

Working with Screen Forms 6
Displaying a Screen Form 6
The OPEN FORM Statement 6
The DISPLAY FORM Statement 7
Prompting for Input 8
The MESSAGE Statement 8
Erasing Messages 9
The DISPLAY AT Statement 10
The PROMPT Statement 11
The ERROR Statement 11

Writing Interactive Programs that Use a Screen Form 12
The INPUT Statement 12
INPUT FROM 12
INPUT BY NAME 17
The DISPLAY Statement 18
DISPLAY TO 18
DISPLAY BY NAME 19
Erasing Values in Screen Fields: the CLEAR Statement 20
More About INPUT: the WITHOUT DEFAULTS Clause 20
The SQLCA Record 22
Programs for Entering Data 23
Example 1 24
Example 2 26
Programs for Querying the **customer** Table 27
Example 1 28
Example 2 30
A Program for Updating Rows in the **customer** Table 32
A Program for Deleting Rows from the **customer** Table 35

Chapter Summary 38

Chapter Overview

This chapter explains how to write programs that use screen forms for entering, retrieving, updating, and deleting information in a database. The discussion includes

- How to display a screen form
- How to prompt the user for input
- How to use the INPUT statement for entering data
- How to use the DISPLAY statement to display data
- How to enter information in a database using a screen form
- How to query a database using a screen form
- How to update information in a database using a screen form
- How to delete information in a database using a screen form

The examples in this chapter are based on the following programs and forms included with the demonstration database:

```
ch7add.4gl
ch7add2.4gl
ch7query.4gl
ch7qry2.4gl
ch7del.4gl
ch7upd.4gl
customer.per
```

For complete information about the statements that you use with screen forms, refer to the *INFORMIX-4GL Reference Manual.*

Working with Screen Forms

This chapter explains how to write programs that use the **customer** form that comes with the INFORMIX-4GL demonstration database. Before you work through the following examples, make sure you are familiar with this screen form that is discussed in Chapter 6.

Displaying a Screen Form

To write an INFORMIX-4GL program that displays a screen form, follow these steps:

1. Use the OPEN FORM statement to associate a name with a compiled screen form.

2. Use the DISPLAY FORM statement to display the form on the screen.

The OPEN FORM Statement

The OPEN FORM statement has the following general format

```
OPEN FORM form-name FROM "form-file"
```

where *form-name* is the name for a screen form and *form-file* is the pathname of a compiled screen form.

To assign a name to the **customer** screen form, for example, you could substitute **cust_form** for the *form-name* and **customer** for the *form-file* as follows:

```
OPEN FORM cust_form FROM "customer"
```

As you can see, the **customer** form name appears in quotation marks without the extension **.frm**.

The DISPLAY FORM Statement

Once you have opened a form, you can display it on the screen with a DISPLAY FORM statement like the following:

```
DISPLAY FORM cust_form
```

When **INFORMIX-4GL** encounters this statement, it displays **cust_form** starting on the 3rd line of the screen, reserving the first line for prompts, the second line for messages, the 23rd line for comments, and the 24th line for error messages:

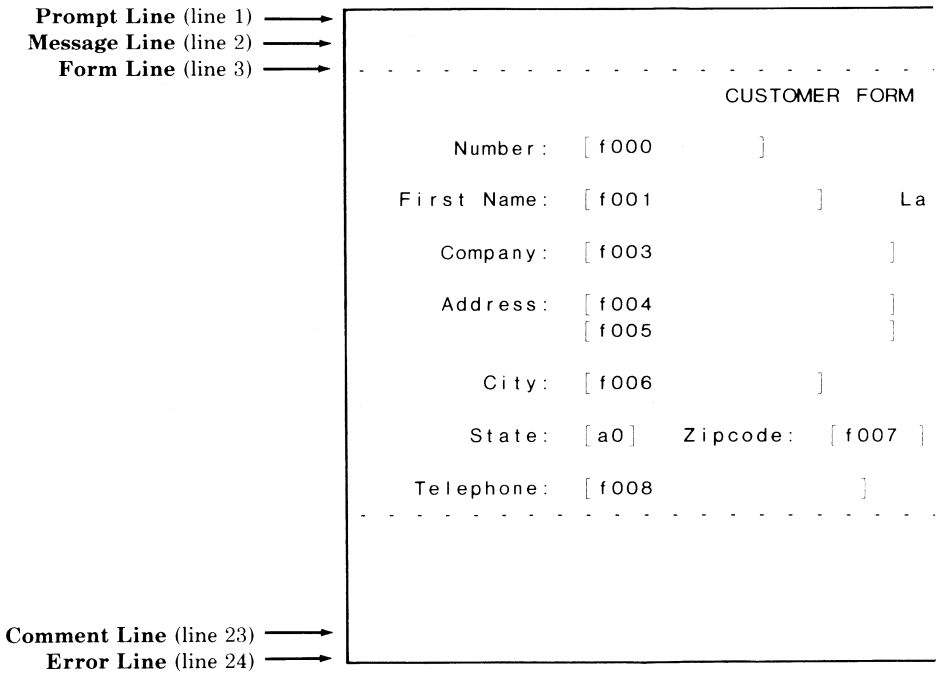


Figure 7-1. The Screen Display

Notes: The previous chapter explained how to use the COMMENT attribute in a form specification for displaying comments on the comment line. This chapter shows you how to use the MESSAGE, PROMPT, and ERROR statements to display text on the message, prompt, and error lines.

Figure 7-1 shows the default screen layout. If you want, you can change the prompt, message, form, comment, and error lines using the OPTIONS statement. See Chapter 8 and the *INFORMIX-4GL Reference Manual* for more information.

Prompting for Input

When you use a screen form in an application, you will want to include instructions telling the user how to work with the form. This section describes three commands, MESSAGE, DISPLAY AT, and PROMPT, for guiding the user through an application.

The MESSAGE Statement

You can use the MESSAGE statement to display a message on the message line (by default, line 2 of the screen). The MESSAGE statement has the following general format:

MESSAGE *display-list*

where *display-list* is a list of one or more variables or string constants, separated by commas. Each item in the display list may include an optional format clause such as CLIPPED or USING.

When INFORMIX-4GL encounters a MESSAGE statement, it prints the values in the *display-list* on the screen. In a simple MESSAGE statement, the *display-list* may consist of a single string constant enclosed in double quotes as shown below:

```
MESSAGE "Enter a customer row."
```

In a more complex MESSAGE statement, the *display-list* might contain both variables and string constants:

```
LET last_name = "Smith"

MESSAGE "Selecting rows for all ",
        "customers with the last name: ", last_name
```

In this example, INFORMIX-4GL displays the message Selecting rows for all customers with the last name: Smith.

Erasing Messages

When you display a message, it remains on the screen until you display another message. This means you can erase an existing message only by displaying a message that consists of zero or more spaces. For example,

```
MESSAGE "Row changed"
MESSAGE ""
```

When INFORMIX-4GL encounters the statements in this example, it displays Row changed on the message line and then immediately erases it. To give the user time to read the Row changed message, insert a SLEEP statement between the two MESSAGE statements as shown below:

```
MESSAGE "Row changed"

SLEEP 3

MESSAGE ""
```

When INFORMIX-4GL executes these statements, it displays the Row changed message, and then waits three seconds before erasing it.

The DISPLAY AT Statement

The MESSAGE statement described in the previous sections only displays text on the message line. If you want to display text at some other location, you can use the DISPLAY AT statement. The format is

```
DISPLAY display-list
  AT row, column
```

where *display-list* is a list of one or more variables or constants, *row* is an integer expression indicating a row on the screen, and *column* is an integer expression indicating a column on the screen.

Note: If either *row* or *column* exceeds the dimensions of the screen (usually 24 rows by 80 columns), **INFORMIX-4GL** will generate a runtime error.

For example, if you want **INFORMIX-4GL** to display a message beginning at the fifth column of the 22nd row, you could use a statement like the following:

```
DISPLAY "Deleting Information on ", cust_name AT 22,5
```

Make sure, however, that you do not display text where you could accidentally overwrite part of your screen form or some other message.

Text that you display with DISPLAY AT remains on the screen until you write over it. To erase existing text to the end of the line, use DISPLAY AT with an argument that consists of zero or more spaces, as shown in the following example:

```
DISPLAY "Enter a customer row." AT 23,1
SLEEP 3
DISPLAY "" AT 23,1
```

Note: You can use the CLIPPED and USING keywords with the DISPLAY AT statement to control the format of the text. (See Chapter 9 and the *INFORMIX-4GL Reference Manual* for more information.)

The PROMPT Statement

You can have the user supply a value for a variable with the PROMPT statement described in Chapter 3. **INFORMIX-4GL** displays all prompts on the prompt line (by default, line 1 of the screen). A prompt remains on the screen until the user enters a response.

The ERROR Statement

The ERROR statement allows you to display an error message on the error line (by default, line 24 of the screen). The format is

ERROR *display-list*

where *display-list* produces a message that fits on one line.

For example, you could use the following ERROR statement to indicate that the user has not entered an acceptable value:

```
ERROR "The following answer is not a valid: ", answ
```

When **INFORMIX-4GL** encounters this ERROR statement, it displays the message in reverse video on the error line and rings the terminal bell. **INFORMIX-4GL** automatically erases the error message when the user types the next key.

Writing Interactive Programs that Use a Screen Form

This section explains how to write programs that allow the user to enter, retrieve, change, and delete data on the **customer** form. These programs rely on two statements, INPUT and DISPLAY, which control the interaction between user and program. With the INPUT statement, you assign values to program variables from data entered by the user on a screen form. The DISPLAY statement allows you to display the values of program variables on a screen form.

The INPUT Statement

INPUT is a very powerful statement that allows you to assign values to program variables from data entered in screen fields. In addition, you can use INPUT to execute **INFORMIX-4GL** statements before or after the user enters data in a screen field, or when the user presses a function or CONTROL key. For example, you can specify valid entries for a screen field or activate a help routine if the user presses the designated help key. This section explains some of the most important features of the INPUT statement. Refer to the *INFORMIX-4GL Reference Manual* for a full description of the INPUT statement.

INPUT FROM

In its simplest form, the INPUT statement has the following format

INPUT *variable-list* FROM *field-list*

where *variable-list* is a list of program variables that correspond to the screen fields in the *field-list*.

When you use the INPUT statement, you should take the following into consideration:

- The INPUT statement always refers to the form that was most recently displayed with the DISPLAY FORM statement.

- Each variable in the *variable-list* should have the same (or compatible) data type as the corresponding field in the *field-list*. (However, data entered in a screen field is checked against the data type of the variable, not the screen field.)

- The order of fields in the *field-list* determines the order in which the cursor moves from field to field on the associated screen form.

- You can refer to a field in two ways: by specifying the full field name (*table.column*) or by specifying the extension (*column*) if it uniquely identifies the field.

For example, suppose you want the user to enter data on the **customer** form, which has the following screen fields:

```
f000 = customer.customer_num;
f001 = customer.fname;
f002 = customer.lname;
f003 = customer.company;
f004 = customer.address1;
f005 = customer.address2;
f006 = customer.city;
a0 = customer.state;
f007 = customer.zipcode;
f008 = customer.phone;
```

To assign the values in the **fname** and **lname** fields to program variables, you could use an INPUT statement like the following:

```
DEFINE p_customer RECORD LIKE customer.*

INPUT p_customer.fname, p_customer.lname
   FROM fname, lname
```

When INFORMIX-4GL encounters the INPUT statement in this example, it displays default values (if any) in the fields with the field names **fname** and **lname**, and then waits for the user to enter values in those fields. INFORMIX-4GL assigns the value in the **fname** field to the variable **p_customer.fname** when the user uses the RETURN or ARROW keys to move the cursor out of the **fname** field. Similarly, INFORMIX-4GL assigns the value in **lname** to **p_customer.lname** when the user moves the cursor out of the **lname** field. The INPUT statement in this example ends when the user presses ESCAPE at any point or RETURN after entering data in the **lname** field.

Notes: If you do not want to use ESCAPE as the input accept key, you can specify another value in the ACCEPT KEY clause of the OPTIONS statement. See the *INFORMIX-4GL Reference Manual* for details.

If the user presses the INTERRUPT key (usually DEL for UNIX systems and CONTROL-C for DOS systems) or the QUIT key (usually CONTROL-\), the program stops unless you have included a DEFER INTERRUPT or a DEFER QUIT statement in your program. See Chapter 10 or the *INFORMIX-4GL Reference Manual* for more information.

Similarly, you can have the user enter values in all fields except **customer_num** with an INPUT statement like the following:

```
DEFINE p_customer RECORD LIKE customer.*

INPUT p_customer.fname, p_customer.lname, p_customer.company,
      p_customer.address1, p_customer.address2, p_customer.city,
      p_customer.state, p_customer.zipcode, p_customer.phone
FROM fname, lname, company,
      address1, address2, city,
      state, zipcode, phone
```

Note: You do not want the user to enter a value in a serial field like **customer_num** since **INFORMIX-4GL** supplies serial numbers automatically when you insert the other values in the **customer** table.

Since the ordering of variables in the **p_customer** record corresponds to the ordering of screen fields in the field list, you can simplify the previous INPUT statement by using the THRU keyword instead of listing the variables in the **p_customer** record individually:

```
INPUT p_customer.fname THRU p_customer.phone
 FROM fname, lname, company,
      address1, address2, city,
      state, zipcode, phone
```

You can simplify the previous INPUT statement even further if the **customer** form specification contains a screen record like the following:

```
TABLES
customer

ATTRIBUTES
f000 = customer.customer_num;
f001 = customer.fname;
f002 = customer.lname;
f003 = customer.company;
f004 = customer.address1;
f005 = customer.address2;
f006 = customer.city;
a0 = customer.state;
f007 = customer.zipcode;
f008 = customer.phone;
END

INSTRUCTIONS
SCREEN RECORD sc_cust (fname THRU phone)
END
```

Since the screen fields in the **sc_cust** screen record correspond to the variables listed after the INPUT keyword, you can substitute **sc_cust.*** for the *field-list* in this example:

```
INPUT p_customer.fname THRU p_customer.phone
  FROM sc_cust.*
```

Note: You must include the **.*** notation after the record name to specify all fields in the screen record. **INFORMIX-4GL** interprets **sc_cust** as a single screen field and displays an error message.

INPUT BY NAME

If a program variable has the same name as a screen field, you can use the INPUT statement with the BY NAME keywords to assign a value to the variable without explicitly naming the screen field. For example, the variables in the **p_customer** record have the same names as the corresponding fields in the **customer** screen form:

Variable Name	Field Name
p_customer.customer_num	customer_num
p_customer.fname	fname
p_customer.lname	lname
p_customer.company	company
p_customer.address1	address1
p_customer.address2	address2
p_customer.city	city
p_customer.state	state
p_customer.zipcode	zipcode
p_customer.phone	phone

Note: INFORMIX-4GL looks only at extensions to determine whether a variable name matches a field name.

This means, for example, that you can have the user supply values for all components of **p_customer** except **p_customer.customer_num** with the following INPUT statement:

```
INPUT BY NAME p_customer.fname THRU p_customer.phone
```

When INFORMIX-4GL encounters this statement, it waits for the user to enter values in the screen fields with field names that match the named variables (**fname**, **lname**, **company**, **address1**, **address2**, **city**, **state**, **zipcode**, and **phone**). INFORMIX-4GL assigns a value to a variable when the user moves the cursor out of the screen field with the matching name.

An earlier section explained how to use the DISPLAY statement to display information at a particular row and column on the screen. This section describes how to use DISPLAY to display the values of program variables in screen fields.

DISPLAY TO

In simplest form, the DISPLAY statement has the following format:

DISPLAY *display-list* TO *field-list*

where *display-list* is a list of program variables (or constants) that correspond to the screen fields in the *field-list*.

When you use the DISPLAY statement, make sure that each program variable has the same (or compatible) data type as the corresponding screen field.

For example, you can use the following DISPLAY statement to display the name John Smith on the **customer** form:

```
LET p_customer.fname = "John"
LET p_customer.lname = "Smith"
DISPLAY p_customer.fname, p_customer.lname TO fname, lname
```

Often, you will want to select values for the variables from a table and then display them on the screen form.

In the following statements, for example, INFORMIX-4GL selects a row from the **customer** table, stores it in the record **p_customer**, and then displays the values in **p_customer** in the corresponding screen fields:

```
DEFINE p_customer RECORD LIKE customer.*

SELECT * INTO p_customer.* FROM customer
         WHERE customer_num = 103

DISPLAY p_customer.*
TO customer_num, fname, lname, company,
      address1, address2, city,
      state, zipcode, phone
```

You can simplify the DISPLAY statement in this example even further by substituting the default screen record, **customer.***, for the *field-list* as in this example:

```
DISPLAY p_customer.* TO customer.*
```

DISPLAY BY NAME

If a program variable has the same name as a screen field, you can use the DISPLAY statement with the BY NAME keywords to display the value of the variable without explicitly naming the screen field. For example, you can display an entire row of the **customer** table with the following statements:

```
SELECT * INTO p_customer.* FROM customer
     WHERE customer_num = 103

DISPLAY BY NAME p_customer.*
```

When INFORMIX-4GL encounters the DISPLAY statement in this example, it displays the value of each variable in the **p_customer** record in the screen field with the matching field name.

Erasing Values in Screen Fields: the CLEAR Statement

When you display a value to a screen field or input a value from a screen field, it remains on screen until you display or input another value, or until you erase it with the CLEAR statement. The CLEAR statement can take the following forms:

```
CLEAR SCREEN

CLEAR FORM

CLEAR field-list
```

You can use CLEAR SCREEN to clear the entire screen (including the prompt, message, and error lines), CLEAR FORM to clear the values in all screen fields, or CLEAR *field-list* to clear one or more screen fields.

If, for example, you want to erase the values in the **fname** and **lname** fields, use a statement like the following:

```
CLEAR fname, lname
```

Similarly, you could erase the values in all screen fields with the following statement:

```
CLEAR FORM
```

More About INPUT: the WITHOUT DEFAULTS Clause

When INFORMIX-4GL encounters the INPUT statements described above, it displays blanks or default values from the form (if any) in the specified fields before the user begins entering data. This is appropriate when you are requesting input prior to inserting a new row into a table.

If, however, you are requesting input prior to updating an existing row, you will want to display the row and leave the current values on screen when the user begins changing data. To do this, you can use the INPUT statement with the WITHOUT DEFAULTS clause. When INFORMIX-4GL encounters this type of INPUT statement, it displays the current values in the *variable-list* in the associated screen fields before accepting input from the user. For example,

```
SELECT * INTO p_customer.* FROM customer
    WHERE customer_num = 103

DISPLAY BY NAME p_customer.*

INPUT BY NAME p_customer.fname THRU p_customer.lname
    WITHOUT DEFAULTS
```

When INFORMIX-4GL encounters the statements in this example, it performs the following operations:

1. Selects the row for customer 103 and stores it in the record **p_customer**.

2. Displays the values in **p_customer** on the screen form.

3. Leaves the current values of **p_customer.fname** through **p_customer.phone** on screen while waiting for the user to add or change data in the associated screen fields.

4. Assigns the values in the screen fields **fname** through **lname** to the corresponding program variables.

The SQLCA Record

You now have most of the tools that you will need to write programs for entering, retrieving, updating, and deleting information through screen forms. Before you study the sample programs in this chapter, however, you should become familiar with the SQLCA record. This record is defined by **INFORMIX-4GL** and stores information about the most recently executed **RDSQL** statement. (**RDSQL** statements are those statements you use to create and maintain databases; examples include CREATE TABLE, INSERT, SELECT, UPDATE, and DELETE.)

For example, if **INFORMIX-4GL** automatically assigns a value to a serial column during an INSERT statement, it records the serial number in the SQLERRD[2] component of the SQLCA record. The sample programs in this chapter use SQLCA.SQLERRD[2] to determine the customer number that **INFORMIX-4GL** automatically assigns to each row it inserts into the **customer** table. In the following example, the LET statement assigns the serial number in SQLCA.SQLERRD[2] to **p_customer.customer_num** immediately after the INSERT statement that adds a row to the **customer** table:

```
LET p_customer.customer_num = 0
INSERT INTO customer VALUES (p_customer.*)
LET p_customer.customer_num = SQLCA.SQLERRD[2]
```

For more information about the SQLCA record, see Chapter 2 of the *INFORMIX-4GL Reference Manual.*

Programs for Entering Data

This section discusses the **ch7add** and **ch7add2** programs that are included with the demonstration database. The first program allows the user to enter information for one customer on a screen form, while the second program allows the user to enter information for zero or more customers.

Example 1

The **ch7add** program lets the user enter a customer row on the **customer** form, inserts the row into the **customer** table, and then displays the customer number for the newly added row.

```
DATABASE stores
GLOBALS
    DEFINE          p_customer RECORD LIKE customer.*
END GLOBALS

MAIN
    OPEN FORM cust_form FROM "customer"
    DISPLAY FORM cust_form
    MESSAGE "Enter customer information."
    CALL enter_cust()
    MESSAGE "End program."
    SLEEP 3
    CLEAR SCREEN
END MAIN

FUNCTION enter_cust()
    INPUT BY NAME p_customer.fname THRU p_customer.phone
    LET p_customer.customer_num = 0
    INSERT INTO customer VALUES (p_customer.*)
    LET p_customer.customer_num = SQLCA.SQLERRD[2]
    DISPLAY p_customer.customer_num TO customer_num
    MESSAGE "Row added."
    SLEEP 3
    MESSAGE ""
END FUNCTION
```

As you can see, this program contains a function called **enter_cust** that performs the following operations:

1. Assigns values to all variables in the **p_customer** record except **p_customer.customer_num** from data entered on the screen form.

2. Assigns the value 0 to **p_customer.customer_num** as a place holder for the SERIAL value that INFORMIX-4GL inserts automatically.

3. Inserts the values in the **p_customer** record into the **customer** table.

4. Assigns the value in SQLCA.SQLERRD[2] to **p_customer.customer_num**. (SQLCA.SQLERRD[2] contains the serial number of the row that INFORMIX-4GL just inserted into the **customer** table.)

5. Displays the serial number for the newly added row in the **customer_num** field, along with the message Row added.

Example 2

The **ch7add2** program allows the user to enter zero or more customer rows through a screen form:

```
DATABASE stores

GLOBALS

    DEFINE     p_customer RECORD LIKE customer.*

END GLOBALS


MAIN

    DEFINE answer CHAR(1)

    OPEN FORM cust_form FROM "customer"

    DISPLAY FORM cust_form

    PROMPT "Do you want to enter a customer row (y/n) ? "
        FOR answer

    WHILE answer = "y"

        CALL enter_cust()

        PROMPT "Do you want to enter another customer row (y/n) ? "
            FOR answer

    END WHILE

    MESSAGE "End program."

    SLEEP 3

    CLEAR SCREEN

END MAIN


FUNCTION enter_cust()

    CLEAR FORM

    INPUT BY NAME p_customer.fname THRU p_customer.phone

    LET p_customer.customer_num = 0

    INSERT INTO customer VALUES (p_customer.*)

    LET p_customer.customer_num = SQLCA.SQLERRD[2]

    DISPLAY p_customer.customer_num TO customer_num

    MESSAGE "Row added."

    SLEEP 3

    MESSAGE ""

END FUNCTION
```

As you can see, this program is almost identical to the first sample program, except for the inclusion of a WHILE loop in the main program and a CLEAR FORM statement in the **enter_cust** function. The WHILE statement allows the user to decide how many customer rows to enter in the database. The CLEAR FORM statement clears all the screen fields on the **customer** form before the user enters information for a new customer.

Programs for Querying the **customer** *Table*

This section describes the **ch7query** and **ch7qry2** programs included with the demonstration database. Both programs select rows from the **customer** table and display them on the screen form.

Example 1

The **ch7query** program selects all rows from the **customer** table and displays them at the user's request:

```
DATABASE stores

MAIN

    DEFINE      p_customer RECORD LIKE customer.*,
                answer     CHAR(1),
                found      SMALLINT

    OPEN FORM cust_form FROM "customer"

    DISPLAY FORM cust_form

    MESSAGE "Selecting all rows from the customer table . . ."

    SLEEP 3

    MESSAGE ""

    DECLARE q_curs CURSOR FOR
              SELECT * FROM customer

    LET found = FALSE

    FOREACH q_curs INTO p_customer.*

              LET found = TRUE

              DISPLAY BY NAME p_customer.*

              PROMPT "Do you want to see the ",
                   "next customer row (y/n) ? "
                    FOR answer

              IF answer = "n" THEN

                   EXIT FOREACH

              END IF

    END FOREACH


    IF found = FALSE THEN

                MESSAGE "No rows found.  End program."
    ELSE

                IF answer = "y" THEN

                     MESSAGE "No more rows.  End program."

                ELSE

                     MESSAGE "End program."

                END IF

    END IF

    SLEEP 3

    CLEAR SCREEN

END MAIN
```

As you can see, this program contains a FOREACH loop that performs the following operations:

1. Selects a row from the **customer** table.

2. Assigns the value TRUE to the variable **found** to indicate that at least one row has been selected.

3. Displays the customer row on the screen form.

4. Asks whether the user wants to see the next customer row.

5. Exits the FOREACH loop if the user enters n in response to the prompt.

6. Repeats the previous steps until all rows have been processed, or until the user enters n in response to the prompt.

Example 2

The **ch7qry2** program selects rows based on a last name that the user supplies:

```
DATABASE stores

MAIN

    DEFINE p_customer RECORD LIKE customer.*,
    last_name           CHAR(15),
    answer              CHAR(1),
    found               SMALLINT

    OPEN FORM cust_form FROM "customer"

    DISPLAY FORM cust_form

    PROMPT "Enter a last name: " FOR last_name

    MESSAGE "Selecting rows for customers with the last name: ",
             last_name

    SLEEP 3

    MESSAGE ""

    DECLARE q_curs CURSOR FOR
            SELECT * FROM customer
                 WHERE lname MATCHES last_name

    LET found = FALSE

    FOREACH q_curs INTO p_customer.*

            LET found = TRUE

            DISPLAY BY NAME p_customer.*

            PROMPT "Do you want to see the ",
                   "next customer row (y/n) ? "
                   FOR answer

            IF answer = "n" THEN

                   EXIT FOREACH

            END IF

    END FOREACH
```

```
    IF found = FALSE THEN
        MESSAGE "No rows found.   End program."
    ELSE
        IF answer = "y" THEN
            MESSAGE "No more rows.  End program."
        ELSE
            MESSAGE "End program."
        END IF
    END IF
    SLEEP 3
    CLEAR SCREEN
END MAIN
```

This program is almost identical to the previous program, except for the inclusion of a PROMPT statement that allows the user to enter a value for the variable **last_name**. The program then selects rows from the **customer** table based on the name the user supplied and displays them upon request.

A Program for Updating Rows in the **customer** *Table*

The **ch7upd** program included with the demonstration database selects all rows from the **customer** table and allows each row to be displayed and changed at the user's discretion.

```
DATABASE stores

GLOBALS

        DEFINE p_customer RECORD LIKE customer.*

END GLOBALS


MAIN

    DEFINE answer    CHAR(1),
           found     SMALLINT

    OPEN FORM cust_form FROM "customer"

    DISPLAY FORM cust_form

    MESSAGE "Selecting all rows from the customer table . . ."

    SLEEP 3

    MESSAGE ""

    DECLARE q_curs CURSOR FOR
            SELECT * FROM customer

    LET found = FALSE

    FOREACH q_curs INTO p_customer.*

            LET found = TRUE

            DISPLAY BY NAME p_customer.*

            PROMPT "Do you want to add or ",
                   "change any information (y/n) ?  "
                   FOR answer

            IF answer = "y" THEN

                   CALL change_row()
            END IF

            PROMPT "Do you want to see the ",
                   "next customer row (y/n) ? "
                        FOR answer

            IF answer = "n" THEN

                   EXIT FOREACH

            END IF

    END FOREACH
```

```
        IF found = FALSE THEN

                MESSAGE "No rows found.  End program."
        ELSE

                IF answer = "y" THEN

                        MESSAGE "No more rows.  End program."

                ELSE

                        MESSAGE "End program."

                END IF

        END IF

        SLEEP 3

        CLEAR SCREEN

END MAIN



FUNCTION change_row()

         INPUT BY NAME p_customer.fname THRU p_customer.phone
              WITHOUT DEFAULTS

         UPDATE customer
         SET customer.* = p_customer.*
         WHERE customer_num = p_customer.customer_num

         MESSAGE "Row updated."

         SLEEP 3

         MESSAGE ""

END FUNCTION
```

This program is similar to the sample programs for querying the **customer** table. If you examine the FOREACH statement in this example, however, you will notice a new prompt and a conditional call to the function **change_row**:

```
PROMPT "Do you want to add or ",
       "change any information (y/n) ? "
       FOR answer

IF answer = "y" THEN

       CALL change_row()

END IF
```

If the user enters y in response to this prompt, INFORMIX-4GL invokes the **change_row** function that performs the following tasks:

1. Uses the INPUT statement without defaults to assign values to **p_customer.fname** through **p_customer.phone** from data on the screen form.

2. Updates all columns in the current row except **customer_num**.

   Note: SERIAL columns cannot be updated. If you use the following form of the UPDATE statement

   UPDATE *table-name*
   SET *table-name.** = *record-name.**
   [WHERE *condition*]

   INFORMIX-4GL automatically skips any SERIAL column in the column list produced by *table-name.** and the corresponding value in the expression list produced by *record-name.**.

3. Displays a message indicating that the row has been changed.

A Program for Deleting Rows from the **customer** *Table*

The **ch7del** program included with the demonstration database selects all rows from the **customer** table and allows each row to be displayed and deleted at the user's discretion.

```
DATABASE stores

GLOBALS

    DEFINE p_customer RECORD LIKE customer.*

END GLOBALS


MAIN
    DEFINE answer      CHAR(1),
           found       SMALLINT

    OPEN FORM cust_form FROM "customer"

    DISPLAY FORM cust_form

    MESSAGE "Selecting all rows from the customer table . . ."

    SLEEP 3

    MESSAGE ""

    DECLARE q_curs CURSOR FOR
            SELECT * FROM customer

    LET found = FALSE

    FOREACH q_curs INTO p_customer.*

            LET found = TRUE

            DISPLAY BY NAME p_customer.*

            PROMPT "Do you want to delete ",
                      "this customer row (y/n) ?  "
                      FOR answer

            IF answer = "y" THEN

                      CALL delete_row()

            END IF

            PROMPT "Do you want to see the ",
                      "next customer row (y/n) ? "
                      FOR answer

            IF answer = "n" THEN

                      EXIT FOREACH

            END IF

    END FOREACH
```

```
    IF found = FALSE THEN
            MESSAGE "No rows found.  End program."
    ELSE
            IF answer = "y" THEN
                    MESSAGE "No more rows.  End Program."
            ELSE
                    MESSAGE "End program."
            END IF
    END IF
    SLEEP 3
    CLEAR SCREEN
END MAIN
FUNCTION delete_row()
    DELETE FROM customer
            WHERE customer_num = p_customer.customer_num
    CLEAR FORM
    MESSAGE "Row deleted."
    SLEEP 3
    MESSAGE ""
END FUNCTION
```

Again, this program is similar to the sample programs for querying the **customer** table. However, the FOREACH statement in this example contains a new prompt and a conditional call to the function **delete_row**:

```
PROMPT "Do you want to delete ",
       "this customer row (y/n) ? "
       FOR answer

IF answer = "y" THEN

       CALL delete_row()

END IF
```

If the user enters y in response to this prompt, INFORMIX-4GL invokes the **delete_row** function that performs the following operations:

1. Deletes the current row from the **customer** table.
2. Clears the **customer** form.
3. Displays a message that the row has been deleted.

Chapter Summary

- You can use screen forms for entering, retrieving, updating, and deleting information in a database.
- You can associate a form name with a compiled screen form using the OPEN FORM statement.
- You can display a form on the screen with the DISPLAY FORM statement.
- You can use the MESSAGE statement to display a message on the message line of your terminal.
- You can use the DISPLAY AT statement to display text at a specified location on the screen.
- You can prompt the user to supply a value for a program variable with the PROMPT statement.
- With the ERROR statement you can display an error message on the error line and ring the terminal bell.
- You can use the INPUT statement to assign values to program variables from data entered by the user on a screen form.
- You can display the values of program variables in screen fields with the DISPLAY statement.

Chapter 8

Menus, Options, and Help

Chapter 8 Table of Contents

Chapter Overview 5

Creating Menus with INFORMIX-4GL 6
The Appearance of a Menu 6
Selecting Menu Options 8
Requesting Help 9

Creating a Menu 9
The MENU Statement 10
The MENU and END MENU Keywords 11
The COMMAND Clause 12

Changing the Message and Prompt Lines 18
The OPTIONS Statement 18
The MESSAGE LINE Clause 18
The PROMPT LINE Clause 20

Creating Help Messages 21
Requesting Online Help 21
Creating a Help File 23
The **mkmessage** Utility 25
Using the Help File with a Program 25
More About the OPTIONS Statement 26
Specifying a Help Message for a Menu Option 27

A Sample Menu Program 29
The GLOBALS Statement 35
The MAIN Statement 35
The **show_menu** Function 36
The **enter_row** Function 38
The **query_data** Function 39
The **change_data** Function 40
The **delete_row** Function 41

Nested Menus 42
The **show_menu** Function 44
The **query_data** Function 45
The **view_cust** Function 46

Chapter Summary 48

Chapter Overview

This chapter describes how to design menus, change default layouts for screens, and create help files using **INFORMIX-4GL.** Subjects discussed include

- What a menu looks like
- How the user selects options from the menu
- How to create a menu using the MENU statement
- How to use the OPTIONS statement to change the prompt and message lines and to specify a help file and help key
- How to create help messages that the user can display from the menu
- How to create a menu within a menu

The examples in this chapter are based upon the following programs, forms, and help files included with the demonstration database:

```
custmenu.4gl
c_menu2.4gl
custhelp.msg
customer.per
```

For complete information about the statements used in this chapter, see the *INFORMIX-4GL Reference Manual.*

Creating Menus with INFORMIX-4GL

INFORMIX-4GL allows you to design menus that are similar to those of the INFORMIX-4GL Programmer's Environment. This chapter shows what an INFORMIX-4GL menu looks like and explains how you can create menus using the MENU statement.

The Appearance of a Menu

The layout of a menu is shown below:

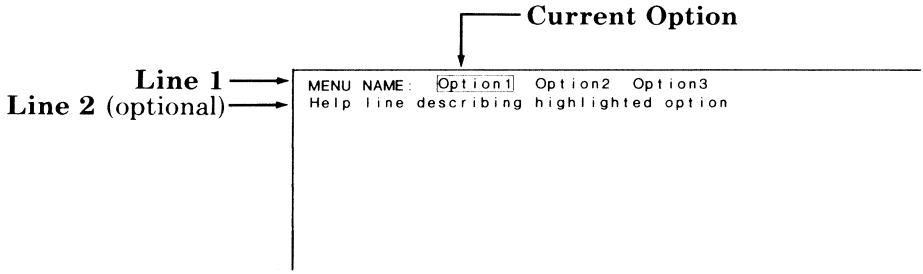


Figure 8-1. Menu Layout

When INFORMIX-4GL initially displays a menu, the menu name and menu options appear on the first line of the screen. One option is always marked as the current option. It is highlighted by a cursor if the terminal can display highlighting or is surrounded by angle brackets if it cannot. A help line describing the current option may appear on line 2.

Whenever the user presses the **SPACEBAR** or **ARROW** keys, the cursor moves from one option to the next and the help line (if any) changes accordingly.

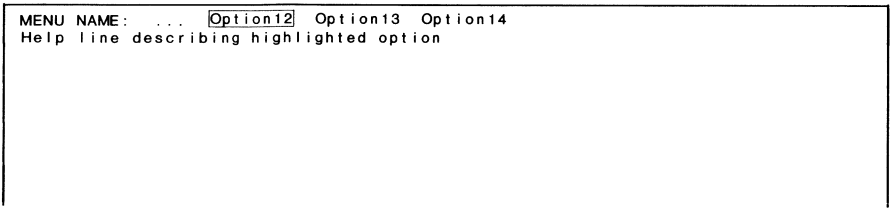
If the length of the menu exceeds the number of characters that the screen can display on a single line, **INFORMIX-4GL** displays the first "page" of options followed by an ellipsis (...) indicating that additional options exist. For example,

```
MENU NAME:  Option1  Option2  Option3  Option4  Option5  Option6  ...
Help line describing highlighted option
```

If the user presses the **SPACEBAR** or **RIGHT ARROW** key to move past the rightmost option (**Option6** in this case), **INFORMIX-4GL** displays the next "page" of menu options. In the following example, the ellipses at each end indicates that more menu options exist in both directions.

```
menu-name:  ...  Option7  Option8  Option9  Option10  Option11  ...
Help line describing highlighted option
```

If the user moves the highlight to the right past Option11 in this example, INFORMIX-4GL might display another page of menu options like the following:



Since no ellipsis appears to the right of Option14, the user has come to the end of the menu options. The user can display the previous page of menu options again by using the **LEFT ARROW** key to move the highlight past the leftmost option in this example.

The **UP ARROW** key moves the highlight to the first item on the previous page; the **DOWN ARROW** key moves the highlight to the first item on the subsequent page.

Selecting Menu Options

The user selects an item from the menu in either of two ways:

- By pressing the **SPACEBAR** or using the **ARROW** keys to move the cursor to an option and then pressing **RETURN**
- By typing the first letter of the option (regardless of whether the option currently appears on the screen).

In the latter case, the user does not have to press **RETURN** to select the menu option.

Requesting Help

If you have linked a help file to the menu, the user can display a help message for each menu option by pressing the help key. By default this key is CONTROL-W. You can change the default with the OPTIONS statement, as explained later in this chapter.

Creating a Menu

In this section, you will learn how to create a menu using the MENU statement. This section includes examples from the **custmenu** program shown in the section "A Sample Menu Program" later in this chapter.

The MENU Statement

The MENU statement determines the appearance of the menu and provides a structure in which to place statements that perform the activities described by the menu options.

You can use the MENU statement to design a menu with the layout shown in Figure 8-1. In simplest form, the MENU statement has the following format:

```
MENU "menu-name"

        COMMAND "menu-option" ["help-line"]
                statement
                ...
                [CONTINUE MENU]
                ...
                [NEXT OPTION "menu-option"]
                ...
                [EXIT MENU]
                ...

        COMMAND "menu-option" ["help-line"]
                statement
                ...
END MENU
```

where *menu-name* is the title of the menu, *menu-option* is the name of a menu option, *help-line* is an optional one-line message that describes the *menu-option*, and *statement* is an **INFORMIX-4GL** statement.

When **INFORMIX-4GL** encounters a MENU statement, it performs the following operations:

- Displays the menu name, menu options, and help line (if any) for the highlighted option. Menu options appear according to the order of the COMMAND clauses in the MENU statement.

- When the user selects an option from the menu, INFORMIX-4GL executes the statements in the COMMAND clause for the chosen option.

- After INFORMIX-4GL executes all the statements in a COMMAND clause, it redisplays the menu so that the user may choose another option.

- If INFORMIX-4GL encounters a CONTINUE MENU statement in a COMMAND clause, it immediately redisplays the menu without executing the remaining statements (if any).

- If INFORMIX-4GL encounters an EXIT MENU statement in a COMMAND clause, it executes the first statement following the END MENU keywords.

The MENU statement continues to execute in this way until INFORMIX-4GL encounters the EXIT MENU keywords or until the user presses the INTERRUPT or QUIT keys (assuming you have not used the DEFER INTERRUPT or DEFER QUIT statement in your program).

The following sections explain the parts of the MENU statement in more detail.

The MENU and END MENU Keywords

The MENU statement begins with the MENU keyword followed by the name of the menu in quotation marks. For example,

```
MENU "CUSTOMER"
```

places the title CUSTOMER in the upper left corner of the screen.

The END MENU keywords indicate the end of the MENU statement. END MENU, however, does not cause the MENU statement to terminate. **INFORMIX-4GL** ends the MENU statement when it encounters the EXIT MENU keywords or when the user presses the INTERRUPT or QUIT keys.

The COMMAND Clause

A simple COMMAND clause has the following format:

```
COMMAND "menu-option" ["help-line"]
   statement
   ...
```

You must include at least two COMMAND clauses in the MENU statement. The order of the COMMAND clauses in the MENU statement determines the order in which menu options appear on the screen. If you include a help line after a menu option in a COMMAND clause, it is displayed on line 2 whenever the user moves the cursor to that option.

Naming Menu Options. When you are choosing names for menu options, you should make sure that each menu option begins with a different letter since the user will choose an option by its first letter. If you do not want the user to choose an option by typing its first letter, you must include a KEY clause that lists the letter the user may type to select the given option. (See the *INFORMIX-4GL Reference Manual* for details.)

When **INFORMIX-4GL** displays a menu, it adds a colon (:) and a space after the menu name, as well as a space before and after each option.

A Sample MENU Statement. The following MENU statement contains COMMAND clauses for five menu options (**Add**, **Query**, **Modify**, **Delete**, and **Exit**).

```
MENU "CUSTOMER"

        COMMAND "Add" "Add a new row."
            statements

        COMMAND "Query" "Search for a row."
            statements

        COMMAND "Modify" "Modify a row."
            statements

        COMMAND "Delete" "Delete a row."
            statements

        COMMAND "Exit" "Leave the menu."
             EXIT MENU

END MENU
```

This MENU statement creates the menu shown in Figure 8-2.

```
CUSTOMER: [Add] Query Modify Delete Exit
Add a new row.
```

Figure 8-2. A Sample Menu

The cursor, or highlighting, is initially located on the **Add** option because it is the menu option listed in the first COMMAND clause. Subsequent menu options are listed on the screen in the same order that their COMMAND clauses occur in the MENU statement.

Statements in a COMMAND Clause. Within each COMMAND clause, include a series of statements that perform the activity described by the given menu option. These statements can be function calls, loops, conditional statements—whatever is necessary to carry out the activity of the option. You can even call a function that displays another menu to create a menu interface for your program. Try to keep the number of statements in each COMMAND clause to a minimum, however, so that the structure of the MENU statement remains clear. In most cases, a COMMAND clause will invoke a function that performs the activity described by the menu option.

The following example shows part of a MENU statement that enables a user to enter customer data on the **customer** form:

```
MENU "CUSTOMER"
    COMMAND "Add" "Add a new row."
        LET answer = "y"
        WHILE answer = "y"
            CALL enter_row()
            PROMPT "Do you want to enter another row (y/n)? "
                FOR answer
        END WHILE
    COMMAND "Query" "Search for a row"
        ...
END MENU
```

If the user selects the **Add** option, INFORMIX-4GL performs the following operations:

- Assigns the value y to the variable **answer**.
- Executes the statements in the WHILE loop since the initial value of **answer** is y.
- Calls the **enter_row** function, which allows the user to enter a row on the **customer** form and then inserts the row into the **customer** table.
- Prompts the user to supply a value for the variable **answer**.

- If the user enters a character other than y in response to the prompt, **INFORMIX-4GL** leaves the WHILE loop and redisplays the menu so that the user can select another option.

- If the user enters y in response to the prompt, **INFORMIX-4GL** executes the statements in the WHILE loop again.

The CONTINUE MENU Statement. You can use the CONTINUE MENU statement if you sometimes want **INFORMIX-4GL** to ignore the remaining statements in the COMMAND clause and redisplay the menu so that the user can select another option. Here is an example:

```
MENU "CUSTOMER"

    COMMAND "Add" "Add a new row."

        PROMPT "Do you want to enter a row (y/n)? "
            FOR answer

        IF answer = "n" THEN

            CONTINUE MENU

        END IF

        WHILE answer = "y"

            CALL enter_row()

            PROMPT "Do you want to enter another row (y/n)? "
                FOR answer

        END WHILE

    COMMAND "Query" "Search for a row."
        ...

END MENU
```

If the user enters n in response to the first prompt in this COMMAND clause, **INFORMIX-4GL** redisplays the menu without executing the remaining statements.

The NEXT OPTION Statement. When **INFORMIX-4GL** finishes executing the statements in a COMMAND clause, it leaves the highlight on the menu option for that clause. If you want **INFORMIX-4GL** to highlight another option instead of the current option, you should include a NEXT OPTION statement in the COMMAND clause.

The format of the NEXT OPTION statement is

NEXT OPTION "*menu-option*"

where *menu-option* is the menu option that you want INFORMIX-4GL to highlight when it redisplays the menu.

The following example shows a COMMAND clause that contains a NEXT OPTION statement:

```
MENU "CUSTOMER"
      COMMAND "Query" "Search for a row."
            CALL query_data()
            NEXT OPTION "Modify"
      COMMAND "Modify" "Modify a row."
            . . .
END MENU
```

If the user selects the **Query** option, INFORMIX-4GL executes the statements in the **query_data** function and then redisplays the menu with the highlight on the **Modify** option. (If the COMMAND clause for the **Query** option did not include a NEXT OPTION clause, INFORMIX-4GL would highlight the **Query** option when it redisplays the menu.)

The EXIT MENU Statement. Use the EXIT MENU keywords at any point in the MENU statement where you want to leave the menu. When INFORMIX-4GL encounters EXIT MENU, it terminates the MENU statement and executes the first statement following the END MENU keywords.

For example, you can use EXIT MENU with an IF statement to let the user decide whether to leave the menu after adding customer rows:

```
MENU "CUSTOMER"
    COMMAND "Add" "Add a new row."
        LET answer "y"
        WHILE answer = "y"
            CALL enter_row()
            PROMPT "Do you want to enter another row (y/n)? "
                FOR answer
        END WHILE
        PROMPT "Do you want to leave the menu (y/n)? "
            FOR answer
        IF answer = "y" THEN
            EXIT MENU
        END IF
    COMMAND "Query" "Search for a row."
        ...
END MENU
```

If the user enters y in response to the second prompt in this example, **INFORMIX-4GL** exits the menu; otherwise **INFORMIX-4GL** redisplays the menu to let the user select another option.

You can also create an **Exit** option for your menu, as follows:

```
MENU "CUSTOMER"
    COMMAND "Add" "Add a new row."
        ...
    COMMAND "Exit" "Leave the CUSTOMER menu"
        EXIT MENU
END MENU
```

When the user selects the **Exit** option, **INFORMIX-4GL** stops this MENU statement.

Changing the Message and Prompt Lines

In the previous sections you learned how to design menus using the MENU statement. This section explains how to use the OPTIONS statement to change the lines where messages and prompts are displayed so they do not overwrite your menu.

The OPTIONS Statement

The OPTIONS statement enables you to change the default layout for screens and the default keys for screen operations. For example, you can use the OPTIONS statement to specify lines for prompts, messages, forms, comments, and errors, or to designate keys for insertion, deletion, and help.

The following table shows the format for an OPTIONS statement that includes the optional clauses to be discussed in this section:

```
OPTIONS{ MESSAGE LINE integer |
         PROMPT LINE integer } [,...]
```

The MESSAGE LINE Clause

By default, INFORMIX-4GL displays the text in a MESSAGE statement on the second line of the screen. This means that a message will temporarily erase part of your menu unless you change the default with the OPTIONS statement.

Use the MESSAGE LINE clause to designate the screen line where you want a message displayed by the MESSAGE statement to appear. The format is

```
OPTIONS MESSAGE LINE integer
```

where *integer* is a line on the screen.

For example, suppose you want to display a message telling the user how to select a menu option. Since you do not want the message to overwrite the menu, you should designate another line for messages as follows:

```
OPTIONS MESSAGE LINE 22
```

Then you can display a message and a menu at the same time as shown below:

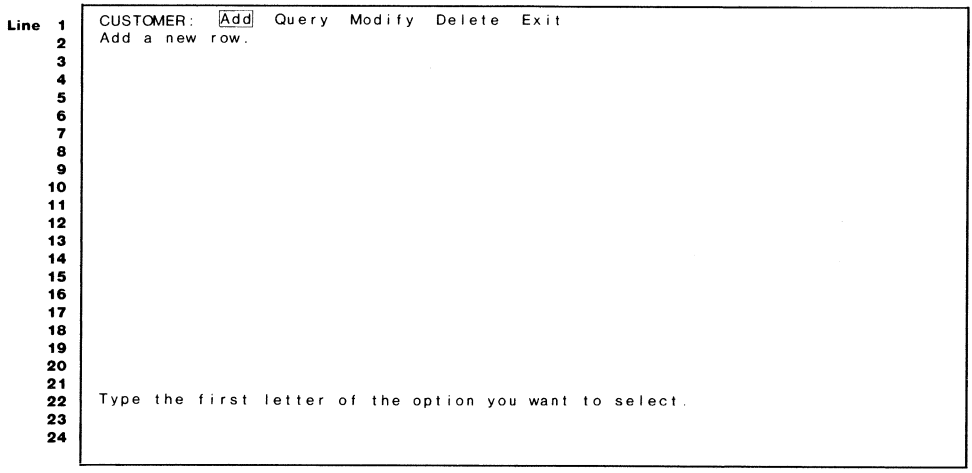


Figure 8-3. An Optional Message Line

Any messages that occur after the execution of this OPTIONS MESSAGE LINE statement will appear on line 22 until another OPTIONS MESSAGE LINE statement is executed.

Note: Usually, it is not a good idea to change the message line to line 23 (the default comment line) or line 24 (the default error line) since the INPUT statement clears the comment and error lines automatically.

The PROMPT LINE Clause

By default, INFORMIX-4GL displays prompts on line 1 of the screen. Unless you change the prompt-line default, part of your menu will be temporarily erased while the prompt is on the screen.

Use the PROMPT LINE clause to change the line on the screen where the prompt issued by a PROMPT statement will appear. The format is

```
OPTIONS PROMPT LINE integer
```

where *integer* is a line on the screen.

For example, the following OPTIONS statement changes the prompt line to 21:

```
OPTIONS PROMPT LINE 21
```

Any prompts that occur after this OPTIONS statement will appear on line 21 until another OPTIONS PROMPT LINE statement is executed.

Note: Usually, it is not a good idea to change the prompt line to line 23 (the default comment line) or to line 24 (the default error line).

Creating Help Messages

You can set up your INFORMIX-4GL program so that the user can request help for a particular display field, prompt, or menu option. This next section explains how to provide online help messages for menu options. You will learn how to

- Create and compile a help file.
- Use the OPTIONS statement to indicate the location of the help file.
- Designate an alternate help key with the OPTIONS statement.
- Indicate in each COMMAND clause of a MENU statement the message to be displayed when the user requests help for the specified option.

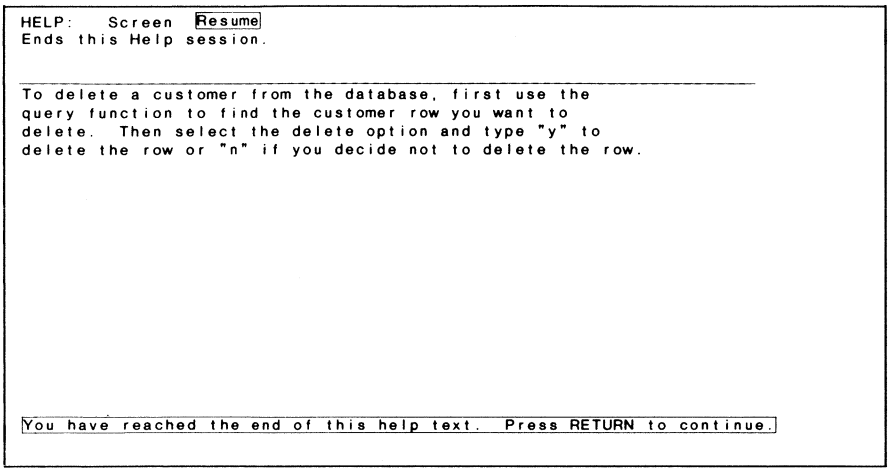
Requesting Online Help

This section shows how to write an INFORMIX-4GL program that displays a help message for each option on the CUSTOMER Menu:

```
CUSTOMER:  Add  Query  Modify  Delete  Exit
Add a new row.
```

The user can display a help message for a particular option by moving the highlight to the option and pressing the help key. INFORMIX-4GL then displays the help message for the current option and allows the user either to view the next screen of help information (if any) or return to the menu.

For example, if the user presses the help key while the cursor highlights the **Delete** option, your program might display the message shown below:



To return to the menu, the user presses **RETURN** (or types r to select the **Resume** option).

Creating a Help File

A help file is a text file containing numbered messages that you can use with MENU, INPUT, and PROMPT statements. The following help file, for example, contains help messages for the menu options on the CUSTOMER Menu:

```
.1
To add a row to the database, type the information
about the customer into the spaces between the brackets,
which are called "fields."  The label to the
left of a field tells what type of information
the field should contain.

After you have filled in one field, press RETURN to move the
cursor to the next field.  When you have filled the last
field, press ESCAPE or RETURN to add this customer to the
database.  If you want to add the customer without filling
in all fields, press ESCAPE.

When the program prompts you, type "y" to add another
customer, or type "n" to return to the CUSTOMER Menu.
.2
To search the database for information about a parti-
cular customer, type the customer's last name.
The program will find all the customers with the
given last name and display the first customer on the screen.

When the menu prompts you, enter RETURN if you want to
see information about the next customer with the
given name.  Otherwise, type "y" to return to the menu.
At this point you can modify the customer information
currently on your screen or delete the customer from
the database, if you choose.
.3
To modify a customer row, first use the query function to
find the customer row you want to modify.  Once the
customer row is on the screen, select the modify option.
Then use the arrow keys to move the cursor to any field you
want to change and enter the new information.  When you have
finished making changes, press ESCAPE to return to the menu.
.4
To delete a customer from the database, first use the
query function to find the customer row you want to
delete.  Then select the delete option and type "y" to
delete the row or "n" if you decide not to delete the row.
.5
Select this option to leave the CUSTOMER Menu.
```

Figure 8-4. The **custhelp.msg** File

As you can see, this help file consists of a series of messages separated by lines that contain only a period and a message number. The help file must always have this format, whether you are using the messages to provide help for an INPUT, PROMPT, or MENU statement.

Note: If a message is longer than 20 lines, INFORMIX-4GL automatically breaks the message into "pages" of 20 lines each. If you want different page breaks, you can type CONTROL-L at the beginning of a line to start a new page in the same message. If a message consists of more than one page, INFORMIX-4GL displays the first page of information on screen and allows the user to view the next page of information or return to the menu.

To create a help file, follow these instructions:

1. Use a text editor to create a file that stores help messages. Although you may choose any name you like for the help file, you may want to give the file an extension like **.msg** so that you can identify it easily.

2. Type the text for the help file using the format shown in Figure 8-4. Make sure you type a period followed by a message number on a separate line and then type the message below it. You do not have to enter messages in any specific order, but you must make sure that each message has a unique number.

3. Save the file when you are finished entering the help messages.

4. Run the **mkmessage** utility (described in the following section).

The **mkmessage** *Utility*

Unlike INFORMIX-4GL forms and programs, you cannot compile a help file in the INFORMIX-4GL Programmer's Environment. Instead, you must compile it using the **mkmessage** utility. You can do this by entering the following command after the system prompt

```
mkmessage filename1 filename2
```

where *filename1* is the name of the text file that contains the help messages, and *filename2* is the name of the compiled file that INFORMIX-4GL will use. Although you can choose any name you like for *filename2*, you may want to make it similar to *filename1* so that you can easily associate the compiled file with the text file.

For example, if you want to convert the text file in Figure 8-4 to a compiled file, you could enter

```
mkmessage custhelp.msg custhelp.ex
```

When you finish compiling a help file, you can call the messages in it from INFORMIX-4GL programs by using the statements discussed in the next section.

Using the Help File with a Program

To have your INFORMIX-4GL program display help messages, you must provide information about the help file in your program. First, you must use the OPTIONS statement to specify the pathname of the help file so that your program can find it. Then, you must supply the number of the help message at the point in the program where you want the message to be displayed.

More About the OPTIONS Statement

The OPTIONS statement provides two optional clauses for setting up help within a program: HELP FILE and HELP KEY.

The HELP FILE Clause. Use the HELP FILE clause to specify the pathname of the help file. Its format is

OPTIONS HELP FILE "*help-file*"

where *help-file* is the name of the compiled help file. If the help file is not in the current directory, make sure you designate the full pathname of the compiled help file after the HELP FILE keywords.

The HELP KEY Clause. The HELP KEY clause defines the key that the user must press to display a help message. The format of this statement is

OPTIONS HELP KEY *help-key*

where *help-key* is a function or CONTROL key.

To designate a function key as the help key, type Fn, where *n* is the number of the function key. CONTROL keys are designated with the format

CONTROL-*X*

where *X* is a letter key to be pressed along with the CONTROL key. (The default help key for **INFORMIX-4GL** is **CONTROL-W.**) You may not use the following keys in the HELP KEY clause of the OPTIONS statement:

Keys That Cannot Be Used in the Key-List

INTERRUPT
ESCAPE*
CONTROL-A
CONTROL-D
CONTROL-H
CONTROL-L
CONTROL-R
CONTROL-X

* unless you have specified another value as the ACCEPT KEY in the OPTIONS statement

Note: In addition, you may not be able to specify other keys in the HELP KEY clause, depending on your operating system.

For example, if you want to use the help file **custhelp.ex** and want F1 to be the help key, you could add the following OPTIONS statement to your program:

```
OPTIONS HELP FILE "custhelp.ex",
      HELP KEY F1
```

Specifying a Help Message for a Menu Option

You can associate a help message with a menu option by including its number in the COMMAND clause for the menu option. Use the following format

COMMAND "*menu-option*" ["*help-line*"] HELP *help-id*

where *help-id* is the message number for the help message that pertains to the *menu-option.*

For example, you can link the messages in the **custhelp.ex** file to the menu options in the CUSTOMER Menu with the following statements:

```
OPTIONS HELP FILE "custhelp.ex",
        HELP KEY F1

    . . .

MENU "CUSTOMER"

    COMMAND "Add" "Add a new row." HELP 1
        statements

    COMMAND "Query" "Search for a row." HELP 2
        statements

    COMMAND "Modify" "Modify a row." HELP 3
        statements

    COMMAND "Delete" "Delete a row." HELP 4
        statements

    COMMAND "Exit" "Leave the menu." HELP 5
        statements

END MENU
```

When, for example, the user presses the help key while the **Modify** option is highlighted, INFORMIX-4GL looks for help message 3 in the **custhelp.ex** help file and displays it on the screen.

A Sample Menu Program

This section describes the **custmenu** program that generates the menu and uses the screen form shown in Figure 8-6. It is available in the demonstration database.

```
CUSTOMER:  Add  Query  Modify  Delete  Exit
Add a new row.
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                          CUSTOMER FORM

       Number: [           ]

   First Name: [                ]     Last Name: [                ]

      Company: [                     ]

      Address: [                     ]
               [                     ]

         City: [                ]

        State: [  ]   Zipcode: [      ]

    Telephone: [                   ]
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

Type the first letter of the option you want to select or F1 for help.
```

Figure 8-5. The CUSTOMER Menu and **customer** Form

With this program, the user can add, retrieve, modify, and delete rows using the **customer** form introduced earlier in Chapter 6.

As you study this example, notice how the menu structure enables the various functions to share the form, messages, and prompts, as well as the contents of program variables. Explanatory notes follow the program.

```
DATABASE stores

GLOBALS

        DEFINE p_customer RECORD LIKE customer.*,
               chosen SMALLINT

END GLOBALS


MAIN

        OPEN FORM cust_form FROM "customer"

        DISPLAY FORM cust_form

        LET chosen = FALSE

        OPTIONS MESSAGE LINE 22,
                PROMPT LINE 21,
                HELP FILE "custhelp.ex",
                HELP KEY F1

        CALL show_menu()

        MESSAGE "End program."

        SLEEP 3

        CLEAR SCREEN

END MAIN
```

```
FUNCTION show_menu()

        DEFINE answer CHAR(1)

        MESSAGE "Type the first letter of the option ",
                "you want to select or F1 for Help."

        MENU "CUSTOMER"

                COMMAND "Add" "Add a new customer." HELP 1

                        LET answer = "y"

                        WHILE answer = "y"

                                CALL enter_row()

                                PROMPT "Do you want to ",
                                       "enter another row (y/n)? "
                                       FOR answer

                        END WHILE

                COMMAND "Query" "Search for a customer."  HELP 2

                        CALL query_data()

                        IF chosen THEN

                                NEXT OPTION "Modify"

                        END IF

                COMMAND "Modify" "Modify a customer."  HELP 3

                        IF chosen THEN

                                CALL change_data()
                        ELSE

                                MESSAGE "No customer has been chosen. ",
                                        "Use the Query option to select ",
                                        "a customer."

                                NEXT OPTION "Query"

                        END IF

                COMMAND "Delete" "Delete a customer."  HELP 4

                        IF chosen THEN

                                PROMPT   "Are you sure you want to ",
                                         "delete this customer (y/n)? "
                                         FOR answer

                                IF answer = "y" THEN

                                        CALL delete_row()

                                        LET chosen = FALSE

                                END IF

                        ELSE

                                MESSAGE "No customer has been chosen. ",
                                        "Use the Query option to select ",
                                        "a customer."

                                NEXT OPTION "Query"

                        END IF
```

```
                COMMAND "Exit" "Leave the CUSTOMER menu." HELP 5
                    EXIT MENU
        END MENU
END FUNCTION

FUNCTION enter_row()
        MESSAGE ""
        CLEAR FORM
        INPUT p_customer.fname THRU p_customer.phone
                FROM sc_cust.*
        LET p_customer.customer_num = 0
        INSERT INTO customer VALUES (p_customer.*)
        LET p_customer.customer_num = SQLCA.SQLERRD[2]
        DISPLAY p_customer.customer_num TO customer_num
        MESSAGE "Row added."
        SLEEP 3
        MESSAGE ""
END FUNCTION
```

```
FUNCTION query_data()

        DEFINE last_name CHAR(15),
                answer CHAR(1),
                exist SMALLINT

        MESSAGE ""

        CLEAR FORM

        PROMPT "Enter a last name: " FOR last_name

        MESSAGE "Selecting rows for customers with last name ",
                last_name, ". . ."

        SLEEP 3

        MESSAGE ""

        DECLARE a_curs CURSOR FOR
                SELECT * FROM customer WHERE lname MATCHES last_name

        LET exist = FALSE

        LET chosen = FALSE

        FOREACH a_curs INTO p_customer.*

                LET exist = TRUE

                DISPLAY BY NAME p_customer.*

                PROMPT "Enter 'y' to select this customer ",
                        "or RETURN to view next customer: "
                        FOR answer

                IF answer = "y" THEN

                        LET chosen = TRUE

                        EXIT FOREACH
                END IF

        END FOREACH

        IF exist = FALSE THEN

                MESSAGE "No customer rows found."

                SLEEP 3

                MESSAGE ""

        ELSE
                IF chosen = FALSE THEN

                        MESSAGE "There are no more customer rows."

                        SLEEP 3

                        MESSAGE ""

                        CLEAR FORM
                END IF

        END IF

END FUNCTION
```

```
FUNCTION change_data()

        INPUT p_customer.fname THRU p_customer.phone
                WITHOUT DEFAULTS FROM sc_cust.*

        UPDATE customer
                SET customer.* = p_customer.*
                WHERE customer_num = p_customer.customer_num

        MESSAGE "Row updated."

        SLEEP 3

        MESSAGE ""

END FUNCTION



FUNCTION delete_row()

        DELETE FROM customer WHERE customer_num =
                p_customer.customer_num

        CLEAR FORM

        MESSAGE "Row deleted."

        SLEEP 3

        MESSAGE ""

END FUNCTION
```

The GLOBALS Statement

The GLOBALS section of the program defines two global variables:

- **p_customer**, a record that contains variables corresponding to the columns in the **customer** table
- **chosen**, a flag indicating whether the user has selected a customer

The MAIN Statement

The MAIN section of the program performs the following operations:

1. Opens and displays the **customer** form.
2. Sets **chosen** to FALSE, indicating that no customer has been selected as yet.
3. Sets up line 22 as the message line, line 21 as the prompt line, specifies the pathname of the help file, and designates F1 as the help key.
4. Calls the **show_menu** function, which contains the MENU statement for the CUSTOMER menu.
5. Once **show_menu** is finished executing, INFORMIX-4GL displays a message indicating that the program is over and clears the screen.

The **show_menu** *Function*

The **show_menu** function sets up the CUSTOMER menu and calls the functions that carry out actions described by the menu options. This function performs the following operations:

1. Displays a message telling the user how to choose an option or display a help message.

2. Displays the menu name, menu options, and the help line for the highlighted option, as specified in the MENU statement.

3. Executes the Add COMMAND clause when the user chooses **Add** from the menu. If the user presses **F1** while the **Add** option is highlighted, the function displays help message 1 in the **custhelp.ex** help file on the screen.

   The Add COMMAND clause performs the following operations:

   a. Assigns the value y to the variable **answer**.

   b. Executes the statements in the WHILE loop since the initial value of **answer** is y.

   c. Calls the **enter_row** function that allows the user to enter a row on the **customer** form and then inserts the row into the **customer** table.

   d. Prompts the user to supply another value for **answer**.

   e. If the user enters a value other than y, **INFORMIX-4GL** leaves the WHILE loop and redisplays the menu so that the user can select another option.

   f. If the user enters y, **INFORMIX-4GL** executes the statements in the WHILE loop again.

4. Executes the Query COMMAND clause when the user chooses the **Query** option. If the user presses F1 while the highlight is on the **Query** option, INFORMIX-4GL displays message 2 in the help file.

   The Query COMMAND clause performs the following operations:

   a. Calls the **query_data** function, which retrieves customer rows based on a last name that the user supplies. The value of the global variable **chosen** is changed to TRUE if the query is successful.

   b. Highlights the **Modify** option if a customer has been chosen.

5. Executes the Modify COMMAND clause when the user chooses the **Modify** option. If the user presses F1 while the highlight is on the **Modify** option, INFORMIX-4GL displays message 3 in the help file.

   The Modify COMMAND clause performs the following operations:

   a. If the value of **chosen** is TRUE (indicating that the user has selected a customer), INFORMIX-4GL calls the **change_data** function, which allows the user to update the current row.

   b. Otherwise, INFORMIX-4GL displays a message indicating that no customer row has been chosen and highlights the **Query** option.

6. Executes the Delete COMMAND clause when the user selects **Delete** from the menu. If the user presses F1 while the highlight is on the **Delete** option, INFORMIX-4GL displays message 4 in the help file.

The Delete COMMAND clause performs the following operations:

a. If the value of **chosen** is TRUE (indicating that the user has selected a customer), INFORMIX-4GL asks the user if the current row should be deleted. If the user enters y, INFORMIX-4GL calls the **delete_row** function and then resets **chosen** to FALSE.

b. Otherwise, INFORMIX-4GL displays a message indicating that no customer row has been chosen and highlights the **Query** option.

7. Executes the Exit COMMAND clause when the user selects the **Exit** option. If the user presses F1 while the highlight is on the **Exit** option, INFORMIX-4GL displays message 5 in the help file.

   The Exit COMMAND clause terminates the MENU statement.

The **enter_row** *Function*

The **enter_row** function is similar to the sample programs for entering data shown in Chapter 7. This function performs the following operations:

1. Clears the message line and form.

2. Assigns values to all variables in the **p_customer** record except **p_customer.customer_num** from the data entered by the user on the screen form.

3. Assigns the value 0 to **p_customer.customer_num** as a placeholder for the SERIAL value that INFORMIX-4GL will insert automatically.

4. Inserts the data in **p_customer** into the **customer** table.

5. Assigns the value in SQLCA.SQLERRD[2] to **p_customer.customer_num**. (SQLCA.SQLERRD[2] stores the serial value of the row that INFORMIX-4GL just inserted into the **customer** table.)

6. Displays the serial number for the row just added to the **customer** table, along with the message Row added.

The query_data *Function*

The **query_data** function selects customer rows based on a last name that the user supplies. The function performs the following operations:

1. Clears the message line and screen form.

2. Prompts the user for the last name of a customer and assigns this value to the **last_name** variable.

3. Declares a cursor for that SELECT statement that retrieves all customers with the specified last name.

4. Initializes the local variable **exist** to FALSE, indicating that no rows have been found as yet.

5. Assigns the value FALSE to the global variable **chosen** to indicate that the user has not yet selected a customer.

6. Uses a FOREACH loop to display the customers on the screen form. The FOREACH loop performs the following operations:

   a. Selects a customer from the **customer** table and stores it in the record **p_customer**.

   b. Assigns the value TRUE to the local variable **exist** to indicate that at least one customer has been selected.

   c. Displays the values in **p_customer** on the screen form.

d. Asks whether the user wants to select the current customer or view the next customer. If the user enters y in response to the prompt, INFORMIX-4GL sets **chosen** to TRUE and exits the FOREACH loop.

e. Repeats steps a–d until all customer rows are processed, or until the user enters y to choose a customer and exit the FOREACH statement.

7. Displays a message if no customers were found (**exist**=FALSE).

8. Displays a message and clears the form if the user wanted to see another customer (**chosen**=FALSE) and INFORMIX-4GL could not find one.

The **change_data** *Function*

The **change_data** function enables the user to change a customer row. INFORMIX-4GL calls the **change_data** function only if the current value of the **chosen** variable is TRUE, indicating that the user has selected a customer row using the **query_data** function.

The **change_data** function is similar to the sample program for updating rows found in Chapter 7. This function performs the following operations:

1. Uses the INPUT statement without defaults to assign values to all variables in **p_customer** except **p_customer.customer_num** from data on the screen form.

2. Updates the row in the **customer** table where the value in the **customer_num** column equals the value currently stored in **p_customer.customer_num**.

3. Displays a message indicating that the row has been updated.

The **delete_row** *Function*

The **delete_row** function deletes the currently displayed row from the **customer** table. INFORMIX-4GL calls the **delete_row** function only if the current value of the **chosen** variable is TRUE, indicating that the user has selected a customer row using the **query_data** function.

The **delete_row** function is similar to the sample program for deleting a customer row found in Chapter 7. This function performs the following operations:

1. Deletes the row in the **customer** table where the value in the **customer_num** column equals the value currently stored in **p_customer.customer_num**.

2. Clears the form.

3. Informs the user that the row has been deleted.

Nested Menus

You can easily create a submenu by including a MENU statement in the COMMAND clause of another MENU statement. The MENU statement can appear in the COMMAND clause or can be embedded in a function that you call directly or indirectly from the COMMAND clause.

This section shows how you can modify the **Query** option of the CUSTOMER menu so that it displays a submenu if any customers exist with the last name specified by the user. This submenu includes **Next**, **Previous**, **First**, **Last**, **Choose**, and **Exit** options that allow the user to examine the customers in sequential or non-sequential order.

Following is an excerpt from the **c_menu2** program included with the demonstration database. This program is identical to the **custmenu** program except for a modification to the **query_data** function and the addition of a new function called **view_cust**. Explanatory notes follow the program.

```
...
FUNCTION show_menu()
        ...
        MENU "CUSTOMER"
                COMMAND "Add" "Add a new customer." HELP 1
                        ...
                COMMAND "Query" "Search for a customer." HELP 2
                        CALL query_data()
                        IF chosen THEN
                                NEXT OPTION "Modify"
                        END IF
                COMMAND "Modify" "Modify a customer." HELP 3
                        ...
                COMMAND "Delete" "Delete a customer." HELP 4
                        ...
                COMMAND "Exit" "Leave the CUSTOMER menu." HELP 5
                        ...
        END MENU
END FUNCTION
...
```

```
FUNCTION query_data()

        DEFINE last_name CHAR(15)

        MESSAGE ""

        CLEAR FORM

        PROMPT "Enter a last name: " FOR last_name

        MESSAGE "Selecting rows for customers with last name ",
                last_name, ". . ."

        SLEEP 3

        MESSAGE ""

        DECLARE a_curs SCROLL CURSOR FOR
                SELECT * FROM customer WHERE lname MATCHES last_name

        LET chosen = FALSE

        OPEN a_curs

        FETCH FIRST a_curs INTO p_customer.*

        IF status = NOTFOUND THEN
                MESSAGE "No customers found."
                SLEEP 3
                MESSAGE ""
        ELSE
                DISPLAY BY NAME p_customer.*
                CALL view_cust()
        END IF

        CLOSE a_curs

END FUNCTION
```

```
FUNCTION view_cust()

        MENU "BROWSE"

                COMMAND "Next" "View the next customer in the list."
                        FETCH NEXT a_curs INTO p_customer.*
                        IF status = NOTFOUND THEN
                                MESSAGE "No more customers in this direction."
                                SLEEP 3
                                MESSAGE ""
                                FETCH LAST a_curs INTO p_customer.*
                        END IF
                        DISPLAY BY NAME p_customer.*

                COMMAND "Previous" "View the previous customer in the list."
                        FETCH PREVIOUS a_curs INTO p_customer.*
                        IF status = NOTFOUND THEN
                                MESSAGE "No more customers in this direction."
                                SLEEP 3
                                MESSAGE ""
                                FETCH FIRST a_curs INTO p_customer.*
                        END IF
                        DISPLAY BY NAME p_customer.*

                COMMAND "First" "View the first customer in the list."
                        FETCH FIRST a_curs INTO p_customer.*
                        DISPLAY BY NAME p_customer.*

                COMMAND "Last" "View the last customer in the list."
                        FETCH LAST a_curs INTO p_customer.*
                        DISPLAY BY NAME p_customer.*

                COMMAND "Choose" "Choose the current customer."
                        LET chosen = TRUE
                        MESSAGE "This customer has been chosen."
                        SLEEP 3
                        MESSAGE ""

                COMMAND "Exit" "Leave the menu."
                        EXIT MENU

        END MENU

END FUNCTION

...
```

The **show__menu** *Function*

This function is identical to the **show__menu** function in the **custmenu** program. Like its counterpart, this function sets up the CUSTOMER menu and calls the functions that carry out actions described by the menu options.

The **query_data** *Function*

This **query_data** function differs from the **query_data** function in the **custmenu** program in the following respects:

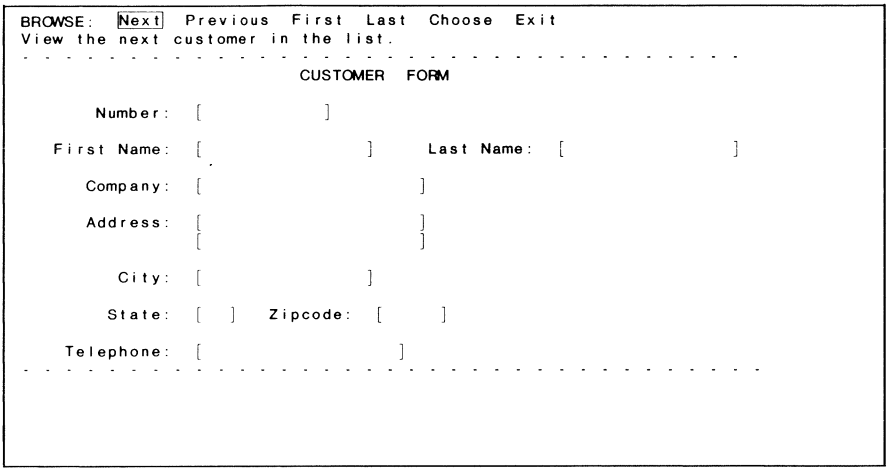
- It declares a scrolling cursor rather than a non-scrolling cursor for the SELECT statement that selects rows based on a last name that the user supplies.

- It uses OPEN, FETCH, and CLOSE instead of the FOREACH statement to retrieve and process the rows specified by the SELECT statement.

- The function attempts to FETCH only the first row in the active set. If the first row exists (**status** <> NOTFOUND), **query_data** displays the row and calls the **view_cust** function, which allows the user to view other rows in the active set by selecting options from a menu. If the first row does not exists (**status** = NOTFOUND), **query_data** displays the message No customers found and closes the cursor.

The **view_cust** *Function*

The **view_cust** function displays the following menu that lets the user browse through the rows in the active set.



The user can scroll forward and backward through the active set by selecting the **Next**, **Previous**, **First**, and/or **Last** options. The **Choose** option allows the user to select the current customer for a future update or delete operation, while the **Exit** option allows the user to leave the submenu.

The **view_cust** function includes a MENU statement that contains the following COMMAND clauses:

Next includes a FETCH NEXT statement that attempts to retrieve the next row of the active set. If the value of the **status** variable indicates that the cursor has moved beyond the last row of the active set, the IF statement returns the cursor to the last row and displays a message. The DISPLAY BY NAME statement displays the row retrieved by the appropriate FETCH statement on the screen.

Previous includes a FETCH PREVIOUS statement that attempts to retrieve the previous row of the active set. If the value of **status** indicates that the cursor has moved beyond the first row in the active set, the IF statement returns the cursor to the first row and displays a message. The DISPLAY BY NAME statement displays the row retrieved by the appropriate FETCH statement on the screen.

First includes a FETCH FIRST statement that retrieves the first row in the active set and displays it on screen. (There is no need to test the **status** variable after the FETCH FIRST statement, because the **view_cust** function is called only if the active set contains at least one row.)

Last includes a FETCH LAST statement that retrieves the last row in the active set and displays it on screen. (Again, there is no need to test the **status** variable after the FETCH LAST statement, because the **view_cust** function is called only if the active set contains at least one row.)

Choose assigns the value TRUE to the global variable **chosen** and displays a message indicating that the user has selected a customer for a future update or delete operation.

Exit leaves the BROWSE menu.

Chapter Summary

- You can create menus similar to those of the INFORMIX-4GL Programmer's Environment. A menu consists of a menu name, two or more menu options, and an optional help line that describes the current option.

- If the length of a menu exceeds the number of characters that the screen can display on a single line, INFORMIX-4GL displays the first "page" of options followed by an ellipsis (...) indicating that additional options exist.

- The user selects an option from a menu by typing the first letter of the option or by moving the highlight to the option and pressing RETURN. If you have created a help file for the menu, the user can display a help message for each menu option by pressing the help key.

- You can use the MENU statement to create a menu with two or more options. The MENU statement includes two or more COMMAND clauses, each of which specifies a menu option, an optional help line, an optional HELP clause, and one or more statements that perform the activity described by the menu option.

- You can use the CONTINUE MENU statement in a COMMAND clause if you sometimes want INFORMIX-4GL to ignore the remaining statements in the COMMAND clause and redisplay the menu so that the user can select another menu option.

- You can use the NEXT OPTION statement in a COMMAND clause if you want INFORMIX-4GL to highlight another option instead of the current option when it finishes executing the statements in a COMMAND clause.

- Use the EXIT MENU keywords at any point in the MENU statement where you want to leave the menu.

- You can change the lines where messages and prompts are displayed using the OPTIONS statement.

- You can use a text editor to create a help file containing numbered messages that you can use with MENU, INPUT, and PROMPT statements.

- You can compile a help file with the **mkmessage** utility.

- You can use a help file with a program by using the OPTIONS statement to specify the pathname of the help file and by supplying the appropriate message numbers in MENU, INPUT, or PROMPT statements.

- You can create a submenu by including a MENU statement in the COMMAND clause of another MENU statement. The MENU statement can appear in the COMMAND clause or can be embedded in a function that you call directly or indirectly from the COMMAND clause.

Chapter 9

Designing Reports

Chapter 9 Table of Contents

Chapter Overview 5

Designing Reports 6

How to Create a Report 7
- The REPORT Statement 8
  - The REPORT Keyword 8
  - The DEFINE Statement 9
  - The OUTPUT Section 9
  - The ORDER BY Statement 10
  - The FORMAT Section 11
  - The END REPORT Keywords 11
- Selecting Data to Include in the Report 11
- Running a Report 12

Basic Report Formatting 14
- FORMAT for a Default Report 14
- FORMAT for a Custom Report 15
- Page Headers and Trailers 16
  - Where Headers and Trailers Appear 16
  - Printing a Report Heading 17
  - The PRINT Statement 17
  - Format Keywords 18
  - The SKIP Statement 18
  - Printing Column Headings 19
  - Numbering Pages Automatically 19
- Printing Rows of Data 20
- Grouping Data 22
  - The Group Control Blocks 22

Calculations in a Report 27
- Arithmetic Operators 27
  - Example of Arithmetic Operations 28
- Controlling the Appearance of Numbers 29
  - Using Fill Characters 29
  - The Fill Characters 30
  - Adding a Dollar Sign 31
  - Making the Dollar Sign Float 31

Math Functions 32
Totaling a Column 33
Counting Rows in a Report 34
Group Functions 36

Displaying a Report 39
Printing the Report on a Printer 39
Writing the Report to a File 40
Piping the Report to Another Program 40

Running ACE Reports Using INFORMIX-4GL 41

Chapter Summary 42

Chapter Overview

This chapter explains how to create, display, and print reports with INFORMIX-4GL. In this chapter, you will learn

- How to write INFORMIX-4GL programs that produce reports
- How to select information from a database to include in a report
- How to group and format the information
- How to add titles, headings, and other text to customize a report
- How to perform calculations on report data
- How to run a report

The examples in this chapter are based on the following programs included with the demonstration database:

```
report1.4gl
report2.4gl
report3.4gl
report4.4gl
report5.4gl
report6.4gl
report7.4gl
```

For complete information about the statements for creating and running reports, see the *INFORMIX-4GL Reference Manual.*

Designing Reports

After you have stored information in a database, you can select all or some of the information and display it in a report. A report consists of information from a database, arranged and formatted according to your instructions, and displayed on the screen, printed (on paper), or stored in a file for future use.

You can combine information from a database to create whatever kind of report you need. Following are some of the reports you can produce with INFORMIX-4GL:

- Mailing labels
- Form letters
- Payroll summaries
- Medical histories
- Accounting records
- Inventory lists
- Telephone directories
- Project schedules
- Sales and marketing reports
- Payroll checks

Several sample reports are included in Appendix D. You may find them useful models for creating similar reports for your databases.

How to Create a Report

To create a report in INFORMIX-4GL, you need to complete the following tasks:

1. Include a REPORT statement outside the MAIN statement. You can use the REPORT statement to tell INFORMIX-4GL how you want to organize the information in a report, what headings or titles you want to add, what calculations you want to perform, and what margins, page size, and other formatting elements you want to use.

2. Select the information you want to include in the report either in the MAIN statement or in a function.

3. Use a loop (such as FOREACH or WHILE) in conjunction with the START REPORT, OUTPUT TO REPORT, and FINISH REPORT statements to send the data to the report.

Following is the basic loop structure for running an INFORMIX-4GL report:

```
MAIN
    START REPORT report1
    begin loop
        statement that assigns values to a, b, c
        OUTPUT TO REPORT report1 (a, b, c)
        . . .
    end loop
    FINISH REPORT report1
END MAIN

REPORT report1 (x, y, z)
. . .
END REPORT
```

Note: This example does not represent a real INFORMIX-4GL program.

You design a report by using a REPORT statement outside the MAIN statement. The REPORT statement has the following format:

```
REPORT report-name (argument-list)

    DEFINE-statement

    [OUTPUT output-statement ... ]

    [ORDER BY sort-list]

    FORMAT
        control block
            statement
            ...

        ...

END REPORT
```

The following sections describe how to use the REPORT statement.

The REPORT Keyword

Like a function, you must use the REPORT statement outside the MAIN statement. Follow the REPORT keyword with the name of the report. List the variables that contain the values you want to include in the report between the parentheses that follow the report name.

Make sure you do not use arrays or records that contain array elements in the argument list of the REPORT statement.

The DEFINE Statement

The variables in the *argument-list* must be defined in the REPORT statement. Following are two examples:

```
REPORT cust_report (fname, lname, company)

    DEFINE fname, lname CHAR(15),
           company LIKE customer.company
    ...
```

```
REPORT cust_list (r_customer)

    DEFINE r_customer RECORD LIKE customer.*
    ...
```

These variables receive values from the calling routine. This means that the data type of each variable should be compatible with or identical to the type of value it receives.

In addition, you may define local variables to use for counters, totals, and so on.

The OUTPUT Section

You should include an OUTPUT section in the REPORT statement if you want to change the default margins and page length for your report. In addition, you can use the OUTPUT section to send the report to a printer, write the report to a file, or pipe the report output to another program (see the section "Displaying a Report" at the end of this chapter for more information). If you include an OUTPUT section, you must place it immediately after the DEFINE statement in your REPORT statement.

Without an OUTPUT section, the report appears on the screen with the following default settings:

- LEFT MARGIN 5 spaces
- TOP MARGIN 3 lines
- BOTTOM MARGIN 3 lines
- PAGE LENGTH 66 lines (11 inches)

The ORDER BY Statement

Include an ORDER BY statement if the data the REPORT statement receives from the calling routine has not been ordered and you want the report to contain ordered data. The ORDER BY statement has the following format

ORDER BY *sort-list*

where *sort-list* is a list of one or more variables that you want to sort by.

Note: You can only sort on the variables that appear in the argument list of the REPORT statement.

For example, the following ORDER BY statement sorts customer rows by last name, and then by first name within each last-name group:

```
REPORT cust_list (fname, lname, company)

    DEFINE fname, lname CHAR(15),
           company LIKE customer.company

    ORDER BY lname, fname
    ...
```

The FORMAT Section

You can set up a format to suit any reporting need. With INFORMIX-4GL, you can create columnar reports, as well as reports with irregular formats, such as payroll checks or mailing labels. The FORMAT section of the REPORT statement controls the appearance of a report. In this section, you can write control blocks and statements to

- Create page headers and trailers
- Print information in any format you like
- Group information
- Perform calculations, such as totals and averages

The statements you use in the FORMAT section of a REPORT statement are described in the section "Basic Report Formatting" later in this chapter.

The END REPORT Keywords

Include the END REPORT keywords after the FORMAT section to conclude the REPORT statement.

Selecting Data to Include in the Report

Usually you will want to select data from a database to present in a report. This means that you will need to use a SELECT statement with a cursor and a program loop such as FOREACH to retrieve the rows and columns you want to include in the report. You can refer to Chapter 4 of this guide or to the *INFORMIX-4GL Reference Manual* for more information on SELECT and its associated statements.

Running a Report

To send data to the report from the main program or a function, you can use a loop such as WHILE or FOREACH with the following report statements:

START REPORT *report-name*

OUTPUT TO REPORT *report-name*(*variable-list*)

FINISH REPORT *report-name*

Use the START REPORT statement to specify the report you want to run. This statement should occur just before the loop that sends data to the report.

Note: Optionally, you may send a report to a printer, file, or pipe by including a TO clause in the START REPORT statement. See the *INFORMIX-4GL Reference Manual* for details.

With the OUTPUT TO REPORT statement, you pass the values of the variables in the *variable-list* to the named report. Make sure the variables in the OUTPUT TO REPORT statement correspond in number and in type to the variables in the *argument-list* of the REPORT statement. (If appropriate, you can use the THRU keyword or *record.** in the *variable-list* of the OUTPUT TO REPORT statement so that you do not have to list each variable explicitly.) The OUTPUT TO REPORT statement should occur within the program loop so that you can send data to the report one row at a time.

You must always include a FINISH REPORT statement after the loop that sends data to the named report.

If, for example, you want to use a FOREACH loop to send data to a report, you could write the following program. (This program is included with the demonstration database as **report1**.)

```
DATABASE stores

MAIN
     DEFINE p_customer RECORD LIKE customer.*

     DECLARE q_curs CURSOR FOR
          SELECT * FROM customer

     START REPORT cust_list

     FOREACH q_curs INTO p_customer.*

          OUTPUT TO REPORT cust_list (p_customer.*)

     END FOREACH

     FINISH REPORT cust_list

END MAIN


REPORT cust_list (r_customer)

     DEFINE r_customer RECORD LIKE customer.*

     FORMAT

          EVERY ROW

END REPORT
```

When INFORMIX-4GL encounters the FOREACH statement in this example, it performs the following operations:

- Retrieves a row from the **customer** table and stores it in the record **p_customer**.

- Passes the variables in the record **p_customer** to the **cust_list** report where they are assigned to the corresponding variables in the record **r_customer**.

- Formats the data according to your specifications.

  Note: The REPORT statement in this example contains an EVERY ROW statement that produces a report with a default format. INFORMIX-4GL also provides flexible formatting statements that allow you to design reports to your exact specifications. See the following section, "Basic Report Formatting," for more information about formatting data.

- Repeats the previous steps until all the rows in the **customer** table are processed.

Basic Report Formatting

The FORMAT section can include instructions for printing a simple default report, or it can contain instructions for printing a custom report that includes page headers, column headings, rows of data, and page trailers.

FORMAT for a Default Report

If you want a default report, you can use the EVERY ROW statement in the FORMAT section of your report. (Do not confuse the EVERY ROW statement with the ON EVERY ROW control block, which is described in the section "Printing Rows of Data," later in this chapter.) If you use an EVERY ROW statement, you cannot use any other statements in the FORMAT section. INFORMIX-4GL uses a default format for your report, displaying all the information that you select.

Following is a REPORT statement that uses EVERY ROW in the FORMAT section:

```
REPORT cust_list (fname, lname, company)

     DEFINE fname,
            lname          CHAR(15),
            company        CHAR(20)

     FORMAT EVERY ROW

END REPORT
```

This statement produces a report like the one that follows:

```
fname          lname          company

Ludwig         Pauli          All Sports Supplies
Carole         Sadler         Sports Spot
Philip         Currie         Phil's Sports
Anthony        Higgins        Play Ball!
Raymond        Vector         Los Altos Sports
George         Watson         Watson & Son
```

Note: The EVERY ROW statement only prints out the values in the argument list, not the values of global or local variables.

FORMAT for a Custom Report

The FORMAT section of a custom report may include up to seven types of control blocks. You can combine one or more of the control blocks in a variety of ways to design reports to your exact specifications. The order of control blocks in the FORMAT section of a REPORT statement is not significant. The seven control blocks you can use are

- FIRST PAGE HEADER
- PAGE HEADER
- PAGE TRAILER

- ON EVERY ROW
- ON LAST ROW
- BEFORE GROUP OF
- AFTER GROUP OF

The following sections describe these control blocks in more detail.

Page Headers and Trailers

You can print titles and column headings at the top of each page of a report. You can also print information at the bottom of each page. Use the following control blocks in your FORMAT section to control these report features:

- FIRST PAGE HEADER
- PAGE HEADER
- PAGE TRAILER

Where Headers and Trailers Appear

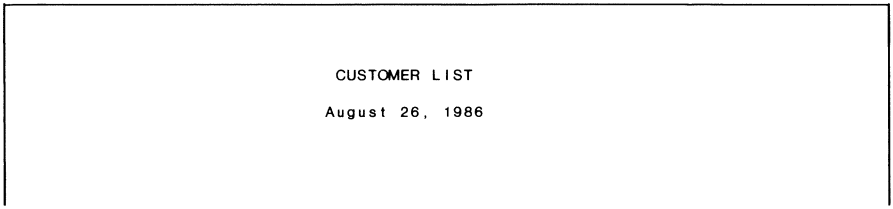
Text you specify in the PAGE HEADER control block appears on the line immediately under the top margin on every page of a report. Text you specify in the PAGE TRAILER control block appears at the bottom of each page, on the line immediately above the bottom margin.

If you want different text to appear on the first page of a report, use the FIRST PAGE HEADER control block to specify headings for the first page.

Printing a Report Heading

You can use the PAGE HEADER control block with format statements such as PRINT and SKIP to specify the text you want in the heading and where you want the header positioned.

For example, you may want to print a report title at the top of the page like this:



The following example shows the PAGE HEADER control block that produces this heading:

```
PAGE HEADER
     PRINT COLUMN 25, "CUSTOMER LIST"
     SKIP 1 LINE
     PRINT COLUMN 24, "August 26, 1986"
```

The PRINT Statement

The PRINT statement is very similar to the DISPLAY statement introduced earlier in Chapter 3. You can use the PRINT statement with variables and string constants to print information in a format you specify. A simple PRINT statement may contain a string constant enclosed in double quotes:

```
PRINT "CUSTOMER LIST"
```

A more complex PRINT statement might include both string constants and variables, separated by commas:

```
PRINT "Last Name:  ", lname
```

Format Keywords

You can use keywords such as CLIPPED, SPACE or SPACES, and COLUMN to control the spacing of the information you want to print. The effect of the CLIPPED keyword in a PRINT statement is the same as its effect in the DISPLAY statement. The SPACE, SPACES, and COLUMN keywords are described below.

The SPACE or SPACES Keyword. If you need to insert spaces between expressions in a PRINT statement, you can use the SPACE or SPACES keyword instead of a quoted string of spaces. For example, the following PRINT statements are equivalent, although the first contains quoted spaces and the second contains the SPACE and SPACES keywords:

```
PRINT city CLIPPED, ", ", state, "  ", zipcode
PRINT city CLIPPED, ",", 1 SPACE, state, 2 SPACES, zipcode
```

The COLUMN Keyword. You can use the COLUMN keyword to print text starting at a specific column position. For example, the following statement prints the heading CUSTOMER LIST starting at column 25:

```
PRINT COLUMN 25, "CUSTOMER LIST"
```

The SKIP Statement

You can use the SKIP statement to specify how many blank lines you want to leave. For example, you can leave two blank lines by entering the following statement:

```
SKIP 2 LINES
```

If you want to advance to the top of the next page, use SKIP TO TOP OF PAGE.

Printing Column Headings

You can also use the PAGE HEADER or FIRST PAGE HEADER control blocks to specify headings that you want to appear above the columns of a report. For example, to print the title and the column headings for a customer list, you can use a PAGE HEADER control block like the following:

```
PAGE HEADER
     PRINT COLUMN 25, "CUSTOMER LIST"
     SKIP 1 LINE
     PRINT COLUMN 24, "August 26, 1986"
     SKIP 2 LINES
     PRINT "Last Name",
           COLUMN 26, "First Name",
           COLUMN 50, "Company"
     SKIP 1 LINE
```

These statements produce the following result:

```
                    CUSTOMER LIST

                    August 26, 1986

Last Name           First Name          Company
```

Numbering Pages Automatically

You can include a page number in either the header or trailer by using a PAGENO expression in a PRINT statement. When you print the report, each page is numbered automatically.

For example, the following PAGE TRAILER control block produces text and a page number at the bottom of each page:

```
PAGE TRAILER
     PRINT "Confidential Information",
           COLUMN 50, PAGENO USING "Page ##"
```

(For more information about the USING keyword, see the section "Controlling the Appearance of Numbers" later in this chapter.) This control block produces the following page trailer:

```
Confidential Information                    Page 1
```

Printing Rows of Data

To print rows of information, you must use the ON EVERY ROW control block with statements such as PRINT and SKIP in the FORMAT section of your report.

You can print only the values of global variables or variables that are defined in the DEFINE section of the REPORT statement. For example, if **fname** is not a global variable or is not included in the DEFINE section, you cannot print it in the ON EVERY ROW control block.

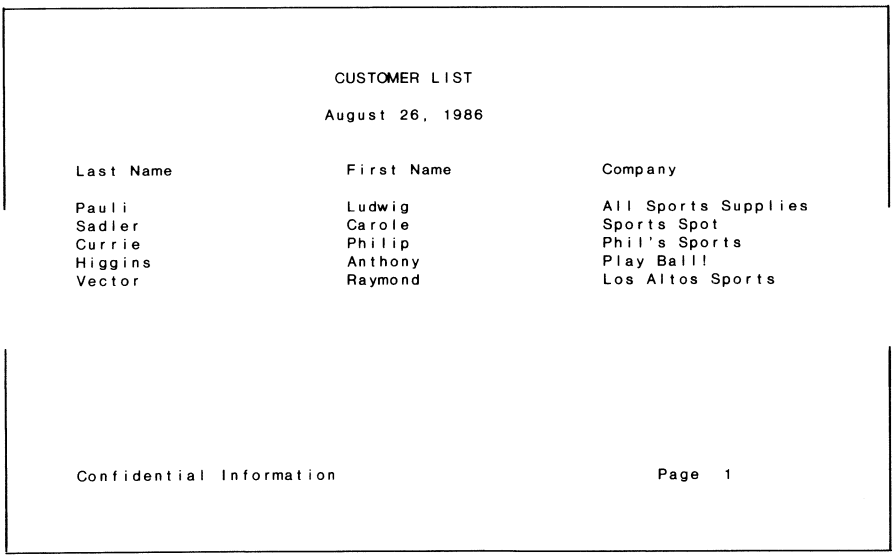
The following example from the **report2** program shows the FORMAT section of a REPORT statement that uses the ON EVERY ROW control block:

```
FORMAT
   PAGE HEADER
        PRINT COLUMN 25, "CUSTOMER LIST"
              SKIP 1 LINE
              PRINT COLUMN 24, "August 26, 1986"
              SKIP 2 LINES
              PRINT "Last Name",
                    COLUMN 26, "First Name",
                    COLUMN 50, "Company"
              SKIP 1 LINE

        ON EVERY ROW
              PRINT lname,
                    COLUMN 26, fname,
                    COLUMN 50, company

        PAGE TRAILER
              PRINT "Confidential Information",
                    COLUMN 55, PAGENO USING "Page ##"
```

This example produces a report that lists the last name, first name, and company for every customer:



```
                        CUSTOMER LIST

                       August 26, 1986


Last Name               First Name              Company

Pauli                   Ludwig                  All Sports Supplies
Sadler                  Carole                  Sports Spot
Currie                  Philip                  Phil's Sports
Higgins                 Anthony                 Play Ball!
Vector                  Raymond                 Los Altos Sports




Confidential Information                                  Page  1
```

You can divide the information in a report into groups. For example, you might want to group customers by city as follows:

CUSTOMER LIST

August 26, 1986

Last Name	Company	City	State
Vector	Los Altos Sports	Los Altos	CA
Lawson	Runners & Others	Los Altos	CA
Beatty	Sportstown	Menlo Park	CA
Grant	Gold Medal Sports	Menlo Park	CA
Watson	Watson & Son	Mountain View	CA
Parmelee	Olympic City	Mountain View	CA
Baxter	Blue Ribbon Sports	Oakland	CA
Currie	Phil's Sports	Palo Alto	CA
Ream	Athletic Supplies	Palo Alto	CA
Higgins	Play Ball!	Redwood City	CA
Quinn	Quinn's Sports	Redwood City	CA
Jaeger	AA Athletics	Redwood City	CA

The Group Control Blocks

To group the information in a report, use the following control blocks in your FORMAT section:

- BEFORE GROUP OF
- AFTER GROUP OF

These control blocks specify what **INFORMIX-4GL** does at the beginning or end of a group. After the BEFORE GROUP OF or AFTER GROUP OF keywords, enter the variable name that contains the information you want to group by, and the statements you want **INFORMIX-4GL** to execute.

Note: You can only use a variable that appears in the argument list of a REPORT statement after the BEFORE GROUP OF or AFTER GROUP OF keywords.

The BEFORE GROUP OF and AFTER GROUP OF control blocks are designed to work with ordered data. If you plan to sort your data *in* the REPORT statement, make sure that the same variable name appears in both the ORDER BY statement and the BEFORE GROUP OF or AFTER GROUP OF control block. In the following report, for example, the variable **city** appears in both the ORDER BY statement and the AFTER GROUP OF control block:

```
REPORT cust_list (lname, company, city, state)

     DEFINE lname      CHAR(15),
     company           CHAR(20),
     city              CHAR(15),
     state             CHAR(2)

     ORDER BY city

     FORMAT

          ON EVERY ROW
               PRINT lname,
               COLUMN 18, company,
               COLUMN 40, city,
               COLUMN 60, state

          AFTER GROUP OF city
               SKIP 2 LINES

END REPORT
```

If you plan to sort your data outside the REPORT statement, make sure the column name that appears in the ORDER BY clause of the SELECT statement corresponds to the variable name that appears in the BEFORE GROUP OF or AFTER GROUP OF control block of the REPORT statement.

An example appears in the **report3** program included with the demonstration database:

```
DATABASE stores

MAIN
      DEFINE p_customer RECORD LIKE customer.*

      DECLARE q_curs CURSOR FOR
            SELECT lname, company, city, state
            FROM customer
            ORDER BY city

      START REPORT cust_list

      FOREACH q_curs INTO p_customer.lname,
            p_customer.company,
            p_customer.city,
            p_customer.state

            OUTPUT TO REPORT cust_list (p_customer.lname,
                  p_customer.company,
                  p_customer.city,
                  p_customer.state)

      END FOREACH

      FINISH REPORT cust_list

END MAIN


REPORT cust_list (lname, company, city, state)

      DEFINE lname      CHAR(15),
            company     CHAR(20),
            city        CHAR(15),
            state       CHAR(2)

      FORMAT
            ON EVERY ROW
                  PRINT lname,
                  COLUMN 18, company,
                  COLUMN 40, city,
                  COLUMN 60, state

      AFTER GROUP OF city
            SKIP 2 LINES

END REPORT
```

Note: If you sort data outside a report that contains BEFORE GROUP OF or AFTER GROUP OF control blocks on two or more variables, include an ORDER EXTERNAL BY statement in the report that specifies the hierarchy of ordering.

For example, if you sorted your data by last name and then by city outside a report that contains the following AFTER GROUP OF control blocks,

```
AFTER GROUP OF lname
    ...
AFTER GROUP OF city
    ...
```

you should precede the FORMAT section of the report with the following statement:

```
ORDER EXTERNAL BY lname, city
```

The instructions you include in a BEFORE GROUP OF control block apply to actions **INFORMIX-4GL** performs *before* printing a group of rows. You can use BEFORE GROUP OF to print headings above a group of rows, for example.

The instructions you include in the AFTER GROUP OF control block apply to actions **INFORMIX-4GL** performs *after* printing a group of rows. You can use AFTER GROUP OF to calculate totals or subtotals under a group of rows.

The following example from the **report4** program shows a REPORT statement that groups customers by city. As you can see, the same variable name appears after the ORDER BY and AFTER GROUP OF keywords.

```
REPORT cust_list (lname, company, city, state)

     DEFINE lname      CHAR(15),
            company    CHAR(20),
            city       CHAR(15),
            state      CHAR(2)

     ORDER BY city

     FORMAT

          PAGE HEADER
               PRINT COLUMN 25, "CUSTOMER LIST"
               SKIP 1 LINE
               PRINT COLUMN 24, "August 26, 1986"
               SKIP 2 LINES
               PRINT "Last Name",
                     COLUMN 18, "Company",
                     COLUMN 40, "City",
                     COLUMN 60, "State"
               SKIP 2 LINES

          ON EVERY ROW
               PRINT lname,
                     COLUMN 18, company,
                     COLUMN 40, city,
                     COLUMN 60, state

          AFTER GROUP OF city
               SKIP 2 LINES

          PAGE TRAILER
               PRINT "Confidential Information",
                     COLUMN 55, PAGENO USING "Page ##"

END REPORT
```

This REPORT statement produces the report that appears earlier in this chapter in the section "Grouping Data."

Calculations in a Report

Often you want to perform calculations on the numeric data you include in a report. For example, you may want to total the values in a column, multiply the values in a column by a constant, or add the values in one column to the values in another.

INFORMIX-4GL provides arithmetic operators, built-in math functions (also called aggregates), and group functions you can include in a REPORT statement to compute values in reports.

Arithmetic Operators

You can perform addition, subtraction, multiplication, division, or exponentiation on any of the numeric columns in a report. Use any of the following arithmetic operators in the FORMAT section:

+	Addition
−	Subtraction
*	Multiplication
/	Division
**	Exponentiation
mod	Modulus

Example of Arithmetic Operations

If you want to calculate a 15 percent increase in price for every item in the **stock** table, you could enter the following ON EVERY ROW control block in the FORMAT section of your report. (This example is based on the **report5** program included with the demonstration database.)

```
ON EVERY ROW
     PRINT stock_num,
          COLUMN 15, manu_code,
          COLUMN 30, description,
          COLUMN 50, unit,
          COLUMN 57, unit_price * 1.15 USING "$$$$$.$$"
```

Note: The USING clause is described in the next section.

When you run the report, INFORMIX-4GL multiplies the current unit price for each item by 1.15 and prints the results. The values in the **unit_price** column of the **stock** table are not changed by this operation.

Here is what a portion of the report might look like:

```
1       HRO       baseball gloves     case       $287.50
1       HSK       baseball gloves     case       $920.00
1       SMT       baseball gloves     case       $517.50
2       HRO       baseball            case       $144.90
3       HSK       baseball bat        case       $276.00
4       HSK       football            case      $1104.00
```

You can format numbers to include dollar signs and commas, like this:

```
$5,970

$    8,520

$5,550.20
```

Note: If a program variable has the data type MONEY, INFORMIX-4GL automatically formats the money value with dollar signs.

To format numbers, include a USING clause in the PRINT statement that shows the format you want to use for the numbers. The USING clause consists of the USING keyword followed by a *format string* like the one shown below:

```
PRINT unit_price * 1.15 USING "####"
```

Using Fill Characters

The previous example uses pound signs (####) as *fill characters*. A collection of fill characters between quotation marks is a *format string*. You use a format string to set up a number format. When INFORMIX-4GL prints a report, it substitutes actual numbers in place of the format string.

In the example, the format string tells INFORMIX-4GL that every number should occupy four spaces. Thus, if the numbers in the report are 45, 456, and 4560, INFORMIX-4GL prints them as

```
  45
 456
4560
```

INFORMIX-4GL prints leading blanks so that every number contains four places. When you use the # fill character, INFORMIX-4GL adds as many leading blanks as necessary so that every number fits the format you set up.

If a number is larger than the pattern you set up in the PRINT USING statement, INFORMIX-4GL prints a group of asterisks instead of the number. If the numbers are 45, 456, 4560, and 45600, they appear in the report as

```
  45
 456
4560
****
```

The Fill Characters

You can use the following special fill characters with format strings:

#	Prints leading blanks if a number does not completely fill the format (as shown in the preceding examples).
&	Prints leading zeros if a number does not completely fill the format (for example, 0045 or 0456).
–	Prints leading minus signs if the number is negative. If you only use one minus sign and the number is negative, INFORMIX-4GL will place a minus sign in the position you indicate in the format string.
*	Prints leading asterisks (*45 or **456) if a number does not completely fill the format (useful for printing checks).
<	Prints numbers left-justified.

You can combine these and other symbols in a variety of ways to control and print numbers in the format you want. Refer to the *INFORMIX-4GL Reference Manual* for a complete explanation of all the formatting features available in INFORMIX-4GL.

Adding a Dollar Sign

If you want to include a dollar sign and a comma with the numbers, include those symbols in the PRINT USING statement, like this:

```
PRINT unit_price * 1.15 USING "$#,###"
```

Now INFORMIX-4GL prints the numbers 45, 456, and 4560 like this:

```
$    45
$   456
$4,560
```

Making the Dollar Sign Float

In the preceding example, the dollar sign always appears in the same position because the # fill character tells INFORMIX-4GL to print blanks if there are no numbers. It fixes the dollar sign in a specific position in the number.

If you want the dollar sign to float, or change position according to the length of the number, enter the PRINT USING statement like this:

```
PRINT unit_price * 1.15 USING "$$,$$$"
```

Now the dollar sign appears at the beginning of the numbers, regardless of their length, like this:

```
   $45
  $456
$4,560
```

Math Functions

You can perform calculations by using arithmetic operators or any of the following math functions (also called aggregates):

SUM(*expression*)	Adds the values in a report column
MIN(*expression*)	Prints the smallest (minimum) value in a report column
MAX(*expression*)	Prints the largest (maximum) value in a report column
AVG(*expression*)	Calculates the average of the values in a report column
PERCENT(*)	Calculates the percentage of all rows that are in a group (refer to the *INFORMIX-4GL Reference Manual* for more information)
COUNT(*)	Counts the number of rows in the report

Note: You can only use aggregate functions on variables that appear in the *argument-list* of the REPORT statement, not on local variables.

Totaling a Column

To add all the values in a report column, you can use the ON LAST ROW control block with a PRINT statement that contains the SUM keyword followed by the name of the variable you want to total.

For example, the following report contains an ON LAST ROW control block that adds all the numbers in the **quantity** column and prints the total:

```
REPORT qty_rep1 (stock_num, manu_code, quantity, description, unit)

    DEFINE stock_num      LIKE items.stock_num,
           manu_code      LIKE items.manu_code,
           quantity       LIKE items.quantity,
           description    LIKE stock.description,
           unit           LIKE stock.unit

    FORMAT
        PAGE HEADER
            PRINT "Stock Number",
                  COLUMN 15, "Manufacturer",
                  COLUMN 30, "Description",
                  COLUMN 50, "Unit",
                  COLUMN 60, "Quantity"
                  SKIP 2 LINES

        ON EVERY ROW
            PRINT stock_num,
                  COLUMN 15, manu_code,
                  COLUMN 30, description,
                  COLUMN 50, unit,
                  COLUMN 60, quantity

        ON LAST ROW
            SKIP 1 LINE
            PRINT COLUMN 20, "Total Quantity on Order: ",
                  COLUMN 55, SUM(quantity) USING "###"
END REPORT
```

If you select items with the stock number 5, this statement produces a report like the following:

```
Stock Number  Manufacturer  Description      Unit       Quantity

           5  NRG           tennis racquet   each             10
           5  NRG           tennis racquet   each              5
           5  SMT           tennis racquet   each              5
           5  ANZ           tennis racquet   each              5
           5  ANZ           tennis racquet   each             10
           5  ANZ           tennis racquet   each              5
           5  ANZ           tennis racquet   each              5
           5  ANZ           tennis racquet   each              1

                    Total Quantity on Order:              46
```

Counting Rows in a Report

You can use the ON LAST ROW control block with a PRINT statement that contains the COUNT aggregate to tally the number of rows in a report.

For example, the following report contains an ON LAST ROW control block that prints the number of orders as well as the total quantity on order. (This example is based on the **report6** program included with the demonstration database.)

```
REPORT qty_rep2 (stock_num, manu_code, quantity, description, unit)

        DEFINE stock_num      LIKE items.stock_num,
               manu_code      LIKE items.manu_code,
               quantity       LIKE items.quantity,
               description    LIKE stock.description,
               unit           LIKE stock.unit

        FORMAT
                PAGE HEADER
                PRINT "Stock Number",
                      COLUMN 15, "Manufacturer",
                      COLUMN 30, "Description",
                      COLUMN 50, "Unit",
                      COLUMN 60, "Quantity"
                      SKIP 2 LINES

        ON EVERY ROW
                PRINT stock_num,
                      COLUMN 15, manu_code,
                      COLUMN 30, description,
                      COLUMN 50, unit,
                      COLUMN 60, quantity

        ON LAST ROW
                SKIP 1 LINE
                PRINT COLUMN 20, "Total Quantity on Order: ",
                      COLUMN 55, SUM(quantity) USING "###"
                SKIP 1 LINE
                PRINT COLUMN 20, "Total Number of Orders: ",
                      COLUMN 55, COUNT(*) USING "###"

END REPORT
```

Here is what the report looks like:

```
Stock Number  Manufacturer  Description      Unit      Quantity

           5  NRG           tennis racquet   each            10
           5  NRG           tennis racquet   each             5
           5  SMT           tennis racquet   each             5
           5  ANZ           tennis racquet   each             5
           5  ANZ           tennis racquet   each            10
           5  ANZ           tennis racquet   each             5
           5  ANZ           tennis racquet   each             5
           5  ANZ           tennis racquet   each             1

                    Total Quantity on Order:            46

                    Total Number of Orders:              8
```

Group Functions

In addition to arithmetic and math functions, **INFORMIX-4GL** contains functions you can use to perform calculations on groups of information in a report. The group functions allow you to calculate subtotals or interim results.

The group functions you can use in your REPORT statement include

- GROUP SUM(*expression*)
- GROUP COUNT(*)
- GROUP MIN(*expression*)
- GROUP MAX(*expression*)
- GROUP PERCENT(*)
- GROUP AVG(*expression*)

To perform calculations on groups of rows, use an AFTER GROUP OF control block with a PRINT statement that contains a group function followed by a variable name.

Note: You can only use a variable that appears in the *argument-list* of the REPORT statement with group functions.

Any time you use a group function with a variable, the data must be sorted on that variable. You may ORDER BY that variable inside the REPORT statement or ORDER BY the corresponding database column in a SELECT statement outside the REPORT statement.

For example, the following REPORT statement sorts items by stock number and prints the total quantity on order for each item. (This example is based on the **report7** program included with the demonstration database.)

```
REPORT qty_rep3 (stock_num, manu_code, quantity, description, unit)

        DEFINE stock_num        LIKE items.stock_num,
               manu_code        LIKE items.manu_code,
               quantity         LIKE items.quantity,
               description      LIKE stock.description,
               unit             LIKE stock.unit

         ORDER BY stock_num

         FORMAT
               PAGE HEADER
                    PRINT "Stock Number",
                          COLUMN 15, "Manufacturer",
                          COLUMN 30, "Description",
                          COLUMN 50, "Unit",
                          COLUMN 60, "Quantity"
                    SKIP 2 LINES

               ON EVERY ROW
                    PRINT stock_num,
                          COLUMN 15, manu_code,
                          COLUMN 30, description,
                          COLUMN 50, unit,
                          COLUMN 60, quantity

               AFTER GROUP OF stock_num
                    SKIP 1 LINE
                    PRINT COLUMN 20, "Total Quantity on Order: ",
                          COLUMN 55, GROUP SUM(quantity) USING "###"
                    SKIP 2 LINES

END REPORT
```

This statement produces a report like the following:

Stock Number	Manufacturer	Description	Unit	Quantity
1	HRO	baseball gloves	case	1
1	HRO	baseball gloves	case	1
1	HRO	baseball gloves	case	1
1	HSK	baseball gloves	case	1
1	SMT	baseball gloves	case	1
1	SMT	baseball gloves	case	1
	Total Quantity on Order:			6
2	HRO	baseball	case	1
2	HRO	baseball	case	1
	Total Quantity on Order:			2
3	HSK	baseball bat	case	1
3	HSK	baseball bat	case	1
3	HSK	baseball bat	case	1
	Total Quantity on Order:			3
4	HSK	football	case	1
4	HSK	football	case	1
4	HRO	football	case	1
4	HRO	football	case	1
	Total Quantity on Order:			4

Displaying a Report

There are four ways you can handle the output of a report. You can run the report and have the results

- Displayed on your screen
- Printed on a printer
- Written to a file
- Piped as input to another program (UNIX systems only)

You specify how you want the report routed in the START REPORT statement (see the *INFORMIX-4GL Reference Manual* for details) or in the OUTPUT section of the REPORT statement. You can use the OUTPUT section to change the margins and page length as well.

If you include an OUTPUT section in the REPORT statement, it must appear immediately after the DEFINE section.

Printing the Report on a Printer

To print the report on a printer, use the REPORT TO PRINTER statement in the OUTPUT section. The following example shows an OUTPUT section that prints the report on a printer. In addition, it changes the left margin to 10 and the page length to 50.

```
OUTPUT
      LEFT MARGIN 10
      PAGE LENGTH 50
      REPORT TO PRINTER
```

Writing the Report to a File

To send the report results to a file, use the REPORT TO keywords followed by the name of the file in quotation marks.

If you want to write the report to a file named **cust_rep**, for example, you would include an OUTPUT section like the following:

```
OUTPUT
      REPORT TO "cust_rep"
```

When you run the report, INFORMIX-4GL creates a file named **cust_rep** and writes the report results in the file.

Piping the Report to Another Program

Refer to the *INFORMIX-4GL Reference Manual* for a complete explanation of how to pipe report results to another program.

Running ACE Reports Using INFORMIX-4GL

If you have **INFORMIX-SQL** as well as **INFORMIX-4GL**, you can run **ACE** reports created with **INFORMIX-SQL** from your **INFORMIX-4GL** programs using the RUN statement. The format is

```
RUN "isqlu -rr report-name"
```

where *report-name* is the name of a compiled **ACE** report. Make sure that you have the **INFORMIX-SQL** program either in your current directory or in your path.

Note: You can also use the RUN statement to execute other valid operating-system commands. See the *INFORMIX-4GL Reference Manual* for details.

Chapter Summary

- With INFORMIX-4GL, you can select information from your database and display it in a report.
- You use a REPORT statement outside the main program to specify the format you want for the report.
- You select the information for the report by using a SELECT statement either in the main program or in a function.
- To send data to the report, use a program loop such as FOREACH with the statements START REPORT, OUTPUT TO REPORT, and FINISH REPORT.
- To produce a default report, use the EVERY ROW statement in the FORMAT section of your report. If you use an EVERY ROW statement, you cannot use any other statements in the FORMAT section.
- To produce a custom report, you can include up to seven types of control blocks in the FORMAT section of a REPORT statement: FIRST PAGE HEADER, PAGE HEADER, PAGE TRAILER, ON EVERY ROW, ON LAST ROW, BEFORE GROUP OF, and AFTER GROUP OF.
- Use the FIRST PAGE HEADER control block to print information on the first page of your report.
- You can print titles and column headings at the top of each page by using the PAGE HEADER control block.
- Include a PAGE TRAILER control block if you want to print a footer at the bottom of every page.
- Use the ON EVERY ROW control block to print rows of information in the report.
- You can divide rows into groups by using the BEFORE GROUP OF or AFTER GROUP OF control blocks.

- You can perform calculations on all rows by using the ON LAST ROW control block with PRINT statements that contain aggregate functions and/or arithmetic operators.

- You can perform calculations on groups of rows by using the AFTER GROUP OF control block with PRINT statements that contain group aggregate functions and/or arithmetic operators.

- Include an OUTPUT section in the REPORT statement if you want to change the page layout or send the report to a printer, file, or pipe.

Chapter 10

Handling Errors and Interrupts from the User

Chapter 10 Table of Contents

Chapter Overview 5

Anticipating Errors 6
- Determining Whether an Error Occurred: the **status** Variable 6
- Trapping Errors: the WHENEVER Statement 7
  - The *startlog* Function 10
  - The *errorlog* Function 11
- A Sample Program for Handling Errors 12
- Handling Inappropriate Input 13

Handling Interrupts from the User 16
- The INTERRUPT and QUIT Keys 16
- The DEFER Statement 16
  - The DEFER INTERRUPT Statement 17
  - Example 1 18
  - Example 2 21
- An Alternative to the INTERRUPT Key 23
  - More About INPUT: The ON KEY Clause 23
  - Using the ON KEY Clause in the PROMPT Statement 24
  - Example 1 25
  - Example 2 28

Chapter Summary 30

Chapter Overview

This chapter explains how to write programs that react to runtime errors and interrupts from the user. The chapter describes

- How INFORMIX-4GL uses the **status** variable to indicate that an error has occurred
- How to use the WHENEVER ERROR statement to trap errors while a program is running
- How to use the **startlog** function to create an error-log file
- How to use the **errorlog** function to write text in the error-log file
- How to use DEFER INTERRUPT to keep your program from stopping when the user presses the INTERRUPT key
- How to use the ON KEY clause to allow the user to cancel a PROMPT or INPUT statement without using the INTERRUPT key

The examples in this chapter are based on the following programs and forms included with the demonstration database:

ch10def.4gl
ch10def2.4gl
ch10key.4gl
ch10key2.4gl
ch10notf.4gl
ch10when.4gl
customer.per

For complete information about the statements used in this chapter, refer to the *INFORMIX-4GL Reference Manual.*

Anticipating Errors

As you develop a program, you will encounter two kinds of errors: compile-time errors and runtime errors.

Compile-time errors occur if you have not used the correct syntax for one or more statements in your program. You handle compile-time errors by correcting the syntax of the marked statements and recompiling your program until you receive a message indicating that compilation has been successful.

Runtime errors occur while your program is running and result from a variety of conditions. For example, INFORMIX-4GL may not be able to find a specified file, may run out of memory, or may try to insert a value that is too large for a SMALLINT column. As you develop your program, think about the kinds of runtime errors that can occur and how you plan to deal with them. The following conditions, for example, may cause your program to stop or to function in a way you had not anticipated:

- Statements that cannot be executed
- Inappropriate input from the user
- Interrupts from the user

This chapter explains how you can identify these types of errors and how you can respond to them.

Determining Whether an Error Occurred: the **status** *Variable*

The **status** variable is a global variable that indicates whether an error has occurred during the execution of an INFORMIX-4GL statement. If INFORMIX-4GL executes a statement successfully, the value of **status** will be zero. If an error occurs during the execution of a statement, **status** will be a negative number. (Specifically, **status** will be the number of the error message; see the *INFORMIX-4GL Reference Manual* for a list of error

messages.) After a FOREACH statement or a SELECT statement that returns no rows, the value of **status** will be 100 or NOTFOUND.

Note: The value of the **status** variable will also be NOTFOUND if you use the FETCH statement to move the cursor beyond the last row retrieved by a SELECT statement. (See the *INFORMIX-4GL Reference Manual* for more information about FETCH.)

When the user runs a program, **INFORMIX-4GL** automatically tests the **status** variable after every statement to determine whether an error has occurred. If the value of **status** is less than zero, **INFORMIX-4GL** displays an error message and stops the program. An error message shows the line of the program file in which the error occurred, as well as a brief description of the problem. An example follows:

```
Program error at custmenu.4gl line 206
SQL statement error number -1215.
Value too large to fit in an INTEGER
```

If you do not want your program to stop in this way, you can test whether the value of the **status** variable is less than zero and take appropriate action if you include a WHENEVER statement in your program. The WHENEVER statement is described in the following section.

Trapping Errors: the WHENEVER Statement

You can use the WHENEVER statement to trap errors and warnings that occur during the execution of an **INFORMIX-4GL** program. This chapter describes three forms of the WHENEVER statement that you can use to trap errors. See the *INFORMIX-4GL Reference Manual* for complete information about the other forms of the WHENEVER statement that you can use to trap warnings.

The forms of the WHENEVER statement discussed in this chapter are

```
WHENEVER ERROR STOP
WHENEVER ERROR CONTINUE
WHENEVER ERROR CALL function-name
```

You can use WHENEVER ERROR STOP if you want INFORMIX-4GL to exit the program immediately when an error occurs (**status**< 0). Since this is the default condition, you only need to include WHENEVER ERROR STOP in your program if you want to override a previous WHENEVER ERROR statement.

Use the WHENEVER ERROR CONTINUE statement if you want INFORMIX-4GL to continue execution when an error occurs. You should include this statement in your program only if you plan to handle error trapping and recovery yourself.

You can use WHENEVER ERROR CALL *function-name* if you want INFORMIX-4GL to execute the statements in the named function when an error occurs. Include this statement in your program if you plan to write your own error-handling routine but want to let INFORMIX-4GL test for errors.

You may use one or more WHENEVER ERROR statements in a program. A WHENEVER ERROR statement is operative from its physical position in an INFORMIX-4GL source file to the next WHENEVER ERROR statement in the file (if any).

Here is an example:

```
DATABASE stores

MAIN

        WHENEVER ERROR STOP

        CALL abc()

        SELECT * FROM customer

END MAIN


FUNCTION abc()

        WHENEVER ERROR CONTINUE

END FUNCTION
```

If an error occurs during the SELECT statement in this example, the program will stop since SELECT occurs physically before WHENEVER ERROR CONTINUE in this file.

Note: The WHENEVER ERROR statement is a compile-time switch. If your program consists of separately compiled modules, using the WHENEVER ERROR statement in the main module does not affect the other modules.

While you are developing an application, you probably will not use WHENEVER ERROR statements in your program (except WHENEVER ERROR STOP), since **INFORMIX-4GL** displays error messages that may prove useful. After you test an application, however, you might want to write an error-handling routine that displays a message for the user, records the error in a log file, and stops the program. The following section describes two functions you can use if you want to handle errors in this way.

The *startlog* Function

You can use the **startlog** function to create a file in which INFORMIX-4GL records an error message every time an error occurs (assuming you have not specified WHENEVER ERROR CONTINUE). This allows you to keep a record of runtime errors, which makes debugging easier. In its simplest form, the **startlog** function has the following format

```
CALL startlog("filename")
```

where *filename* is the name of the error-log file.

If you use the **startlog** function in your program, INFORMIX-4GL either creates a new file (if a file with the given name *does not* exist) or appends error messages to an existing file (if a file with the given name *does* exist). For example, the statement

```
CALL startlog("error_log")
```

either creates a new file called **error_log** in the current directory or appends error messages to an existing file called **error_log** in the current directory.

Each time an error occurs, INFORMIX-4GL records the date and time, as well as the line of the program in which the error occurred and a brief description of the problem. A typical error-log entry follows:

```
DATE: 10/17/1986     Time: 15:51:36
Program error at custmenu.4gl line 206
SQL statement error number -1215.
Value too large to fit in an INTEGER
```

The *errorlog* Function

If you create an error-log file, you can record your own messages in the file using the **errorlog** function. The **errorlog** function can take two forms:

CALL errorlog("*message*")

CALL errorlog(*variable*)

where *message* is any character string enclosed in double quotes and *variable* is any CHAR variable.

After you create an error-log file, you can use the **errorlog** function to append your own messages to provide additional documentation when an error occurs. When INFORMIX-4GL encounters this function, it records the date and time in the error-log file along with your message.

For example, the statement

```
CALL errorlog("This is a test of the errorlog function.")
```

produces the following entry in the error-log file:

```
Date: 10/17/1986     Time: 15:51:36
This is a test of the errorlog function.
```

A Sample Program for Handling Errors

The **ch10when** program shows how to use the **startlog** function and the WHENEVER ERROR CALL statement to control the termination of a program if an error occurs while it is running:

```
DATABASE stores

GLOBALS

        DEFINE p_customer RECORD LIKE customer.*

END GLOBALS


MAIN

        DEFINE answer CHAR(1)

        CALL startlog("error_log")

        WHENEVER ERROR CALL exit_now

        OPEN FORM cust_form FROM "customer"

        DISPLAY FORM cust_form

        LET answer = "y"

        WHILE answer = "y"

                CALL enter_cust()

                PROMPT "Do you want to enter another row (y/n) ? "
                        FOR answer

        END WHILE

        MESSAGE "End program."

        SLEEP 3

        CLEAR SCREEN

END MAIN


FUNCTION enter_cust()

        . . .

END FUNCTION


FUNCTION exit_now()

        CLEAR SCREEN

        DISPLAY "There is a serious problem. ",
                 "Call your applications designer for help."

        EXIT PROGRAM

END FUNCTION
```

If a runtime error occurs, INFORMIX-4GL records an error message in the **error_log** file and calls the **exit_now** function that clears the screen, displays a message for the user, and stops the program.

If, for example, the program cannot find the **customer** form, it will display the message

```
There is a serious problem. Call your applications designer for help.
```

on screen and record the following error message in the **error_log** file:

```
Date 10/18/1986     Time: 17:12:12
Program error at ch10when.4gl line 16
FORMS statement error number -1110.
Form file not found
```

Note: The WHENEVER statement has no effect when certain runtime errors occur. For example, if INFORMIX-4GL runs out of memory, your program will stop even if it contains a WHENEVER ERROR CONTINUE or WHENEVER ERROR CALL statement.

Handling Inappropriate Input

The previous section describes fatal errors that will cause your program to stop unless you include a WHENEVER ERROR CONTINUE or WHENEVER ERROR CALL statement in your program. This section discusses non-fatal errors that will cause your program to malfunction rather than to stop.

In some cases, you can anticipate the types of non-fatal errors that may occur and write procedures to handle them. If your program performs some operation based on input from the user, you can either test the input before processing it or take remedial action if the input generates a non-fatal error. The approach you take depends on the type of input you request and how you plan to use it.

For example, suppose your program selects a customer row based on a customer number that the user enters and displays it on a screen form. You can ensure that the user enters a valid customer number in two ways:

1. Test the input against the existing customer numbers before executing the SELECT statement.

2. Use the input in the SELECT statement and then test the **status** variable to determine whether **INFORMIX-4GL** found a row with the given customer number. If the value of **status** is 0 (indicating that a row was found), you can display the row on the screen. If the value of **status** is 100 (NOT-FOUND), you can have the user enter another customer number and repeat the process.

Note: As explained in the previous section, if the value of **status** is less than zero (indicating a fatal error), **INFORMIX-4GL** immediately stops execution unless you include a WHENEVER ERROR CONTINUE or WHENEVER ERROR CALL statement in your program.

The **ch10notf** program shows how to use the second approach to select a customer row and display it on the screen:

```
DATABASE stores

MAIN

        DEFINE p_customer RECORD LIKE customer.*

        OPEN FORM cust_form FROM "customer"

        DISPLAY FORM cust_form

        WHILE TRUE

                PROMPT "Enter a customer number :  "
                        FOR p_customer.customer_num

                SELECT * INTO p_customer.* FROM customer
                        WHERE customer_num = p_customer.customer_num

                IF status = 0 THEN

                        EXIT WHILE

                END IF

        END WHILE

        DISPLAY BY NAME p_customer.*

        SLEEP 3

        CLEAR SCREEN

END MAIN
```

When INFORMIX-4GL encounters the WHILE statement in this program, it performs the following operations:

1. Prompts the user to enter a customer number and assigns it to the variable **p_customer.customer_num**.

2. Attempts to select a row where the value in the **customer_num** column equals the value of the variable **p_customer.customer_num**.

3. Exits the WHILE statement if **status** equals 0 (indicating that the SELECT statement retrieved a row). Otherwise, INFORMIX-4GL executes the statements in the WHILE loop again since the Boolean expression TRUE is always true.

Note: Do not define the **status** variable; it is defined by INFORMIX-4GL as a global variable.

Handling Interrupts from the User

The first part of this chapter describes the conditions under which INFORMIX-4GL stops a program automatically. This section explains how the user can stop a program and describes how you can prevent this from happening, if you choose.

The INTERRUPT and QUIT Keys

Unless you intervene, the user can stop a program at any time by pressing the INTERRUPT key (usually DEL for UNIX systems and CONTROL-C for DOS systems) or the QUIT key (usually CONTROL-\ for UNIX systems). In some cases, you may want to let the user use one of these keys to interrupt a process without stopping the program entirely. For example, you can let the user use the INTERRUPT key to cancel a PROMPT, INPUT, INPUT ARRAY, or CONSTRUCT statement if you trap the INTERRUPT key so that it does not stop your program. The following section shows you how to handle interrupts during a PROMPT or INPUT statement in this way. (You can use the same method to trap the INTERRUPT key during an INPUT ARRAY or CONSTRUCT statement. See Chapters 11 and 12 for more information about INPUT ARRAY and CONSTRUCT.)

The DEFER Statement

You can use the DEFER statement to keep your program from stopping when the user presses the INTERRUPT or QUIT keys. The DEFER statement can take the following forms:

```
DEFER INTERRUPT
DEFER QUIT
```

The DEFER INTERRUPT statement is described in the following section. (See the *INFORMIX-4GL Reference Manual* for more information on DEFER QUIT.)

Note: Unlike WHENEVER, the DEFER statement is a runtime switch. Using the DEFER statement in the main program *does* affect other modules at run time.

The DEFER INTERRUPT Statement

You can use DEFER INTERRUPT in the main program to keep your program from stopping when the user presses the INTERRUPT key. Once the DEFER INTERRUPT statement is executed, however, the original function of the INTERRUPT key cannot be restored.

If the user presses the INTERRUPT key after the DEFER INTERRUPT statement has been executed, INFORMIX-4GL sets the global variable **int_flag** to non-zero instead of stopping the program. This means that you can test **int_flag** after a PROMPT or INPUT statement and take appropriate action if it indicates that the INTERRUPT key was pressed. Since INFORMIX-4GL does not reset the flag to zero, make sure you reset it yourself so that subsequent tests will be valid. (Otherwise, the **int_flag** will falsely indicate an interrupt even if the user did not press the INTERRUPT key again.)

Note: Do not define the **int_flag** variable; it is defined by INFORMIX-4GL as a global variable.

The following examples show how to use the DEFER INTERRUPT statement and **int_flag** to test whether the user has pressed the INTERRUPT key during an INPUT or PROMPT statement.

Example 1

The **ch10def** program allows the user to enter zero or more customer rows on the **customer** form. If the user presses **ESCAPE** during the INPUT statement, the program adds the current row to the database. If, however, the user presses the INTERRUPT key, the program discards the current row and displays an appropriate message.

```
DATABASE stores

GLOBALS

        DEFINE p_customer RECORD LIKE customer.*

END GLOBALS


MAIN

        DEFINE answer CHAR(1)

        OPEN FORM cust_form FROM "customer"

        DISPLAY FORM cust_form

        DEFER INTERRUPT

        LET answer = "y"

        WHILE answer = "y"

                CALL enter_cust()

                PROMPT "Do you want to enter another row (y/n) ? "
                        FOR answer

        END WHILE

        MESSAGE "End program."

        SLEEP 3

        CLEAR SCREEN

END MAIN
```

```
FUNCTION enter_cust()

        CLEAR FORM

        DISPLAY "Press ESC to add.  Press DEL to discard." AT 2,1

        LET int_flag = 0

        INPUT p_customer.fname THRU p_customer.phone
                FROM sc_cust.*

        DISPLAY "" AT 2,1

        IF int_flag <> 0 THEN

                MESSAGE "Row discarded."

                SLEEP 3

                MESSAGE ""

                CLEAR FORM

                RETURN

        END IF

        LET p_customer.customer_num = 0

        INSERT INTO customer VALUES (p_customer.*)

        LET p_customer.customer_num = SQLCA.SQLERRD[2]

        DISPLAY p_customer.customer_num TO customer_num

        MESSAGE "Row added."

        SLEEP 3

        MESSAGE ""

END FUNCTION
```

The program includes a DEFER INTERRUPT statement and a conditional call to the function **enter_cust** that performs the following operations:

1. Clears the **customer** form.

2. Displays a message telling the user to either press **ESCAPE** to add a row or press **DEL** to discard a row.

3. Sets **int_flag** to zero prior to the INPUT statement to ensure that the subsequent test is valid.

4. Allows the user to enter customer information on the screen form and stores the values in the record **p_customer**. If the user presses **ESCAPE** to add the row, **int_flag** remains zero. If the user presses **DEL** to discard the row, the function sets **int_flag** to non-zero.

5. If the value of **int_flag** is non-zero, **enter_cust** displays a message indicating that the current row has been discarded, clears the form, and returns control to the main program.

6. If the value of **int_flag** is zero, **enter_cust** inserts the values in the **p_customer** record into the **customer** table and displays the serial number for the row on screen, along with the message Row added.

Example 2

You can use a similar technique to determine if the user pressed the INTERRUPT key during a PROMPT statement. For example, the **ch10def2** program selects customer rows based on a last name that the user supplies in response to a prompt. If the user presses INTERRUPT during the PROMPT statement, the program aborts the search for customer rows and will display an appropriate message.

```
DATABASE stores
GLOBALS
        DEFINE p_customer RECORD LIKE customer.*
END GLOBALS

MAIN
        DEFER INTERRUPT
        OPEN FORM cust_form FROM "customer"
        DISPLAY FORM cust_form
        CALL query_data()
        MESSAGE "End program."
        SLEEP 3
        CLEAR SCREEN
END MAIN
```

```
FUNCTION query_data()

        DEFINE last_name CHAR (15),
                answer CHAR(1),
                exist SMALLINT

        LET int_flag = 0

        PROMPT "Enter a last name or DEL to quit : " FOR last_name

        IF int_flag <> 0 THEN

                MESSAGE "Query aborted."

                SLEEP 3

                MESSAGE ""

                RETURN

        END IF

        MESSAGE "Selecting rows for customers with the last name ",
                last_name, ". . ."

        SLEEP 3

        MESSAGE ""

        DECLARE a_curs CURSOR FOR
                SELECT * FROM customer WHERE lname MATCHES last_name

        LET exist = 0

        FOREACH a_curs INTO p_customer.*

                LET exist = 1

                DISPLAY p_customer.* TO customer.*

                PROMPT "Do you want to see the next customer (y/n) : "
                        FOR answer

                IF answer = "n" THEN

                        EXIT FOREACH

                END IF

        END FOREACH

        IF exist = 0 THEN

                MESSAGE "No customer rows found."

        ELSE

                IF answer = "y" THEN

                        MESSAGE "There are no more customer rows."

                END IF

        END IF

        SLEEP 3

END FUNCTION
```

An Alternative to the INTERRUPT Key

You may prefer to have the user cancel a PROMPT or INPUT statement by pressing another key instead of INTERRUPT. If you choose this method, you do not have to include the DEFER INTERRUPT statement in your program and continually test and reset **int_flag**. The following section describes how to use the ON KEY clause with INPUT and PROMPT to provide the user with such an alternative.

More About INPUT: The ON KEY Clause

You can use the ON KEY clause in an INPUT statement to execute a series of statements after the user presses a designated key. The ON KEY clause has the following format

```
ON KEY (key-list)
   statement
   ...
   [EXIT INPUT]
```

where *key-list* is a list of one or more function or CONTROL keys, *statement* is an **INFORMIX-4GL** statement, and EXIT INPUT are optional keywords that terminate the INPUT statement immediately.

Note: When you use an optional clause in an INPUT statement, you must conclude the INPUT statement with the END INPUT keywords.

You can include either function or CONTROL keys in the *key-list*. The notation for function keys is F1, F2, and so on, through F36, while the notation for CONTROL keys is CONTROL-A, CONTROL-B, and so on, through CONTROL-Z.

The following table lists the keys that you *cannot* include in the ON KEY clause.

Keys that Cannot be Used in the Key-List

INTERRUPT*
ESCAPE†
CONTROL-A
CONTROL-D
CONTROL-H
CONTROL-L
CONTROL-R
CONTROL-X

* unless you execute a DEFER INTERRUPT statement

† unless you specify another key as the ACCEPT KEY in the OPTIONS statement

In addition, you may not be able to use other keys in the ON KEY clause, depending on your operating system. (For example, CONTROL-C is usually the INTERRUPT key on DOS systems.)

Using the ON KEY Clause in the PROMPT Statement

You can use the ON KEY clause to execute a series of statements and leave the PROMPT statement after the user presses a designated key. The ON KEY clause has the following format

```
ON KEY (key-list)
   statement
   ...
```

where *key-list* is a list of one or more function or CONTROL keys and *statement* is an **INFORMIX-4GL** statement. As you can see, this ON KEY clause does not include the optional keywords EXIT PROMPT since **INFORMIX-4GL** automatically exits the PROMPT statement when the user presses a key in the *key-list*.

The notation for function and CONTROL keys, as well as restrictions on the *key-list*, has already been described in the

previous section. When you use this optional clause in the PROMPT statement, make sure you conclude the statement with the END PROMPT keywords.

For example, the following programs allow the user to use CONTROL-E to back out of INPUT and PROMPT statements. As you will see, they are very similar to the sample programs in the section "The DEFER INTERRUPT Statement," earlier in this chapter, except for the omission of DEFER INTERRUPT and **int_flag**, and the inclusion of the ON KEY clause.

Example 1

The **ch10key** program allows the user to enter zero or more customer rows on the **customer** form. If the user presses ESCAPE, the program adds the current row to the database. If, however, the user presses CONTROL-E, the program discards the current row and displays the appropriate message.

```
DATABASE stores

GLOBALS

        DEFINE p_customer RECORD LIKE customer.*

END GLOBALS


MAIN

        DEFINE answer CHAR(1)

        OPEN FORM cust_form FROM "customer"

        DISPLAY FORM cust_form

        LET answer = "y"

        WHILE answer = "y"

                CALL enter_cust()

                PROMPT "Do you want to enter another row (y/n) ?"
                        FOR answer

        END WHILE

        MESSAGE "End program."

        SLEEP 3

        CLEAR SCREEN

END MAIN
```

```
FUNCTION enter_cust()
        DEFINE stop_now SMALLINT
        CLEAR FORM
        LET stop_now = 0
        DISPLAY "Press ESC to add.  Press CONTROL-E to discard." AT 2,1
        INPUT p_customer.fname THRU p_customer.phone
                FROM sc_cust.*
                ON KEY (control-e)
                        LET stop_now = 1
                        EXIT INPUT
        END INPUT
        DISPLAY "" AT 2,1
        IF stop_now = 1 THEN
                MESSAGE "Row discarded."
                SLEEP 3
                MESSAGE ""
                CLEAR FORM
                RETURN
        END IF
        LET p_customer.customer_num = 0
        INSERT INTO customer VALUES (p_customer.*)
        LET p_customer.customer_num = SQLCA.SQLERRD[2]
        DISPLAY p_customer.customer_num TO customer_num
        MESSAGE "Row added."
        SLEEP 3
        MESSAGE ""
END FUNCTION
```

This program contains a function called **enter_cust** that performs the following operations:

1. Clears the **customer** form.

2. Assigns the value 0 to the local variable **stop_now** to indicate that CONTROL-E has not been pressed.

3. Displays a message telling the user to either press ESCAPE to add a row or press CONTROL-E to discard a row.

4. Allows the user to enter customer information on the screen form and stores the values in the record **p_customer**. If the user presses ESCAPE to add a row, the value of **stop_now** remains unchanged. If the user presses CONTROL-E to discard a row, the function assigns the value 1 to **stop_now** and terminates the INPUT statement.

5. If the value of **stop_now** is 1, **enter_cust** displays a message indicating that the current row has been discarded, clears the form, and returns control to the main program.

6. If the value of **stop_now** is 0, **enter_cust** inserts the values in the **p_customer** record into the **customer** table and displays the serial number for the row on screen, along with the message Row added.

Example 2

You can use a similar technique to determine whether the user has pressed a designated key during a PROMPT statement. The **ch10key2** program selects customer rows based on a last name that the user supplies in response to a prompt. If the user presses CONTROL-E during the PROMPT statement, the program aborts the search for customer rows and displays an appropriate message.

```
DATABASE stores
GLOBALS
        DEFINE p_customer RECORD LIKE customer.*
END GLOBALS

MAIN
        OPEN FORM cust_form FROM "customer"
        DISPLAY FORM cust_form
        CALL query_data()
        MESSAGE "End program."
        SLEEP 3
        CLEAR SCREEN
END MAIN
```

```
FUNCTION query_data()

        DEFINE last_name CHAR (15),
                answer CHAR(1),
                stop_now, exist SMALLINT

        LET stop_now = 0

        PROMPT "Enter a last name or CONTROL-E to quit : " FOR last_name

                ON KEY (control-e)

                        LET stop_now = 1

        END PROMPT

        IF stop_now = 1 THEN

                MESSAGE "Query aborted."

                SLEEP 3

                MESSAGE ""

                RETURN

        END IF


        MESSAGE "Selecting rows for customers with the last name ",
                last_name, ". . ."

        SLEEP 3

        MESSAGE ""

        DECLARE a_curs CURSOR FOR
                SELECT * FROM customer WHERE lname MATCHES last_name

        LET exist = 0

        FOREACH a_curs INTO p_customer.*

                LET exist = 1

                DISPLAY p_customer.* TO customer.*

                PROMPT "Do you want to see the next customer (y/n) : "
                        FOR answer

                IF answer = "n" THEN

                        EXIT FOREACH

                END IF

        END FOREACH

        IF exist = 0 THEN

                MESSAGE "No customer rows found."

        ELSE

                IF answer = "y" THEN

                        MESSAGE "There are no more customer rows."

                END IF

        END IF

        SLEEP 3

END FUNCTION
```

Chapter Summary

- INFORMIX-4GL sets the **status** variable after every statement. If INFORMIX-4GL executes the statement successfully, then **status** is zero. If the statement is not executed successfully, **status** is less than zero. After a FOREACH statement or a SELECT statement that returns no rows, **status** equals 100 (NOTFOUND).

- You can use the WHENEVER ERROR STOP statement if you want INFORMIX-4GL to exit your program immediately when an error occurs. (This is the default condition.)

- You can use the WHENEVER ERROR CONTINUE statement if you want INFORMIX-4GL to continue execution when an error occurs.

- Use the WHENEVER ERROR CALL *function-name* statement if you want INFORMIX-4GL to execute the statements in the named function when an error occurs.

- You can create a file in which INFORMIX-4GL records error messages by using the **startlog** function.

- You can use the **errorlog** function to write your own messages in the error-log file after it has been created.

- You can let the user cancel a PROMPT or INPUT statement by pressing the INTERRUPT key if you trap this key so it does not stop the program.

- You can use the DEFER INTERRUPT statement with the global variable **int_flag** to keep your program from stopping when the user presses INTERRUPT.

- As an alternative, you can use the ON KEY clause to let the user cancel a PROMPT or INPUT statement by pressing another key instead of INTERRUPT.

Chapter 11

Using Multiple-Table Forms and Screen Arrays

Chapter 11 Table of Contents

Chapter Overview 5

Working with Multiple-Table Forms 6
The **order** Form 6
The DATABASE Section 8
The SCREEN Section 8
The TABLES Section 9
The ATTRIBUTES Section 9
The INSTRUCTIONS Section 10
Defining Program Arrays 12
Working with Arrays: the FOR Statement 15

Using a Multiple-Table Form in a Program 19
The INPUT ARRAY Statement 19
Scrolling and Editing 21
Built-in Functions for Program Arrays and Screen Arrays 23
Using INPUT ARRAY with Optional Clauses 25
The WITHOUT DEFAULTS Clause 26
The BEFORE FIELD Clause 27
The AFTER FIELD Clause 29
The BEFORE INSERT Clause 36
The AFTER INSERT Clause 38
The AFTER DELETE Clause 46
The DISPLAY ARRAY Statement 51
The **ordmenu** Program 52
The *show_menu* Function 63
The *enter_order* Function 64
The *get_order* Function 65
The *change_order* Function 67
The *delete_order* Function 68

Chapter Summary 69

Chapter Overview

This chapter explains how to write programs that use multiple-table forms for entering, retrieving, updating, and deleting information in a database. The discussion includes

- How to create a multiple-table screen form
- How to work with screen arrays
- How to use the INPUT ARRAY statement for entering data
- How to use the DISPLAY ARRAY statement to display data
- How to look up information or perform calculations, and display the results on screen

The examples in this chapter are based on the following programs and forms included with the demonstration database:

```
ordmenu.4gl
order.per
```

For complete information about the statements used with multiple-table screen forms, see the *INFORMIX-4GL Reference Manual.*

Working with Multiple-Table Forms

This chapter explains how to write programs using the **order** form that comes with the INFORMIX-4GL demonstration database.

The **order** *Form*

The **order** form allows the user to enter, retrieve, change, and delete orders for a given customer. Since each order includes customer information and a list of items, the form must be able to process data from the following tables:

Table	Type of Data
customer	Customer number, name, address, and telephone number.
orders	Order number, order date, and shipment date.
items	Item number, stock number, manufacturer code, quantity, and total price.
stock	Description and unit price.

These tables appear in the **order** form specification file that follows:

```
DATABASE stores

SCREEN
{

                                ORDER FORM

                    Customer Number:  [f000       ]

   First Name:  [f001            ]    Last Name:  [f002            ]
      Address:  [f003                 ]
                [f004                 ]
         City:  [f005            ]    State:  [a0]    Zip:  [f006 ]
    Telephone:  [f007               ]

 Order No: [f008       ]   Order Date: [f009       ]   Ship Date: [f010       ]

Item No.  Stock No.  Code    Description    Quantity    Price       Total

[f011   ]  [f012   ]  [a1 ] [f013           ] [f014   ] [f015      ] [f016      ]
[f011   ]  [f012   ]  [a1 ] [f013           ] [f014   ] [f015      ] [f016      ]
[f011   ]  [f012   ]  [a1 ] [f013           ] [f014   ] [f015      ] [f016      ]
[f011   ]  [f012   ]  [a1 ] [f013           ] [f014   ] [f015      ] [f016      ]

}
END

TABLES
customer
orders
items
stock

ATTRIBUTES
f000 = customer.customer_num;
f001 = customer.fname;
f002 = customer.lname;
f003 = customer.address1;
f004 = customer.address2;
f005 = customer.city;
a0 = customer.state;
f006 = customer.zipcode;
f007 = customer.phone;

f008 = orders.order_num;
f009 = orders.order_date;
f010 = orders.ship_date;

f011 = items.item_num, NOENTRY;
f012 = items.stock_num;
a1 = items.manu_code;
f013 = stock.description, NOENTRY;
f014 = items.quantity;
f015 = stock.unit_price, NOENTRY;
f016 = items.total_price, NOENTRY;
END

INSTRUCTIONS
SCREEN RECORD s_items[4] (item_num, stock_num, manu_code, description,
                          quantity, unit_price, total_price)
END
```

The following sections describe the parts of this multiple-table form in detail.

The DATABASE Section

Like the other forms in this guide, the **order** form gives the user access to tables in the **stores** database.

The SCREEN Section

The screen form has two sections. The first section is designed to handle customer information like the **customer** form described in Chapter 6. The second section is designed to handle order information and includes a *screen array* for listing the items on order, as shown in this example:

```
Item No.  Stock No.  Code   Description    Quantity    Price       Total

[f011   ] [f012   ] [a1 ] [f013          ] [f014   ] [f015      ] [f016      ]
[f011   ] [f012   ] [a1 ] [f013          ] [f014   ] [f015      ] [f016      ]
[f011   ] [f012   ] [a1 ] [f013          ] [f014   ] [f015      ] [f016      ]
[f011   ] [f012   ] [a1 ] [f013          ] [f014   ] [f015      ] [f016      ]
```

The screen array in this example consists of four identical rows of display fields that allow the user to work with up to four items at one time. As you will see, this screen array is actually a *window* on a larger program array that can store information for up to ten items.

The following illustration shows the relationship between the screen array and the program array used in the sample program in this chapter.

Item No	Stock No	Code	Description	Quantity	Price	Total
1	8	ANZ	volleyball	2	840.00	1680.00
2	5	SMT	tennis racquet	15	25.00	375.00
[3]	[2]	[HRO]	[baseball]	[1]	[126.00]	[126.00]
[4]	[5]	[NRG]	[tennis racquet]	[10]	[28.00]	[280.00]
[5]	[6]	[SMT]	[tennis ball]	[2]	[36.00]	[72.00]
[6]	[3]	[HSK]	[baseball bat]	[3]	[240.00]	[720.00]
7	1	HRO	baseball gloves	5	250.00	1250.00
8	9	ANZ	volleyball net	12	20.00	240.00
9	4	HSK	football	3	960.00	2880.00
10	1	SMT	baseball gloves	2	450.00	900.00

The user can scroll through the items in a program array by using the **UP ARROW**, **DOWN ARROW**, PAGE-UP, and PAGE-DOWN keys during an INPUT ARRAY or a DISPLAY ARRAY statement (see the section "Scrolling and Editing," later in this chapter, for more information).

The TABLES Section

The TABLES section includes the **customer**, **orders**, **items**, and **stock** table.

The ATTRIBUTES Section

The ATTRIBUTES section lists the field tags with their corresponding field names and attributes (if any). Each field tag appears only once in the ATTRIBUTES section, even though it may appear more than once in the SCREEN section if it is associated with a screen array.

The screen array in this example contains three types of display fields: random-access, look-up, and calculated fields. The user has access to the **stock_num**, **manu_code**, and **quantity** fields so a stock number, manufacturer code, and a quantity for a given item can be specified. **description** and **unit_price** are look-up fields; when the user enters a stock number and a manufacturer code for an item, the program looks up the description and unit price and displays them in these fields. **item_num** and **total_price** are calculated fields; the program automatically displays an item number (based on the position of an item in the program array) in the **item_num** field and a total price (based on a quantity and unit price) in the **total_price** field.

Note: All look-ups and calculations are handled in the INPUT ARRAY statement, not in the form specification. See the section "The INPUT ARRAY Statement," later in this chapter, for more information.

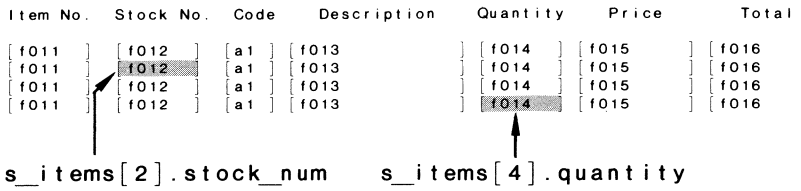
Since the cursor should not move into look-up or calculated fields in this example, **description**, **unit_price**, **item_num**, and **total_price** are designated as NOENTRY fields. (See Chapter 6 or the *INFORMIX-4GL Reference Manual* for more information about the NOENTRY attribute.)

The INSTRUCTIONS Section

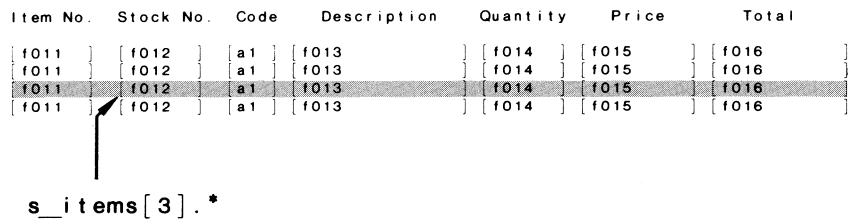
The **order** form contains a screen record called **s_items** that corresponds to the screen array described above:

```
INSTRUCTIONS
SCREEN RECORD s_items[4] (item_num, stock_num, manu_code, description,
              quantity, unit_price, total_price)
END
```

This screen record consists of four rows of the screen fields **item_num** through **total_price**. You can refer to a field in the screen array in an **INFORMIX-4GL** program by specifying the row number and the field name as shown in the following example:



Similarly, you can refer to an entire row of a screen array in an **INFORMIX-4GL** program by specifying the row number and including the **.*** notation:



Note: **INFORMIX-4GL** lets you use any simple integer expression to specify the array subscript of a screen record.

In addition to **s_items**, the **order** form specification includes four implicit screen records, one for each table listed in the TABLES section:

Screen Record	Associated Field Names
customer	customer_num THRU phone
orders	order_num THRU ship_date
items	item_num, stock_num, manu_code, quantity, total_price
stock	description, unit_price

Defining Program Arrays

Usually you will want to use a program array to store values that the user enters in the display fields of a screen array. You can define a program array using the DEFINE statement with the ARRAY keyword. In simplest form, the DEFINE statement for a program array has the following format

DEFINE *variable* ARRAY[*n*] OF *data-type*

where *variable* is the name of the program array, *n* is the number of elements in the array, and *data-type* is the data type of each element. In the DEFINE ARRAY statement, the square brackets are required and do not represent an option.

For example, you can use the following statements to define an array of five integers and assign the value 2 to the fourth element of the array:

```
DEFINE p_array ARRAY[5] OF integer
LET p_array[4] = 2
```

INFORMIX-4GL allows you to define arrays of records as well as arrays of simple variables. For example, you can use the following statement to define a program array that contains ten records consisting of the variables **item_num** through **total_price:**

```
DEFINE

p_items   ARRAY[10] OF RECORD

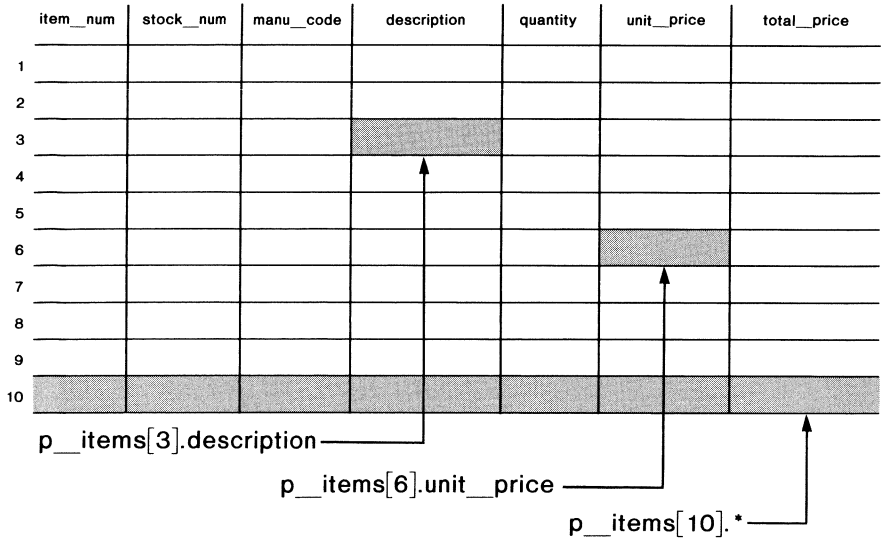
          item_num LIKE items.item_num,
          stock_num LIKE items.stock_num,
          manu_code LIKE items.manu_code,
          description LIKE stock.description,
          quantity LIKE items.quantity,
          unit_price LIKE stock.unit_price,
          total_price LIKE items.total_price

          END RECORD
```

As you can see, the variables in each record of this program array correspond to the display fields in each row of the screen array **s_items**:

```
SCREEN RECORD s_items[4] (item_num, stock_num, manu_code,
description, quantity, unit_price, total_price)
```

Like the screen array, you can refer to individual variables in the program array by specifying a record number and a variable name. Similarly, you can refer to an entire record of the program array by specifying the record number and the .* notation:



Note: A record in the program array **p_items** is referred to as a "row" throughout the rest of this chapter.

Working with Arrays: the FOR Statement

You will often need to use the FOR statement when working with program and screen arrays. With the FOR statement, you can execute a group of statements a specified number of times. The FOR statement has the following format

```
FOR variable = integer-expr TO integer-expr

statement
   ...
   [CONTINUE FOR]
   ...
   [EXIT FOR]
   ...
END FOR
```

where *variable* is a program variable or "counter" of type SMALLINT or INTEGER, *integer-expr* is an expression that evaluates to a SMALLINT or INTEGER, and *statement* is an **INFORMIX-4GL** statement. The FOR loop repeats the sequence of statements as the *variable* takes on successive values from the first *integer-expr* to the second *integer-expr* (inclusive).

Here are the guidelines you should follow when using the FOR statement:

- Assign the variable a range of integer expressions after the FOR keyword.

  Note: If the first integer expression is greater than the second integer expression, **INFORMIX-4GL** will not execute the statements in the FOR loop unless you include an optional STEP keyword followed by a negative integer expression. See the *INFORMIX-4GL Reference Manual* for details.

- List the statements you want **INFORMIX-4GL** to execute as the variable takes on each successive value.

- Use CONTINUE FOR or EXIT FOR if you want to interrupt the sequence of statements in a FOR loop. The CONTINUE FOR statement returns program control to the "top" of the loop. The EXIT WHILE statement causes INFORMIX-4GL to execute the first statement that follows the END FOR keywords.

- Include the END FOR keywords after the last statement you want to execute. When INFORMIX-4GL encounters the END FOR keywords, it executes the statements in the FOR loop again if the next value of the variable is less than or equal to the value of the second integer expression.

Example 1. The following example shows how to use the FOR loop to erase the information in the screen array **s_items**:

```
DEFINE counter SMALLINT

FOR counter = 1 TO 4

      CLEAR s_items[counter].*

END FOR
```

In this example, INFORMIX-4GL executes the CLEAR statement four times as **counter** takes on successive values from 1 to 4:

Counter	Statement executed
1	CLEAR s_items[1].*
2	CLEAR s_items[2].*
3	CLEAR s_items[3].*
4	CLEAR s_items[4].*

Example 2. The following example shows how to use a FOR loop to insert rows in the **items** table using an order number and information stored in the program array **p_items**:

```
DEFINE counter,
       arr_count        SMALLINT

LET arr_count = 3

FOR counter = 1 TO arr_count

       INSERT INTO items
              VALUES (p_items[counter].item_num,
                      p_orders.order_num,
                      p_items[counter].stock_num,
                      p_items[counter].manu_code,
                      p_items[counter].quantity,
                      p_items[counter].total_price)

END FOR
```

In this example, **INFORMIX-4GL** executes the INSERT statement three times as the counter takes on successive values from 1 to 3:

Counter	Statement executed
1	INSERT INTO items VALUES (p__items[1].item__num, p__orders.order__num, p__items[1].stock__num, p__items[1].manu__code, p__items[1].quantity, p__items[1].total__price)
2	INSERT INTO items VALUES (p__items[2].item__num, p__orders.order__num, p__items[2].stock__num, p__items[2].manu__code, p__items[2].quantity, p__items[2].total__price)
3	INSERT INTO items VALUES (p__items[3].item__num, p__orders.order__num, p__items[3].stock__num, p__items[3].manu__code, p__items[3].quantity, p__items[3].total__price)

Using a Multiple-Table Form in a Program

This section explains how to write a program that allows the user to enter, retrieve, change, and delete information on the **order** form. The program relies on two statements, INPUT ARRAY and DISPLAY ARRAY, which control the interaction between the user and the program. You can assign values to a program array from data entered by the user in a screen array with the INPUT ARRAY statement. The DISPLAY ARRAY statement allows you to display the values in a program array to a screen array.

The INPUT ARRAY Statement

The INPUT ARRAY statement performs two important functions. First, it allows the user to enter data in a screen array such as **s_items** with the help of special keys for scrolling and editing. Second, it assigns the values in the screen array to variables in a program array like **p_items**. In addition, you can use INPUT ARRAY to execute **INFORMIX-4GL** statements before or after the user enters data in a particular field or when the user inserts or deletes a row in the screen array. This means that you can perform look-ups and calculations like those described earlier in this chapter in the section "The ATTRIBUTES Section."

This section explains some of the most important features of the INPUT ARRAY statement. (Refer to the *INFORMIX-4GL Reference Manual* for a full description.)

In simplest form, the INPUT ARRAY statement has the following format

INPUT ARRAY *program-array* FROM *screen-array.**

where *program-array* is an array of records (defined in your **INFORMIX-4GL** program) and *screen-array* is the name of a screen array (defined in your form specification). When you use the INPUT ARRAY statement, make sure that the variables in each row of the program array correspond to the

display fields in each row of the screen array. The number of rows in the program array may exceed the number of rows in the corresponding screen array, however.

For example, you can use the following INPUT ARRAY statement to assign values entered by the user in the screen array **s_items** to the program array **p_items**:

```
INPUT ARRAY p_items FROM s_items.*
```

When INFORMIX-4GL encounters the INPUT ARRAY statement in this example, it performs the following operations:

- Displays default values in the first row of **s_items** and moves the cursor to the first field permitting data entry in that row.

- Waits for the user to press ESCAPE after entering data in the screen array. (If you do not want ESCAPE to be used as the accept key, you can specify another value in the ACCEPT KEY clause of the OPTIONS statement. See the *INFORMIX-4GL Reference Manual* for details.)

  Even though the screen array can only display four rows at once, the user may enter up to ten rows (since **p_items** is an array of ten records). In a given row, the user may enter values in all fields except those that were assigned the NOENTRY attribute in the form specification. (See the section "Scrolling and Editing" for more information about using the scrolling and editing keys for data entry.)

  Note: If you want the user to be able to enter more than ten rows in a screen array, simply increase the number of rows that the corresponding program array can store.

- Assigns the values in the screen array to the program array as they are entered by the user.

When INFORMIX-4GL encounters an INPUT ARRAY statement, it positions the cursor in the first field of the screen array that permits data entry. When the user presses RETURN or TAB, the cursor moves to the next display field that permits data entry. In this way, the user can enter data until all rows in the program array have been filled. (If the user tries to add a row when the program array is full, INFORMIX-4GL will not accept it and displays a warning message.) The user can stop data entry at any point by moving the cursor out of the current row, or the user can end the INPUT ARRAY statement entirely by pressing ESCAPE. (Unlike INPUT, the user cannot end the INPUT ARRAY statement by pressing RETURN.) Pressing DEL instead of ESCAPE stops the program unless the DEFER INTERRUPT statement has been executed.

During an INPUT ARRAY statement, the user can use the ARROW keys and four function keys for scrolling and editing. The following table lists these keys and the operations they perform.

LEFT ARROW

Inside a display field, the LEFT ARROW key moves the cursor one character position to the left without erasing the current character. If you press the LEFT ARROW key when the cursor is in the first character position of a display field, the cursor moves to the first character position of the previous display field in the current row. Pressing LEFT ARROW when the cursor is in the first character position of a row moves the cursor to the first character position in the previous row. (The LEFT ARROW key skips over display fields that were assigned the NOENTRY attribute in the form specification.)

RIGHT ARROW

Inside a display field, the RIGHT ARROW key moves the cursor one character position to the right without erasing the current character. If you press the RIGHT ARROW key

when the cursor is in the last character position of a display field, the cursor moves to the first character position of the next display field in the current row. Pressing RIGHT ARROW when the cursor is in the last character position of a row moves the cursor to the first character position in the next row.

UP ARROW

The UP ARROW key moves the cursor to the same display field in the previous row of the screen array. If you press the UP ARROW key when the cursor is in the first row of the screen array, the cursor stays in the same position and the window moves so that the previous row of the program array becomes visible. Pressing the UP ARROW key when the cursor is on the first row of the program array displays a message indicating that no rows precede the first row.

DOWN ARROW

The DOWN ARROW key moves the cursor to the same display field in the next row of the screen array. If you press the DOWN ARROW key when the cursor is in the last row of the screen array, the cursor will stay in the same position and the window will move so that the next row of the program array becomes visible. Pressing the DOWN ARROW key when the cursor is on the blank row beneath the last row of the program array displays a message.

F1

The F1 key inserts a blank row at the current location and moves the cursor to the beginning of the blank row.

F2

The F2 key deletes the current row and moves the following display up one row.

F3

The F3 key scrolls the window to the next full page of program array rows.

F4

The F4 key scrolls the window to the previous full page of program array rows.

Note: You can assign these operations to different keys by using the OPTIONS statement. See the *INFORMIX-4GL Reference Manual* for more information.

Built-in Functions for Program Arrays and Screen Arrays

INFORMIX-4GL has four built-in functions to help you keep track of the relative state of the program array and the screen array during an INPUT ARRAY statement. These functions are described below:

arr_curr( ) Returns the number of the current program array row.

arr_count( ) Returns the number of rows currently stored in the program array.

scr_line( ) Returns the number of the current screen array row. (This number may not be the same as **arr_curr( )** if the program array is larger than the screen array.)

set_count(x) Sets the initial value of **arr_count( )**, and takes the number of rows currently stored in the program array as an argument. You must call this function before executing the INPUT ARRAY WITHOUT DEFAULTS or DISPLAY ARRAY statement. (See the section "The DISPLAY ARRAY Statement" for more information about DISPLAY ARRAY.)

For example, suppose the **s__items** screen array and the **p__items** program array contain the following rows:

Item No.	Stock No.	Code	Description	Quantity	Price	Total
1	8	ANZ	volleyball	2	840.00	1680.00
2	5	SMT	tennis racquet	15	25.00	375.00
[3]	[2]	[HRO]	[baseball]	[1]	[126.00]	[126.00]
[4]	[5]	[NRG]	[tennis racquet]	[10]	[28.00]	[280.00]
[5]	[6]	[SMT]	[tennis ball]	[2]	[36.00]	[72.00]
[6]	[3]	[HSK]	[baseball bat]	[3]	[240.00]	[720.00]

If the cursor is on item 3, the built-in functions perform the following operations:

- **arr__curr( )** returns the value 3.
- **arr__count( )** returns the value 6.
- **scr__line** returns the value 1.
- **set__count(6)** sets the initial value of **arr__count( )** prior to an INPUT ARRAY WITHOUT DEFAULTS or a DISPLAY ARRAY statement.

Using INPUT ARRAY with Optional Clauses

You can include one or more optional clauses in the INPUT ARRAY statement to display messages, perform calculations, look up information, and perform other useful tasks. The following table shows the general format for an INPUT ARRAY statement that includes the optional clauses to be discussed in this chapter:

```
INPUT ARRAY program-array [WITHOUT DEFAULTS]
    FROM screen-array.*

    [BEFORE FIELD field-list
        statement
        ... ]

    [AFTER FIELD field-list
        statement
        ... ]

    [BEFORE INSERT
        statement
        ... ]

    [AFTER INSERT
        statement
        ... ]

    [AFTER DELETE
        statement
        ... ]

END INPUT
```

Note: The END INPUT keywords are required only when you include one or more optional clauses in your INPUT ARRAY statement.

In addition to the clauses listed above, you can also use BEFORE ROW, AFTER ROW, BEFORE DELETE, and AFTER INPUT in an INPUT ARRAY statement. (See the *INFORMIX-4GL Reference Manual* for details.)

The WITHOUT DEFAULTS Clause

To allow the user to make changes to existing rows of the database, you want to display current values in the program array on screen before the user begins changing data. To do this, you must specify the number of records currently stored in the program array and then execute the INPUT ARRAY statement with the WITHOUT DEFAULTS clause. When **INFORMIX-4GL** encounters this type of INPUT ARRAY statement, it displays the current values in the program array in the associated fields of the screen array before accepting input from the user. For example,

```
DECLARE q_curs CURSOR FOR
        SELECT item_num, items.stock_num,
               items.manu_code, description,
               quantity, unit_price, total_price
        FROM items, stock
        WHERE order_num = 1011
               AND items.stock_num = stock.stock_num
               AND items.manu_code = stock.manu_code

LET counter = 1

FOREACH q_curs INTO p_items[counter].*

        LET counter = counter + 1

        IF counter > 10 THEN

               EXIT FOREACH

        END IF

END FOREACH

CALL set_count(counter - 1)

INPUT ARRAY p_items WITHOUT DEFAULTS FROM s_items.*
```

When INFORMIX-4GL encounters the statements in this example, it performs the following operations:

- Declares a cursor for a SELECT statement that selects all the items for the order with the number 1011.
- Assigns the value 1 to the variable **counter**. This variable serves as an index into the program array **p_items**.
- Repeatedly retrieves an item from the database, stores it in the row **p_items[counter].***, and increments **counter** until all items in the given order are processed or the program array becomes full.
- Calls the **set_count( )** function with the argument counter-1 so that the program knows the number of rows currently stored in the program array **p_items**. (You must call this function before using the INPUT ARRAY statement with the WITHOUT DEFAULTS clause.)
- Displays the current values in **p_items** on screen and waits for the user to press ESCAPE after adding, changing, or deleting data in the screen array.
- Assigns the values in the screen array to the program array.

The BEFORE FIELD Clause

You can use the BEFORE FIELD clause to execute a series of statements before the user enters or changes information in a given field. The BEFORE FIELD clause has the following format

```
BEFORE FIELD field-list
   statement
   ...
```

where *field-list* is a list of one or more display fields and *statement* is any INFORMIX-4GL statement except INPUT, INPUT ARRAY, or PROMPT. INFORMIX-4GL executes the statements following the BEFORE FIELD keywords after the user moves

the cursor into one of the named fields but before the user enters or changes data.

Here is an example:

```
INPUT ARRAY p_items FROM s_items.*

            BEFORE FIELD stock_num
                    MESSAGE "Enter a stock number"

            BEFORE FIELD manu_code
                    MESSAGE "Enter the code for a manufacturer"

            BEFORE FIELD quantity
                    MESSAGE "Enter a quantity"

END INPUT
```

When **INFORMIX-4GL** encounters the BEFORE FIELD clauses in this INPUT ARRAY statement, it performs the following tasks:

- Displays the message Enter a stock number as soon as the user moves the cursor into any **stock_num** field in the screen array.

- Displays the message Enter a code for a manufacturer as soon as the user moves the cursor into any **manu_code** field in the screen array.

- Displays the message Enter a quantity as soon as the user moves the cursor into any **quantity** field in the screen array.

The AFTER FIELD Clause

You can use the AFTER FIELD clause to execute a series of statements after the user enters or changes information in a given field. The AFTER FIELD clause has the following format

AFTER FIELD *field-list*
  statement
  ...

where *field-list* is a list of one or more display fields and *statement* is any **INFORMIX-4GL** statement except INPUT, INPUT ARRAY, or PROMPT. **INFORMIX-4GL** executes the statements following the AFTER FIELD keywords when the user presses **RETURN** or **ESCAPE** in one of the named fields. The following example shows how to use the AFTER FIELD clause in an INPUT ARRAY statement to perform look-ups and calculations:

```
INPUT ARRAY p_items FROM s_items.*

        AFTER FIELD stock_num, manu_code
            MESSAGE ""

            LET pa_curr = arr_curr()

            IF p_items[pa_curr].stock_num IS NOT NULL
                AND p_items[pa_curr].manu_code IS NOT NULL
            THEN

                CALL get_item()

                IF p_items[pa_curr].quantity IS NOT NULL THEN
                    CALL get_total()
                END IF

            END IF


        AFTER FIELD quantity
            MESSAGE ""

            LET pa_curr = arr_curr()

            IF p_items[pa_curr].stock_num IS NOT NULL
                AND p_items[pa_curr].manu_code IS NOT NULL
                AND p_items[pa_curr].quantity IS NOT NULL
            THEN

                CALL get_total()

            END IF

END INPUT
```

The INPUT ARRAY statement in this example contains two AFTER FIELD clauses that are described in the following sections.

AFTER FIELD **stock_num, manu_code.** INFORMIX-4GL executes the statements that follow this AFTER FIELD clause when the user presses RETURN or ESCAPE in any **stock_num** or **manu_code** field. In simplest terms, the statements in this AFTER FIELD clause perform the look-ups and the calculation described here:

- If values are entered in both the **stock_num** and **manu_code** fields of the current row, INFORMIX-4GL calls the **get_item** function, which looks up the description and unit price and displays them in the corresponding screen fields of the current row.

- If, in addition, a value is entered in the **quantity** field of the current row, INFORMIX-4GL calls the **get_total** function, which multiplies the quantity by the unit price and displays the total in the corresponding screen field of the current row.

The following example shows the operation of this AFTER FIELD clause more clearly. Suppose the user decides to change the stock number for the indicated item from 1 to 6.

The following illustration shows the values in the screen array and program array before the user enters the new stock number:

Item No.	Stock No.	Code	Description	Quantity	Price	Total
1	8	ANZ	volleyball	2	840.00	1680.00
2	5	SMT	tennis racquet	15	25.00	375.00
[3]	[2]	[HRO]	[baseball]	[1]	[126.00]	[126.00]
[4]	[5]	[NRG]	[tennis racquet]	[10]	[28.00]	[280.00]
[5]	[3]	[HSK]	[baseball bat]	[3]	[240.00]	[720.00]
[6]	[1]	[SMT]	[baseball gloves]	[2]	[450.00]	[900.00]

The following illustration shows the values in the screen array and the program array after the user changes the stock number:

Item No.	Stock No.	Code	Description	Quantity	Price	Total
1	8	ANZ	volleyball	2	840.00	1680.00
2	5	SMT	tennis racquet	15	25.00	375.00
[3]	[2]	[HRO]	[baseball]	[1]	[126.00]	[126.00]
[4]	[5]	[NRG]	[tennis racquet]	[10]	[28.00]	[280.00]
[5]	[3]	[HSK]	[baseball bat]	[3]	[240.00]	[720.00]
[6]	[6]	[SMT]	[tennis ball]	[2]	[36.00]	[72.00]

As you can see, INFORMIX-4GL looked up the description and unit price for the item with the stock number 6 and the manufacturer code SMT and displayed this information on screen. In addition, INFORMIX-4GL multiplied the quantity by the new unit price and displayed the result in the **total_price** field.

AFTER FIELD quantity. INFORMIX-4GL executes the statements that follow this AFTER FIELD clause when the user presses RETURN in any **quantity** field. In simplest terms, this AFTER FIELD clause calls the **get_total** function, which calculates the total price if values have been entered in the **stock_num**, **manu_code**, and **quantity** fields of the current row.

For example, suppose the user decides to change the quantity for the indicated item from 2 to 3. The following illustration shows the values in the screen array and the program array before the user enters the new quantity:

Item No.	Stock No.	Code	Description	Quantity	Price	Total
1	8	ANZ	volleyball	2	840.00	1680.00
2	5	SMT	tennis racquet	15	25.00	375.00
[3]	[2]	[HRO]	[baseball]	[1]	[126.00]	[126.00]
[4]	[5]	[NRG]	[tennis racquet]	[10]	[28.00]	[280.00]
[5]	[3]	[HSK]	[baseball bat]	[3]	[240.00]	[720.00]
⇨ [6]	[6]	[SMT]	[tennis ball]	[2]	[36.00]	[72.00]

The following illustration shows the values in the screen array and the program array after the user enters the new quantity:

Item No.	Stock No.	Code	Description	Quantity	Price	Total
1	8	ANZ	volleyball	2	840.00	1680.00
2	5	SMT	tennis racquet	15	25.00	375.00
[3]	[2]	[HRO]	[baseball]	[1]	[126.00]	[126.00]
[4]	[5]	[NRG]	[tennis racquet]	[10]	[28.00]	[280.00]
[5]	[3]	[HSK]	[baseball bat]	[3]	[240.00]	[720.00]
[6]	[6]	[SMT]	[tennis ball]	[3]	[36.00]	[108.00]

As you can see, INFORMIX-4GL multiplied the unit price by the quantity and displayed the result in the **total_price** field of the current row.

The **get_item** *and* **get_total** *Functions.* The AFTER FIELD statements described above use the functions **get_item** and **get_total** to perform the look-ups and calculations. The program calls the **get_item** function if both the **stock_num** and **manu_code** variables in the current program array row contain values after the user presses RETURN in either the **stock_num** or **manu_code** field of the corresponding screen array row.

The **get_item** function follows:

```
FUNCTION get_item()

        DEFINE pa_curr, sc_curr SMALLINT

        LET pa_curr = arr_curr()

        LET sc_curr = scr_line()

        SELECT description, unit_price
                INTO    p_items[pa_curr].description,
                        p_items[pa_curr].unit_price
                FROM    stock
                WHERE   stock.stock_num = p_items[pa_curr].stock_num
                        AND stock.manu_code = p_items[pa_curr].manu_code

        DISPLAY p_items[pa_curr].description,
                        p_items[pa_curr].unit_price
                TO s_items[sc_curr].description,
                        s_items[sc_curr].unit_price

END FUNCTION
```

When **INFORMIX-4GL** encounters this function, it performs the following operations:

- Determines the number of the current program array row and assigns it to the local variable **pa_curr**.
- Determines the number of the current screen array row and assigns it to **sc_curr**.
- Retrieves a description and unit price based on the stock number and manufacturer code in the current program array row and stores those values in the appropriate variables.
- Displays the values in the **description** and **unit_price** variables in the corresponding fields of the current screen array row.

The program calls the **get_total** function in the following situations:

- If the **stock_num**, **manu_code**, and **quantity** variables in the current program array row all contain values after the user presses **RETURN** in either the **stock_num** or **manu_code** field of the corresponding screen array row.

- If the **stock_num**, **manu_code**, and **quantity** variables in the current program array row contain values after the user presses **RETURN** in the **quantity** field of the corresponding screen array row.

The **get_total** function follows:

```
FUNCTION get_total()

        DEFINE pa_curr, sc_curr SMALLINT

        LET pa_curr = arr_curr()

        LET sc_curr = scr_line()

        LET p_items[pa_curr].total_price =
            p_items[pa_curr].quantity * p_items[pa_curr].unit_price

        DISPLAY p_items[pa_curr].total_price
            TO s_items[sc_curr].total_price

END FUNCTION
```

The **get_total** function performs the following tasks:

- Determines the number of the current program array row and assigns it to the local variable **pa_curr**.
- Determines the number of the current screen array row and assigns it to **sc_curr**.
- Multiplies **quantity** by **unit_price** in the current program array row and assigns the resulting value to the **total_price** variable.
- Displays the value of the **total_price** variable in the corresponding field of the current screen array row.

The BEFORE INSERT Clause

You can use the BEFORE INSERT clause to execute a series of statements before the user inserts a new row into the screen array. The BEFORE INSERT clause has the following format:

```
BEFORE INSERT
   statement
   ...
```

INFORMIX-4GL executes the statements following the BEFORE INSERT keywords in the following situations:

- When the user begins entering rows into a screen array. In this case, **INFORMIX-4GL** executes the BEFORE INSERT statements before the user enters information in each successive row.

- When the user presses the insert key to insert a new row into the middle of a screen array.

- When the user moves the cursor to the blank row at the end of the screen array. Again, **INFORMIX-4GL** executes the BEFORE INSERT statement before the user enters information in each successive row.

For example, the BEFORE INSERT clause in the following INPUT ARRAY statement calls a function that displays an item number for each row that the user plans to insert:

```
INPUT ARRAY p_items FROM s_items.*

      BEFORE INSERT
             CALL get_item_num()

END INPUT
```

The following illustrations show the operation of this BEFORE INSERT clause in three situations:

```
Item No.   Stock No.   Code   Description      Quantity   Price      Total
[      1] [      □] [    ] [                ] [       ] [        ] [        ]
[        ] [        ] [    ] [                ] [       ] [        ] [        ]
[        ] [        ] [    ] [                ] [       ] [        ] [        ]
[        ] [        ] [    ] [                ] [       ] [        ] [        ]
[        ] [        ] [    ] [                ] [       ] [        ] [        ]

[      1] [      8] [ANZ ] [volleyball       ] [      2] [    840.00] [   1680.00]
[      2] [      □] [    ] [                ] [       ] [        ] [        ]
[        ] [        ] [    ] [                ] [       ] [        ] [        ]
[        ] [        ] [    ] [                ] [       ] [        ] [        ]
[        ] [        ] [    ] [                ] [       ] [        ] [        ]
```

```
[      1] [      8] [ANZ ] [volleyball       ] [      2] [    840.00] [   1680.00]
[      2] [      2] [HRO ] [baseball         ] [      1] [    126.00] [    126.00]
[      3] [      □] [    ] [                ] [       ] [        ] [        ]
[      3] [      6] [SMT ] [tennis ball      ] [      2] [     36.00] [     72.00]
```

```
[      5] [      1] [HRO ] [baseball gloves] [      5] [    250.00] [   1250.00]
[      6] [      9] [ANZ ] [volleyball net  ] [     12] [     20.00] [    240.00]
[      7] [      □] [    ] [                ] [       ] [        ] [        ]
[        ] [        ] [    ] [                ] [       ] [        ] [        ]
```

The **get_item_num** *Function.* The BEFORE INSERT clause described above uses the **get_item_num** function to display the item number of the current program array row before the user begins entering information. The **get_item_num** function is shown below:

```
FUNCTION get_item_num()

        DEFINE pa_curr, sc_curr  SMALLINT

        LET pa_curr = arr_curr()

        LET sc_curr = scr_line()

        LET p_items[pa_curr].item_num = pa_curr

        DISPLAY pa_curr TO s_items[sc_curr].item_num

END FUNCTION
```

This function performs the following operations:

- Determines the number of the current program array row and assigns it to the local variable **pa_curr**.
- Determines the number of the current screen array row and assigns it to **sc_curr**.
- Assigns the value of **pa_curr** to the **item_num** variable in the current program array row.
- Displays the value of **pa_curr** in the **item_num** display field of the corresponding screen array row.

The AFTER INSERT Clause

You can use the AFTER INSERT clause to execute a series of statements after the user inserts a new row into the screen array. The AFTER INSERT clause has the following format:

```
AFTER INSERT
   statement
   ...
```

INFORMIX-4GL executes the statements following the AFTER INSERT keywords when the user presses **UP ARROW**, **DOWN ARROW**, or **ESCAPE** after entering information in all REQUIRED fields in the current row and in any other fields that permit data entry.

For example, the AFTER INSERT clause in the following INPUT ARRAY statement calls a function that renumbers the rows in the program array and displays the new numbers after the user inserts a new row:

```
INPUT ARRAY p_items FROM s_items.*

        AFTER INSERT
              CALL renum_items()

END INPUT
```

The following illustration shows the item numbers in the screen array and program array before the user inserts a new row:

Item No.	Stock No.	Code	Description	Quantity	Price	Total
1	8	ANZ	volleyball	2	840.00	1680.00
2	5	SMT	tennis racquet	15	25.00	375.00
[3]	[2]	[HRO]	[baseball]	[1]	[126.00]	[126.00]
[4]	[5]	[NRG]	[tennis racquet]	[10]	[28.00]	[280.00]
[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]
[5]	[6]	[SMT]	[tennis ball]	[2]	[36.00]	[72.00]
6	3	HSK	baseball bat	3	240.00	720.00

The following illustration shows the item numbers in the screen array and program array after the user inserts a new row:

Item No.	Stock No.	Code	Description	Quantity	Price	Total
1	8	ANZ	volleyball	2	840.00	1680.00
2	5	SMT	tennis racquet	15	25.00	375.00
[3]	[2]	[HRO]	[baseball]	[1]	[126.00]	[126.00]
[4]	[5]	[NRG]	[tennis racquet]	[10]	[28.00]	[280.00]
[5]	[1]	[SMT]	[baseball gloves]	[2]	[450.00]	[900.00]
[6]	[6]	[SMT]	[tennis ball]	[2]	[36.00]	[72.00]
7	3	HSK	baseball bat	3	240.00	720.00

As you can see, the program renumbers the items in the screen array and program array automatically after the user inserts the new row.

The **renum_items** *Function.* The AFTER INSERT clause described above uses the **renum_items** function to renumber the items in the screen array and program array after the user inserts a new row. This function follows:

```
FUNCTION renum_items()

        DEFINE pa_curr,
                       pa_total,
                       sc_curr,
                       sc_total,
                       k          SMALLINT

        LET pa_curr = arr_curr()

        LET pa_total = arr_count()

        LET sc_curr = scr_line()

        LET sc_total = 4

        FOR k = pa_curr TO pa_total

                       LET p_items[k].item_num = k

                       IF sc_curr <= sc_total THEN

                              DISPLAY k TO s_items[sc_curr].item_num

                              LET sc_curr = sc_curr + 1

                       END IF

        END FOR

END FUNCTION
```

When INFORMIX-4GL encounters this function, it performs the following operations:

- Assigns the number of the current program array row to the variable **pa_curr**.
- Assigns the number of rows currently stored in the program array to the variable **pa_total**.
- Assigns the number of the current screen array row to the variable **sc_curr**.
- Assigns the total number of rows in the screen array to the variable **sc_total**.

- Uses a FOR loop to renumber the rows in the program array from the current row to the last row, as well as renumber the rows in the screen array from the current row to the last displayed row.

The following illustrations show the operation of the **renum__items** function more clearly.

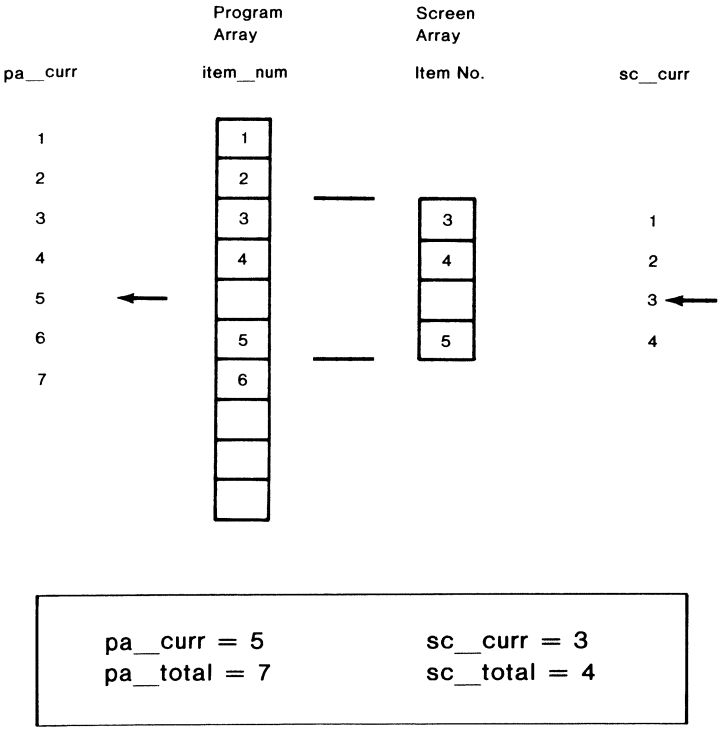
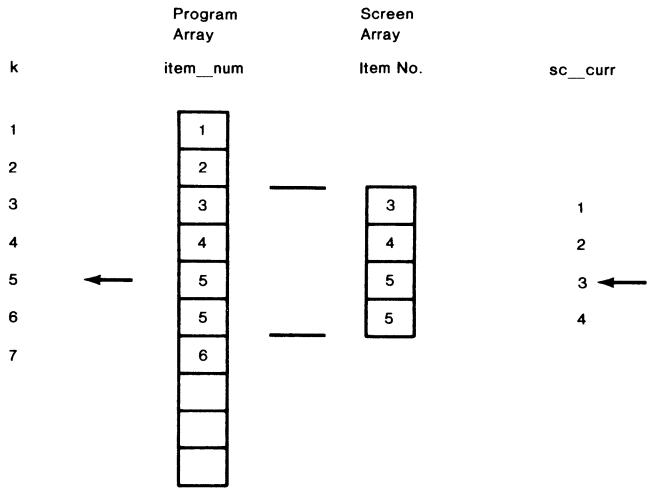


Figure 11-1.

Figure 11-1 shows the state of the program array and screen array before **renum__items** executes the FOR loop that renumbers the rows in each array.



The renum_items Function (Detail)

Values: k = 5 sc_curr = 3 sc_total = 4

Before value substitution:

```
LET p_items[k].item_num = k

IF sc_curr <= sc_total THEN

   DISPLAY k
     TO s_items[sc_curr].item_num

   LET sc_curr = sc_curr + 1

END IF
```

After value substitution:

```
LET p_items[5].item_num = 5

IF 3 <= 4 THEN

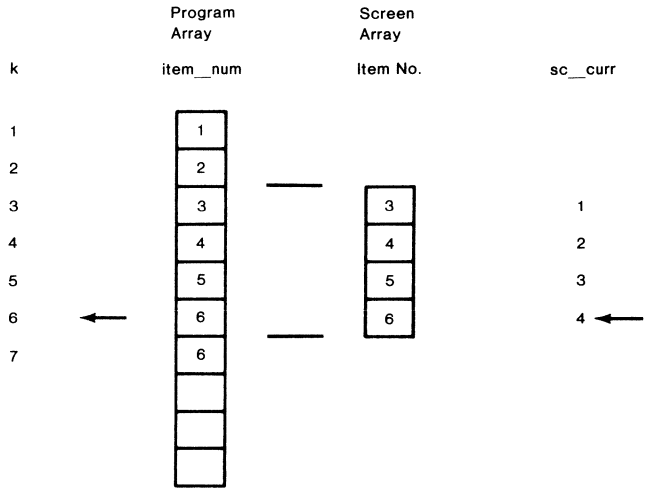
   DISPLAY 5
     TO s_items[3].item_num

   LET sc_curr = 3 + 1

END IF
```

Figure 11-2.

Figure 11-2 shows the state of the program array and screen array after **renum_items** executes the statements in the FOR loop using 5 as the value of **k**. As you can see, item 5 appears in both the fifth row of the program array and in the corresponding row of the screen array.



The renum_items Function (Detail)

Values: k = 6 sc_curr = 4 sc_total = 4

Before value substitution:

```
LET p_items[k].item_num = k

IF sc_curr <= sc_total THEN

   DISPLAY k
     TO s_items[sc_curr].item_num

   LET sc_curr = sc_curr + 1

END IF
```

After value substitution:

```
LET p_items[6].item_num = 6

IF 4 <= 4 THEN

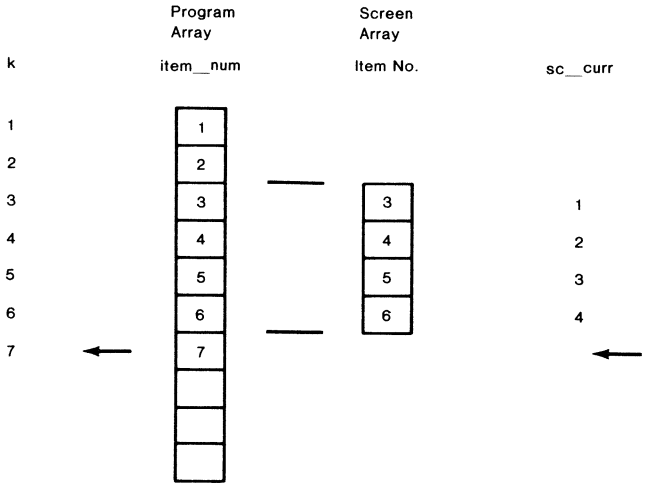
   DISPLAY 6
     TO s_items[4].item_num

   LET sc_curr = 4 + 1

END IF
```

Figure 11-3.

Figure 11-3 shows the state of the program array and screen array after **renum_items** executes the statements in the FOR loop using 6 as the value of **k**. The item has been changed from 5 to 6 in both the sixth row of the program array and in the corresponding row of the screen array.



The renum_items Function (Detail)

Values: k = 7 sc_curr = 5 sc_total = 4

Before value substitution:

```
LET p_items[k].item_num = k

IF sc_curr <= sc_total THEN

   DISPLAY k
     TO s_items[sc_curr].item_num

   LET sc_curr = sc_curr + 1

END IF
```

After value substitution:

```
LET p_items[7].item_num = 7

IF 5 <= 4 THEN

      . . .
      . . .

      . . .

END IF
```

Figure 11-4.

Figure 11-4 shows the state of the program array and screen array after **renum_items** executes the statements in the FOR loop using 7 as the value of **k**. As you can see, item 7 now appears in the seventh row of the program array instead of item 6.

The AFTER DELETE Clause

You can use the AFTER DELETE clause to execute a series of statements after the user deletes a row from the screen array. The AFTER DELETE clause has the following format:

```
AFTER DELETE
   statement
   ...
```

INFORMIX-4GL executes the statements following the AFTER DELETE keywords after the user presses the delete key.

For example, the AFTER DELETE clause in the following INPUT ARRAY statement calls the **renum_items** function, which renumbers the rows in the program array and displays the new numbers on screen after the user deletes a row:

```
INPUT ARRAY p_items FROM s_items.*

        AFTER DELETE
              CALL renum_items()

END INPUT
```

The following illustration shows the item numbers in the screen array and program array before the user deletes a row:

Item No.	Stock No.	Code	Description	Quantity	Price	Total
1	8	ANZ	volleyball	2	840.00	1680.00
2	5	SMT	tennis racquet	15	25.00	375.00
[3]	[2]	[HRO]	[baseball]	[1]	[126.00]	[126.00]
[4]	[5]	[NRG]	[tennis racquet]	[10]	[28.00]	[280.00]
[5]	[6]	[SMT]	[tennis ball]	[2]	[36.00]	[72.00]
[6]	[3]	[HSK]	[baseball bat]	[3]	[240.00]	[720.00]
7	1	HRO	baseball gloves	5	250.00	1250.00

The following illustration shows the item numbers in the screen array and program array after the user deletes the fifth row:

Item No.	Stock No.	Code	Description	Quantity	Price	Total
1	8	ANZ	volleyball	2	840.00	1680.00
2	5	SMT	tennis racquet	15	25.00	375.00
[3]	[2]	[HRO]	[baseball]	[1]	[126.00]	[126.00]
[4]	[5]	[NRG]	[tennis racquet]	[10]	[28.00]	[280.00]
[5]	[3]	[HSK]	[baseball bat]	[3]	[240.00]	[720.00]
[6]	[1]	[HRO]	[baseball gloves]	[5]	[250.00]	[1250.00]

INFORMIX-4GL renumbers the items automatically after the user deletes the fifth row, so that baseball bat becomes the fifth item rather than the sixth item in the list.

The AFTER DELETE clause described above uses the same **renum_items** function as the AFTER INSERT clause described in the previous section. The following figures show the operation of the **renum_items** function when the user deletes rather than inserts a row.

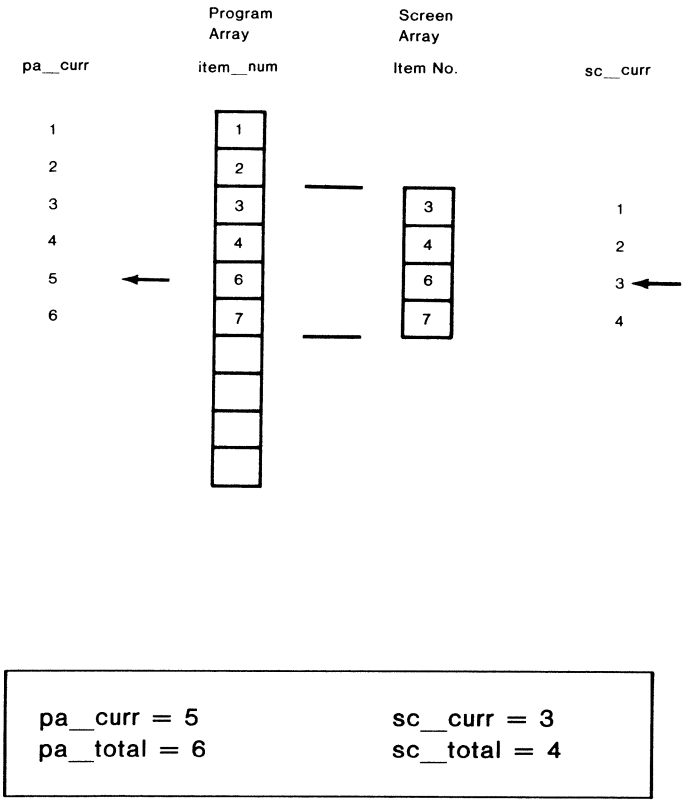
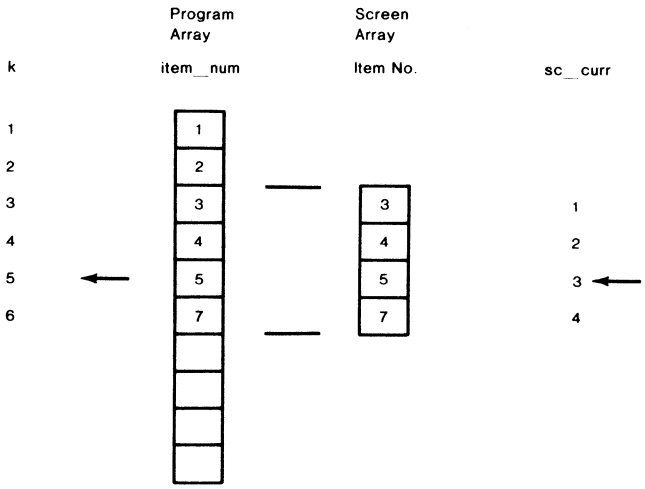


Figure 11-5.

Figure 11-5 shows the state of the program array and screen array before **renum__items** executes the FOR loop that renumbers the items.



The renum_items Function (Detail)

Values: k = 5 sc_curr = 3 sc_total = 4

Before value substitution:

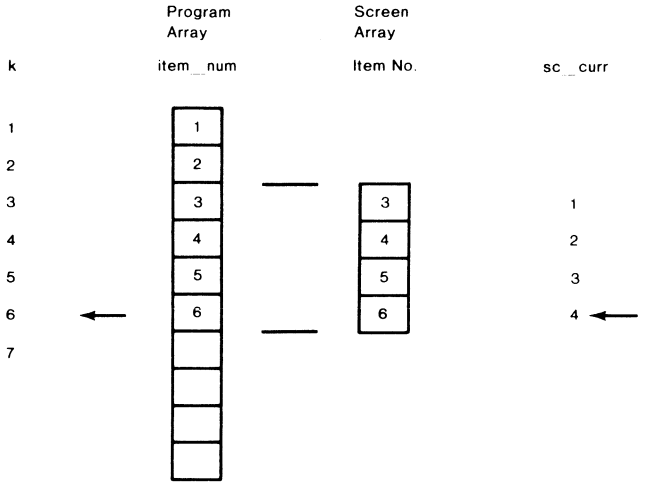
```
LET p_items[k].item_num = k
IF sc_curr <= sc_total THEN
   DISPLAY k
     TO s_items[sc_curr].item_num
   LET sc_curr = sc_curr + 1
END IF
```

After value substitution:

```
LET p_items[5].item_num = 5
IF 3 <= 4 THEN
   DISPLAY 5
     TO s_items[3].item_num
   LET sc_curr = 3 + 1
END IF
```

Figure 11-6.

Figure 11-6 shows the state of the program array and screen array after **renum_items** executes the statements in the FOR loop using 5 as the value of **k**. The item number has been changed from 6 to 5 in both the fifth row of the program array and in the corresponding row of the screen array.



The renum_items Function (Detail)

Values: k = 6 sc_curr = 4 sc_total = 4

Before value substitution:

```
LET p_items[k].item_num = k

IF sc_curr <= sc_total THEN

   DISPLAY k
     TO s_items[sc_curr].item_num

   LET sc_curr = sc_curr + 1

END IF
```

After value substitution:

```
LET p_items[6].item_num = 6

IF 4 <= 4 THEN

   DISPLAY 6
     TO s_items[4].item_num

   LET sc_curr = 4 + 1

END IF
```

Figure 11-7.

Figure 11-7 shows the state of the program array and screen array after **renum_items** executes the statements in the FOR loop using 6 as the value of **k**. As you can see, the item number has been changed from 7 to 6 in both the sixth row of the program array and in the corresponding row of the screen array.

The DISPLAY ARRAY Statement

You can use the DISPLAY ARRAY statement to display the rows in a program array in a screen array. The DISPLAY ARRAY statement also allows the user to use the **ARROW** and function keys to scroll through the rows in the program array.

In simplest form, the DISPLAY ARRAY statement has the following format

```
DISPLAY ARRAY program-array TO screen-array.*
```

where *program-array* is an array of records (defined in your **INFORMIX-4GL** program) and *screen-array* is the name of a screen array (defined in your form specification). When you use the DISPLAY ARRAY statement, make sure that the variables in each row of the program array correspond to the display fields in each row of the screen array. However, the number of rows in the program array may exceed the number of rows in the corresponding screen array.

For example, you can use the following DISPLAY ARRAY statement to display program array rows in a screen array:

```
DISPLAY ARRAY p_items TO s_items.*
```

When **INFORMIX-4GL** encounters the DISPLAY ARRAY statement in this example, it performs the following operations:

- Displays the first four program array rows in the screen array.

- Waits for the user to press **ESCAPE** after using **DOWN ARROW**, **UP ARROW**, **LEFT ARROW**, **RIGHT ARROW**, **RETURN**, **TAB**, **F3**, and **F4** to scroll through the rows in the program array. (If you do not want **ESCAPE** to be used as the accept key, you can specify another value in the ACCEPT KEY clause of the OPTIONS statement. See the *INFORMIX-4GL Reference Manual* for details.)

Note: Since the DISPLAY ARRAY statement does not terminate until the user presses **ESCAPE**, a message should be displayed, telling the user to do so.

The **ordmenu** *Program*

You now have most of the tools needed to write programs that allow the user to enter, retrieve, modify, and delete data through a screen form that includes a screen array. This section contains a sample program called **ordmenu** that lets the user enter, retrieve, modify, and delete orders for a given customer through the **order** form. The program is part of the demonstration database and appears below, along with explanatory notes that describe its **show_menu**, **enter_order**, **get_order**, **change_order**, and **delete_order** functions in detail.

Note: The INPUT ARRAY statements in the **ordmenu** program contain optional clauses that handle look-ups, calculations, and automatic numbering. However, you only need to use a simple INPUT ARRAY statement to have full scrolling and editing capabilities.

```
DATABASE stores

GLOBALS

        DEFINE

        p_customer   RECORD

                     customer_num LIKE customer.customer_num,
                     fname LIKE customer.fname,
                     lname LIKE customer.lname,
                     address1 LIKE customer.address1,
                     address2 LIKE customer.address2,
                     city LIKE customer.city,
                     state LIKE customer.state,
                     zipcode LIKE customer.zipcode,
                     phone LIKE customer.phone

                     END RECORD,

        p_orders     RECORD

                     order_num LIKE orders.order_num,
                     order_date LIKE orders.order_date,
                     ship_date LIKE orders.ship_date

                     END RECORD,

        p_items      ARRAY[10] OF RECORD

                     item_num LIKE items.item_num,
                     stock_num LIKE items.stock_num,
                     manu_code LIKE items.manu_code,
                     description LIKE stock.description,
                     quantity LIKE items.quantity,
                     unit_price LIKE stock.unit_price,
                     total_price LIKE items.total_price

                     END RECORD,

        chosen       SMALLINT

END GLOBALS


MAIN

        OPEN FORM orderform FROM "order"

        DISPLAY FORM orderform

        LET chosen = 0

        CALL show_menu()

        CALL mess("End program.")

        CLEAR SCREEN

END MAIN
```

```
FUNCTION show_menu()
        DEFINE   answer   CHAR(1)
        MENU "ORDER"
             COMMAND "Add" "Add a new order."
                  LET answer = "y"
                  WHILE answer = "y"
                     CALL enter_order()
                     PROMPT "Do you want to ",
                         "enter another order (y/n) : "
                         FOR answer
                  END WHILE
                  LET chosen = 1

             COMMAND "Query" "Search for an order."
                  CALL get_order()

             COMMAND "Modify" "Change an order."
                  IF chosen = 1 THEN
                     CALL change_order()
                  ELSE
                     CALL mess("No order has been chosen.")
                  END IF

             COMMAND "Delete" "Delete an order."
                  IF chosen = 1 THEN
                     CALL clear_menu()
                     PROMPT "Are you sure you want ",
                            "to delete this order (y/n) : "
                            FOR answer
                     IF answer = "y" THEN
                        CALL delete_order()
                        LET chosen = 0
                     END IF

                  ELSE
                     CALL mess("No order has been chosen.")
                  END IF

             COMMAND "Exit" "Exit the ORDER Menu."
                  EXIT MENU
         END MENU
END FUNCTION
```

```
FUNCTION enter_order()

        DEFINE pa_curr      SMALLINT

        CALL clear_menu()

        CLEAR FORM

        CALL get_cust()

        CALL mess("Enter an order date and a ship date.")

        INPUT p_orders.order_date, p_orders.ship_date
             FROM order_date, ship_date

        CALL mess("Enter up to ten items.")


        INPUT ARRAY p_items FROM s_items.*

             BEFORE FIELD stock_num
                  MESSAGE "Enter a stock number."

             BEFORE FIELD manu_code
                  MESSAGE "Enter the code for a manufacturer."

             BEFORE FIELD quantity
                  MESSAGE "Enter a quantity."

             AFTER FIELD stock_num, manu_code
                  MESSAGE ""

                  LET pa_curr = arr_curr()

                  IF      p_items[pa_curr].stock_num IS NOT NULL
                     AND p_items[pa_curr].manu_code IS NOT NULL
                  THEN
                     CALL get_item()

                     IF p_items[pa_curr].quantity IS NOT NULL THEN

                        CALL get_total()

                     END IF

                  END IF

             AFTER FIELD quantity
                  MESSAGE ""

                  LET pa_curr = arr_curr()

                  IF      p_items[pa_curr].stock_num IS NOT NULL
                     AND p_items[pa_curr].manu_code IS NOT NULL
                     AND p_items[pa_curr].quantity IS NOT NULL
                  THEN
                     CALL get_total()

                  END IF

             BEFORE INSERT

                  CALL get_item_num()

             AFTER INSERT

                  CALL renum_items()

             AFTER DELETE

                  CALL renum_items()
```

```
        END INPUT

        INSERT INTO orders (order_num, order_date, customer_num, ship_date)
             VALUES (0,
                  p_orders.order_date,
                  p_customer.customer_num,
                  p_orders.ship_date)

        LET p_orders.order_num = SQLCA.SQLERRD[2]

        DISPLAY p_orders.order_num TO order_num

        CALL insert_items()

        CALL mess("Order added.")

END FUNCTION
```

```
FUNCTION get_order()

        DEFINE counter,exist    SMALLINT,
               answer           CHAR(1)

        CALL clear_menu()

        CLEAR FORM

        CALL get_cust()

        DECLARE q_curs CURSOR FOR
                SELECT order_num, order_date, ship_date
                        FROM orders
                        WHERE customer_num = p_customer.customer_num

        LET exist = 0

        LET chosen = 0

        FOREACH q_curs INTO p_orders.*

                LET exist = 1

                CLEAR orders.*

                FOR counter = 1 TO 4

                        CLEAR s_items[counter].*

                END FOR

                DISPLAY p_orders.* TO orders.*

                DECLARE my_curs CURSOR FOR
                        SELECT item_num, items.stock_num,
                               items.manu_code, description,
                               quantity, unit_price, total_price
                        FROM items, stock
                        WHERE order_num = p_orders.order_num
                               AND items.stock_num = stock.stock_num
                               AND items.manu_code = stock.manu_code
                        ORDER BY item_num

                LET counter = 1

                FOREACH my_curs INTO p_items[counter].*

                        LET counter = counter + 1

                        IF counter > 10 THEN

                                CALL mess("Ten or more items.")

                                EXIT FOREACH

                        END IF

                END FOREACH



                CALL set_count(counter - 1)

                MESSAGE "Press ESC when you finish viewing the items on ord

                DISPLAY ARRAY p_items TO s_items.*

                MESSAGE ""
```

```
                    PROMPT "Enter 'y' to select this order ",
                            "or RETURN to view next order: "
                            FOR answer

                    IF answer = "y" THEN

                            LET chosen = 1

                            EXIT FOREACH

                    END IF

            END FOREACH

            IF exist = 0 THEN

                    CALL mess("No orders found for this customer.")

            ELSE

                    IF chosen = 0 THEN

                            CALL mess("There are no more orders for this custom
                            CLEAR FORM

                    END IF

            END IF

END FUNCTION
```

```
FUNCTION change_order()

        DEFINE pa_curr    SMALLINT,
                  answer      CHAR(1)

        CALL clear_menu()

        PROMPT "Do you want to change the order or ship date (y/n) : "
                FOR answer

        IF answer = "y" THEN

                  INPUT p_orders.order_date, p_orders.ship_date,
                  WITHOUT DEFAULTS
                  FROM order_date, ship_date

        END IF

        PROMPT "Do you want to change any items (y/n) : "
                FOR answer

        IF answer = "y" THEN


                  INPUT ARRAY p_items WITHOUT DEFAULTS FROM s_items.*

                              BEFORE FIELD stock_num
                                   MESSAGE "Enter a stock number."

                              BEFORE FIELD manu_code
                                   MESSAGE "Enter the code for a manufacturer."

                              BEFORE FIELD quantity
                                   MESSAGE "Enter a quantity."

                              AFTER FIELD stock_num, manu_code
                                   MESSAGE ""

                                   LET pa_curr = arr_curr()

                                      IF p_items[pa_curr].stock_num IS NOT NULL
                                      AND p_items[pa_curr].manu_code IS NOT NULL

                                      THEN
                                         CALL get_item()

                                             IF p_items[pa_curr].quantity IS NOT NULL
                                             THEN

                                                CALL get_total()

                                             END IF

                                      END IF

                              AFTER FIELD quantity
                                   MESSAGE ""

                                   LET pa_curr = arr_curr()

                                   IF p_items[pa_curr].stock_num IS NOT NULL
                                      AND p_items[pa_curr].manu_code IS NOT NULL
                                      AND p_items[pa_curr].quantity IS NOT NULL
                                   THEN
                                      CALL get_total()

                                   END IF
```

```
                                BEFORE INSERT

                                    CALL get_item_num()

                                AFTER INSERT

                                    CALL renum_items()

                                AFTER DELETE

                                    CALL renum_items()


                        END INPUT

        END IF

        UPDATE orders
                        SET order_date = p_orders.order_date,
                            ship_date = p_orders.ship_date
                        WHERE order_num = p_orders.order_num

        DELETE FROM items
                        WHERE order_num = p_orders.order_num


        CALL insert_items()

        CALL mess("Order changed.")

END FUNCTION
```

```
FUNCTION delete_order()

        DEFINE counter SMALLINT

        DELETE FROM orders
                WHERE order_num = p_orders.order_num

        DELETE FROM items
                WHERE order_num = p_orders.order_num

        CLEAR orders.*

        FOR counter = 1 to 4

                CLEAR s_items[counter].*

        END FOR

        CALL mess("Order deleted.")

END FUNCTION



FUNCTION get_item()

        DEFINE pa_curr, sc_curr SMALLINT

        LET pa_curr = arr_curr()

        LET sc_curr = scr_line()

        SELECT description, unit_price
                INTO    p_items[pa_curr].description,
                        p_items[pa_curr].unit_price
                FROM    stock
                WHERE   stock.stock_num = p_items[pa_curr].stock_num
                        AND stock.manu_code = p_items[pa_curr].manu_code

        DISPLAY p_items[pa_curr].description,
                        p_items[pa_curr].unit_price
                TO s_items[sc_curr].description,
                        s_items[sc_curr].unit_price

END FUNCTION



FUNCTION get_total()

        DEFINE pa_curr, sc_curr SMALLINT

        LET pa_curr = arr_curr()

        LET sc_curr = scr_line()

        LET p_items[pa_curr].total_price =
                p_items[pa_curr].quantity * p_items[pa_curr].unit_price

        DISPLAY p_items[pa_curr].total_price
                TO s_items[sc_curr].total_price

END FUNCTION
```

```
FUNCTION get_item_num()
        DEFINE pa_curr, sc_curr SMALLINT
        LET pa_curr = arr_curr()
        LET sc_curr = scr_line()
        LET p_items[pa_curr].item_num = pa_curr
        DISPLAY pa_curr to s_items[sc_curr].item_num
END FUNCTION


FUNCTION renum_items()
        DEFINE pa_curr,
                    pa_total,
                    sc_curr,
                    sc_total,
                    k          SMALLINT
        LET pa_curr = arr_curr()
        LET pa_total = arr_count()
        LET sc_curr = scr_line()
        LET sc_total = 4
        FOR k = pa_curr TO pa_total
                    LET p_items[k].item_num = k
                    IF sc_curr <= sc_total THEN
                            DISPLAY k TO s_items[sc_curr].item_num
                            LET sc_curr = sc_curr + 1
                    END IF
        END FOR
END FUNCTION


FUNCTION get_cust()
        WHILE TRUE
                    PROMPT "Enter a customer number:  "
                            FOR p_customer.customer_num
                    SELECT fname, lname, address1, address2,
                            city, state, zipcode, phone
                            INTO p_customer.fname THRU p_customer.phone
                            FROM customer
                            WHERE customer_num = p_customer.customer_num
                    IF status = 0 THEN
                            EXIT WHILE
                    END IF
        END WHILE
        DISPLAY p_customer.* TO customer.*
END FUNCTION
```

```
FUNCTION insert_items()

        DEFINE counter SMALLINT

        FOR counter = 1 TO arr_count()

                INSERT INTO items
                        VALUES (p_items[counter].item_num,
                        p_orders.order_num,
                        p_items[counter].stock_num,
                        p_items[counter].manu_code,
                        p_items[counter].quantity,
                        p_items[counter].total_price)

        END FOR

END FUNCTION



FUNCTION mess(str)

        DEFINE str CHAR(50)

        DISPLAY str CLIPPED AT 23,1

        SLEEP 3

        DISPLAY "" AT 23,1

END FUNCTION



FUNCTION clear_menu()

        DISPLAY "" AT 1,1

        DISPLAY "" AT 2,1

END FUNCTION
```

The *show_menu* Function

The previous program contains a function called **show_menu** that displays an ORDER Menu similar to the CUSTOMER Menu in the sample program in Chapter 8. The ORDER Menu lets the user select **Add**, **Query**, **Modify**, **Delete**, and **Exit** options as orders for a given customer are processed.

The *enter_order* Function

The **enter_order** function allows the user to enter an order for a customer on the **order** form, inserts the order into the **stores** database, and displays the order number for the newly added order. **enter_order** performs the following operations:

1. Calls a function that clears the ORDER Menu.

2. Clears the **order** form.

3. Calls a function that prompts the user to enter a customer number, retrieves a name, address, and phone number based on the customer number, and displays that information on screen.

4. Prompts the user to enter an order date and a shipment date.

5. Uses an INPUT statement to assign values to **p_orders.order_date** and **p_orders.ship_date** from data entered in the **order_date** and **ship_date** fields.

6. Prompts the user to enter up to ten items in the screen array.

7. Uses an INPUT ARRAY statement to assign values to the program array **p_items** from data entered in the screen array **s_items**. As you can see, the INPUT ARRAY statement in this function includes several clauses that were described earlier in this chapter:

   - BEFORE FIELD for displaying messages

   - AFTER FIELD for looking up the description and unit price and calculating the total price for a given item

   - BEFORE INSERT for displaying an item number before the user inserts a row

- AFTER INSERT and AFTER DELETE for renumbering items after the user inserts or deletes a row

8. Inserts a new row in the **orders** table that includes the value 0 as a placeholder for the serial number that INFORMIX-4GL generates automatically, an order date, customer number, and shipment date.

9. Assigns the value in SQLCA.SQLERRD[2] to **p_orders.order_num**. (SQLCA.SQLERRD[2] contains the serial number of the row that INFORMIX-4GL just inserted into the **orders** table.)

10. Displays the order number on screen.

11. Calls a function that inserts the items in the program array into the **items** table.

12. Displays a message indicating that the order has been added to the **stores** database.

The *get_order* Function

The **get_order** function selects all the orders for a given customer and displays them on screen at the user's request. It performs the following operations:

1. Calls a function that clears the ORDERS Menu.

2. Clears the **order** form.

3. Calls a function that prompts the user to enter a customer number, retrieves a name, address, and phone number based on the customer number, and displays that information on screen.

4. Declares a cursor for a SELECT statement that retrieves all the orders placed by the given customer.

5. Assigns the value 0 to the local variable **exist** to indicate that no orders have been retrieved yet.

6. Assigns the value 0 to the global variable **chosen** to indicate that the user has not selected an order yet.

7. Uses a FOREACH loop to display the orders on the screen form. The FOREACH loop performs the following operations:

    a. Selects an order from the **orders** table.

    b. Assigns the value 1 to the local variable **exist** to indicate that at least one order has been selected.

    c. Clears the fields in the screen record **orders** and in the screen array **s_items**.

    d. Displays the order number, order date, and shipment date on the **order** form.

    e. Declares a cursor for a SELECT statement that retrieves all the items for the given order.

    f. Uses a FOREACH statement to store the items for the given order in the program array **p_items**. (If the program array becomes full, the FOREACH loop stops, even if there are more items for the given order.)

    g. Calls the **set_count( )** function to set the initial value of **arr_count( )** to the number of rows stored in the program array.

    h. Uses the DISPLAY ARRAY statement to let the user view the items on order.

    i. Asks whether the user wants to select the current order. If the user enters y in response to the prompt, **INFORMIX-4GL** assigns the value 1 to the global variable **chosen** to indicate that the user has selected an order and exits the FOREACH loop.

j. Repeats steps a—j until all orders are processed, or until the user enters y to choose an order.

8. Displays a message if no orders are placed by the given customer (exist=0).

9. Displays a message and clears the form if the user wants to see another order (chosen=0) and INFORMIX-4GL cannot find one.

The *change_order* Function

The **change_order** function allows the user to change the order that is currently displayed and updates the database accordingly. This function performs the following tasks:

1. Asks if the user wants to change the order date or shipment date.

2. Uses the INPUT statement without defaults to assign values to **p_orders.order_date** and **p_orders.ship_date** if the user enters y.

3. Asks if the user wants to change any items on order.

4. Uses the INPUT ARRAY statement without defaults to assign values to the program array **p_items** from data in the screen array **s_items** if the user enters y in response to the prompt. Again, the INPUT ARRAY statement in this function includes several clauses that were described earlier in this chapter:

   - BEFORE FIELD for displaying messages
   - AFTER FIELD for looking up the description and unit price and calculating the total price for a given item
   - BEFORE INSERT for displaying an item number before the user inserts a row

- AFTER INSERT and AFTER DELETE for renumbering items after the user inserts or deletes a row

5. Updates the order date and shipment date for the current order.

6. Deletes the original items for the current order from the **items** table.

7. Calls a function that inserts the newly modified items into the **items** table.

8. Displays a message indicating that the current order has been changed.

The *delete_order* Function

The **delete_order** function allows the user to delete the order that is currently displayed on screen. The **delete_order** function performs the following operations:

1. Deletes the current order from the **orders** table.

2. Deletes the items for the current order from the **items** table.

3. Clears the fields in the screen record **orders** and in the screen array **s_items**.

4. Displays a message that the order has been deleted.

Chapter Summary

- You can design a screen form that handles data from more than one table in a database.
- You can work with more than one row at a time if your screen form includes a screen array.
- You can use a program array to store values for a screen array.
- You can use the FOR statement when working with program and screen arrays to execute a group of statements a specified number of times.
- You can use the INPUT ARRAY statement to let the user enter data in a screen array and then store it in a program array.
- During the INPUT ARRAY statement, the user can use special keys for scrolling and editing.
- You can use four built-in functions to keep track of the relative state of the program array and screen array during an INPUT ARRAY statement.
- You can use the INPUT ARRAY statement with optional clauses to display messages, perform calculations, look up information, and perform other useful tasks.
- The WITHOUT DEFAULTS clause displays the current values in a program array on screen before the user begins changing data.
- The BEFORE FIELD clause executes a series of statements before the user enters or changes information in a given field.
- The AFTER FIELD clause executes a series of statements after the user enters or changes information in a given field.

- The BEFORE INSERT clause executes a series of statements before the user inserts a new row into the screen array.

- The AFTER INSERT clause executes a series of statements after the user inserts a new row into the screen array.

- The AFTER DELETE clause executes a series of statements after the user deletes a row from the screen array.

- You can display the rows in a program array in a screen array with the DISPLAY ARRAY statement.

Chapter 12

Screen Forms: Query by Example

Chapter 12 Table of Contents

Chapter Overview 5

Query by Example 6
- Specifying Search Criteria 7
- Searching with Relational Operators 8
  - Numeric Fields 9
  - DATE Fields 9
  - MONEY Fields 9
  - CHAR Fields 9
- Using Wildcard Characters 10
- Searching with the Alternation Operator 11
- Searching for a Range of Values 11
- Query Operators and Short Fields 11
- Entering Search Criteria in SMALLFLOAT and FLOAT Fields. 12

Constructing a SELECT Statement from Search Criteria 13
- The CONSTRUCT Statement 14
  - CONSTRUCT BY NAME 16
- Constructing a SELECT Statement 16
- The PREPARE Statement 18
- Executing a Prepared Statement 19
- Sample Programs for Performing Queries by Example 20
  - Example 1 21
  - Example 2 24

Chapter Summary 28

Chapter Overview

This chapter explains how to write programs that retrieve information based on search criteria that the user enters on a screen form. The discussion includes

- How the user specifies search criteria
- How to use the CONSTRUCT statement to let the user perform a query by example
- How to use a LET statement to construct a SELECT statement that satisfies the user's search criteria
- How to use the PREPARE statement to prepare an executable SELECT statement
- How to execute the prepared SELECT statement using a cursor with the DECLARE and FOREACH statements

The examples in this chapter are based on the following programs and forms included with the demonstration database:

```
ch12cust.4gl
ch12ord.4gl
customer.per
order.per
```

For complete information about the statements used for queries by example, see the *INFORMIX-4GL Reference Manual.*

Query by Example

This chapter explains how to write programs using the **customer** and **order** forms that were described in Chapters 6 and 11, respectively. Before you work through the following examples, make sure you are familiar with the screen forms that are discussed in these particular chapters.

You can write programs that let the user specify search criteria by filling in fields on a screen form with the data that is to be found. This process is called *query by example.* What makes this feature so powerful are the relational, range, alternation, and wildcard symbols that the user can include in a query. For example, the user can search for all customers with last names that begin with *B* who live in the city of Palo Alto by filling in the Last Name and City fields of the **customer** form as follows:

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                          CUSTOMER  FORM

       Number: [            ]

   First Name: [                ]     Last Name: [B*               ]

      Company: [                     ]

      Address: [                     ]
               [                     ]

         City: [Palo Alto       ]

        State: [  ]   Zipcode: [      ]

    Telephone: [                  ]
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

Figure 12-1. Query by Example on the **customer** Form

Or, the user can search for all orders placed by customers with the last name "Grant" or "Miller" after June 5, 1986:

```
                                ORDER FORM
--------------------------------------------------------------------------------
                      Customer Number: [            ]

   First Name: [                ]    Last Name: [Grant|Miller    ]
      Address: [                     ]
               [                     ]
         City: [                ]    State: [  ]    Zip: [      ]
    Telephone: [                   ]
--------------------------------------------------------------------------------
 Order No: [            ]  Order Date: [>06/05/86 ]   Ship Date: [           ]

Item No.  Stock No.  Code   Description    Quantity    Price        Total

[       ] [       ] [   ] [                ] [       ] [          ] [          ]
[       ] [       ] [   ] [                ] [       ] [          ] [          ]
[       ] [       ] [   ] [                ] [       ] [          ] [          ]
[       ] [       ] [   ] [                ] [       ] [          ] [          ]
--------------------------------------------------------------------------------
```

Figure 12-2. Query by Example on the **order** Form

The following section describes the different kinds of queries the user can make using relational, range, alternation, and wildcard symbols.

Specifying Search Criteria

To find all rows, the user presses **ESCAPE** without filling in any fields on the screen form. To find rows that contain a particular column value, the user can enter the value in the associated display field. The user can also use

- Relational and range operators to search for values that lie in a particular range.
- Wildcard characters to search for character strings that match a particular pattern.
- An alternation operator to search for alternative values.

These operators are explained in the following sections, and the summary table lists the operators, their names, and the data types that can be used with them.

Operator	Name	Compatible Data Types
=	Equal to	all
>	Greater than	all
<	Less than	all
>=	Greater than or equal to	all
<=	Less than or equal to	all
<>	Not equal	all
*	Wildcard	CHAR
?	Wildcard	CHAR
:	Range	all
|	Alternation	all

Searching with Relational Operators

The first six operators in the table above are *relational operators* that can be used to describe the acceptable values for a field. They are called relational operators because they describe the relationship between a value and the data the user is looking for.

Normally, the user does not have to enter the equal sign; if the user simply enters a value, **INFORMIX-4GL** assumes the meaning "equal to." For example, if the user enters CA in the **state** field of the **order** form, **INFORMIX-4GL** assumes the meaning

```
state="CA"
```

However, the user needs to use an equal sign in two situations:

- The user can search for rows that contain a NULL column by entering an equal sign by itself in the associated display field.

- The user can search for rows that contain an asterisk in a particular CHAR column by entering an equal sign followed by an asterisk (*) in the associated CHAR field. (The user has to use the equal sign with the asterisk because the asterisk is also a query operator.)

Normally, the user enters a relational operator followed by a value, with no blank space between the two. The following sections explain how to use the relational operators in different types of fields. The examples are based on the **order** form in the demonstration database.

Numeric Fields

Enter an operator and a number. For example, to search for all customers with numbers greater than 110, the user enters >110 in the **customer_num** field.

DATE Fields

Enter an operator and a date. *Less than* means before and *greater than* means after the date that is entered. To find orders placed on or before June 5, 1986, the user enters <=6/5/86 in the **order_date** field.

MONEY Fields

Enter an operator and a number, but do not enter a dollar sign or comma (INFORMIX-4GL adds them automatically). To find all items on order where the total price exceeds $250, the user would enter >250 in the **total_price** field.

CHAR Fields

When the user enters a relational operator in a CHAR field

- *Greater than* means later in the alphabet; *less than* means earlier in the alphabet.
- Lowercase letters are greater than uppercase letters.
- Both uppercase and lowercase letters are greater than numbers.

INFORMIX-4GL compares characters based on ASCII values. (A table of ASCII values appears in the *INFORMIX-4GL Reference Manual.*)

To find customers whose names are alphabetized after "Lawson," the user enters >Lawson in the **lname** field. Because lowercase letters are greater than uppercase letters, this search criterion also finds *all* names that begin with lowercase letters.

Using Wildcard Characters

The asterisk (*) and the question mark (?) are wildcard characters that can be used only in CHAR fields. The asterisk replaces any group of zero or more characters, while the question mark replaces any single character.

For example, to find customers whose names start with a capital *S*, the user enters S* in the **lname** field. The user can enter multiple asterisks to search for values that contain a certain pattern of characters. For example, the user might want to search for customers whose last names include the syllable *man*. The user can enter *man* in the **lname** field to search for names like "Brinkman" and "Brinkmann."

The user can use the question mark to find values that match a pattern where the number of characters is fixed. For example, the user can enter Ericks?n to search for the names "Erickson" and "Ericksen." Similarly, the user can enter New??n to search for names like "Newman," "Newton," and "Newson."

Searching with the Alternation Operator

The user can use the alternation operator to search for alternative values in a field. To search for customers with the last name "Grant" or "Lawson," for example, the user could enter Grant | Lawson in the **lname** field. Similarly, the user could enter 112 | 114 in the **customer_num** field to search for the orders placed by customer 112 or customer 114.

Searching for a Range of Values

The colon (:) is the range operator. It lets the user search for all values that lie between one value and another. The range is inclusive.

For example, to search for all customers with numbers between 101 and 115 inclusive, the user enters 101:115 in the **lname** field.

The user can use the range operator with CHAR fields, too. To find names alphabetized after (and including) Smith, omitting names that begin with a lowercase letter, the user can enter Smith:Zzzzzz in the **lname** field.

Query Operators and Short Fields

Occasionally a display field may be too short to hold the search criterion that the user enters. When this occurs, **INFORMIX-4GL** creates a work space on the comment line (by default, line 23 of the screen). When the user presses **RETURN**, the work space disappears. The field contains the criterion the user entered in the work space, even though only part of it fits in the field.

For example, when the user enters 1/1/86:12/31/86 in the **order_date** field, INFORMIX-4GL displays the following information at the bottom of the screen:

```
[1/1/86:12/31/86                                                    ]
```

Entering Search Criteria in SMALLFLOAT and FLOAT Fields

Since the number INFORMIX-4GL displays in a SMALLFLOAT or FLOAT field may differ slightly from the number the user enters, the user may not be able to search a floating-point field for an exact match. For example, if the user enters 1.1 in a floating-point field, INFORMIX-4GL may actually search for 1.10000001 or 1.09999999. The user can solve this by searching for a range of values, such as 1:1.2.

Constructing a SELECT Statement from Search Criteria

The previous section explained how the user specifies search criteria by filling in fields on a screen form with the data to be searched for. This section shows you how to use the search criteria to create a SELECT statement that retrieves the rows specified by the user.

Constructing a SELECT statement for a query by example requires four steps:

1. Use the CONSTRUCT statement to let the user enter search criteria on a screen form and create a character variable that contains conditions for the WHERE clause of a SELECT statement based on the user's input.

2. Concatenate the character variable containing the user's search criteria to one or more character strings that contain the rest of the SELECT statement and store the result in a character variable.

3. Use PREPARE to create an executable statement from the character variable that contains the constructed SELECT statement.

4. Use a cursor with the DECLARE and FOREACH statements to run the prepared SELECT statement.

The following sections describe these procedures in more detail.

The CONSTRUCT Statement

You can use the CONSTRUCT statement to create a CHAR variable that contains the conditions for the WHERE clause of a SELECT statement based on search criteria specified by the user. The CONSTRUCT statement has the following general format

CONSTRUCT *variable* ON *column-list*
FROM *field-list*

where *variable* is a program variable of type CHAR, *column-list* is a list of one or more database column names, and *field-list* is a list of one or more screen field names. When using the CONSTRUCT statement, make sure that *variable* was previously defined as a large character variable and that the column names in the *column-list* correspond to the field names in the *field-list*.

Note: The columns in the *column-list* do not need to belong to the same table.

You should list the column names and corresponding field names for the display fields in which the user specifies search criteria. (The user cannot enter search criteria in a display field if you have not included its field name and associated column in your CONSTRUCT statement.)

For example, you can use the following CONSTRUCT statement to allow the user to enter search criteria in all display fields on the **customer** form:

```
DEFINE query_1 CHAR(300)

CONSTRUCT query_1 ON customer_num, fname, lname, company,
          address1, address2, city, state, zipcode, phone
     FROM customer_num, fname, lname, company,
          address1, address2, city, state, zipcode, phone
```

When INFORMIX-4GL encounters the CONSTRUCT statement in this example, it waits for the user to enter search criteria in one or more of the named fields. When the user presses ESCAPE at any point or RETURN after entering search criteria in the **phone** field, INFORMIX-4GL terminates the CONSTRUCT statement. (If the user presses DEL instead of ESCAPE or RETURN, the program stops unless you have included a DEFER INTERRUPT statement.)

If you want to simplify the CONSTRUCT statement in the previous example, you can make the following substitutions:

- Use the shorthand **customer.*** instead of listing all the columns in the **customer** table.

- Use the default screen record **customer.*** instead of listing all the fields on the **customer** form.

These substitutions have been made in the following CONSTRUCT statement:

```
CONSTRUCT query_1 ON customer.* FROM customer.*
```

When you use shorthands like these, make sure that the columns represented in the ON clause correspond to the fields represented in the FROM clause.

CONSTRUCT BY NAME

If the field names in *field-list* are the same as the corresponding column names in *column-list*, you can use the CONSTRUCT statement with the BY NAME option instead of explicitly naming the display fields. For example, the following CONSTRUCT statements are equivalent; both allow the user to enter search criteria in all display fields on the **customer** form:

```
CONSTRUCT query_1 ON customer_num, fname, lname, company,
          address1, address2, city, state, zipcode, phone
     FROM customer_num, fname, lname, company,
          address1, address2, city, state, zipcode, phone

CONSTRUCT BY NAME query_1 ON customer.*
```

Constructing a SELECT Statement

You create the SELECT statement that satisfies the user's search criteria by concatenating the character variable that contains the search criteria to one or more character strings that contain the rest of the SELECT statement.

For example, you can create the following SELECT statement using the character variable generated by the CONSTRUCT statement in the previous example:

```
DEFINE s1, query_1 CHAR(300),
       p_customer RECORD LIKE customer.*

CONSTRUCT BY NAME query_1 ON customer.*

LET s1 = "SELECT * FROM customer WHERE ",
       query_1 CLIPPED
```

When INFORMIX-4GL encounters the LET statement in this example, it concatenates a character string containing part of the SELECT statement for retrieving customer information to the character variable **query_1**, which contains the conditions for the WHERE clause of the SELECT statement. (As you can see, **query_1** has been CLIPPED to eliminate trailing blanks.) INFORMIX-4GL then assigns the resulting value to **s1**, which has been previously defined as a large character variable.

When you use LET to construct a SELECT statement, make sure you take the following into consideration:

- If a character variable like **s1** is not large enough to store the character string that contains an entire SELECT statement, your program will not work correctly.

- A single character string enclosed in quotes may not exceed one line. If your SELECT statement requires more than one line, make sure you divide the statement into the appropriate character strings and separate them with commas.

- A constructed SELECT statement may not contain an INTO clause.

The PREPARE Statement

Before you can execute a SELECT statement that you have constructed, you must create an executable statement from the character variable that contains the constructed SELECT statement using the PREPARE statement. The PREPARE statement has the following general format

PREPARE *statement-name* FROM *string-spec*

where *statement-name* is a name for the prepared statement and *string-spec* is either a single string constant or a character variable that contains an **RDSQL** statement.

Although you do not have to define the *statement-name* before using it in a PREPARE statement, you do have to execute the PREPARE statement that contains the *statement-name* before using the name elsewhere in the file. Also, you cannot prepare another statement using the same *statement-name* in a given file.

The following example shows how to use the PREPARE statement to prepare the SELECT statement that was constructed in the previous section:

```
DEFINE s1, query_1 CHAR(300),
         p_customer RECORD LIKE customer.*

CONSTRUCT BY NAME query_1 ON customer.*

LET s1 = "SELECT * FROM customer WHERE ",
                    query_1 CLIPPED

PREPARE s_1 FROM s1
```

Executing a Prepared Statement

To execute a prepared SELECT statement, you use a cursor with the DECLARE and FOREACH statements to retrieve the results of the query. For example,

```
DEFINE s1, query_1 CHAR(300),
       p_customer RECORD LIKE customer.*

CONSTRUCT BY NAME query_1 ON customer.*

LET s1 = "SELECT * FROM customer WHERE ",
                    query_1 CLIPPED

PREPARE s_1 FROM s1

DECLARE q_curs CURSOR FOR s_1

FOREACH q_curs INTO p_customer.*

    ...

END FOREACH
```

The statement name, **s_1**, appears after the CURSOR FOR keywords in the DECLARE statement in the previous example.

Note: You can use the EXECUTE statement to run other prepared statements besides the SELECT statement. In simplest form, the EXECUTE statement has the following format

EXECUTE *statement-name*

where *statement-name* is the name of a prepared statement. Once you have prepared a statement, you may execute it as often as you like. (See the *INFORMIX-4GL Reference Manual* for more information about the EXECUTE statement.)

Sample Programs for Performing Queries by Example

You now have most of the tools you will need to write programs that allow the user to perform queries by example on a screen form. This section contains two sample programs, **ch12cust** and **ch12ord**. The first program lets the user enter search criteria on the **customer** form, and the second program allows the user to enter search criteria on the **order** form. Explanatory notes follow each program.

Example 1

The **ch12cust** program retrieves customer rows based on search criteria the user has entered and displays the rows upon request.

```
DATABASE stores
GLOBALS
        DEFINE p_customer RECORD LIKE customer.*
END GLOBALS

MAIN
        OPEN FORM cust_form FROM "customer"
        DISPLAY FORM cust_form
        CALL get_customer()
        MESSAGE "End program."
        SLEEP 3
        CLEAR SCREEN
END MAIN
```

```
FUNCTION get_customer()

        DEFINE   s1, query_1 CHAR(300),
                 exist     SMALLINT,
                 answer    CHAR(1)

        MESSAGE "Enter search criteria for one or more customers."

        SLEEP 4

        MESSAGE ""

        CONSTRUCT BY NAME query_1 ON customer.*

        LET s1 = "SELECT * FROM customer WHERE ", query_1 CLIPPED

        PREPARE s_1 FROM s1

        DECLARE q_curs CURSOR FOR s_1

        LET exist = 0

        FOREACH q_curs INTO p_customer.*

                LET exist = 1

                DISPLAY p_customer.* TO customer.*

                PROMPT "Do you want to see the next customer (y/n) :  "
                        FOR answer

                IF answer = "n" THEN

                        EXIT FOREACH

                END IF

        END FOREACH

        IF exist = 0 THEN

                MESSAGE "No rows found."

        ELSE

                IF answer = "y" THEN

                        MESSAGE "No more rows satisfy the search criteria."

                END IF

        END IF

        SLEEP 3

END FUNCTION
```

As you can see, this program contains a function called **get_customer** that performs the following operations:

1. Displays a message that prompts the user to query for one or more customers.

2. Uses the CONSTRUCT statement to let the user specify search criteria on the **customer** form. When the user presses **ESCAPE** after entering a query, **INFORMIX-4GL** constructs the character variable **query_1** that contains the conditions for the WHERE clause of a SELECT statement.

3. Concatenates **query_1** to the character string that contains the rest of the SELECT statement and assigns the result to the character variable **s1**.

4. Prepares an executable statement called **s_1** from the SELECT statement stored in **s1**.

5. Declares a cursor for the prepared SELECT statement.

6. Uses a FOREACH loop to display the customer rows on the screen form. The FOREACH loop performs the following operations:

   a. Assigns the value 1 to the local variable **exist** to indicate that at least one customer has been selected.

   b. Displays the customer row on screen.

   c. Asks whether the user wants to see the next customer. If the user enters n in response to the prompt, INFORMIX-4GL exits the FOREACH loop.

   d. Repeats steps a—c until all customer rows have been processed or until the user enters n to exit the FOREACH loop.

7. Displays a message if no customer rows satisfied the search criteria (**exist** = 0).

8. Displays a message if the user wanted to see another customer row (**answer** = y) and INFORMIX-4GL could not find one.

Example 2

The **ch12ord** program retrieves orders based on search criteria that the user specifies and displays each order, along with the associated items, upon request.

```
DATABASE stores

        GLOBALS

        DEFINE

        p_customer  RECORD

                    customer_num LIKE customer.customer_num,
                    fname LIKE customer.fname,
                    lname LIKE customer.lname,
                    address1 LIKE customer.address1,
                    address2 LIKE customer.address2,
                    city LIKE customer.city,
                    state LIKE customer.state,
                    zipcode LIKE customer.zipcode,
                    phone LIKE customer.phone

                    END RECORD,

        p_orders    RECORD

                    order_num LIKE orders.order_num,
                    order_date LIKE orders.order_date,
                    ship_date LIKE orders.ship_date

                    END RECORD,

        p_items     ARRAY[10] OF RECORD

                    item_num LIKE items.item_num,
                    stock_num LIKE items.stock_num,
                    manu_code LIKE items.manu_code,
                    description LIKE stock.description,
                    quantity LIKE items.quantity,
                    unit_price LIKE stock.unit_price,
                    total_price LIKE items.total_price

                    END RECORD

END GLOBALS


MAIN

        OPEN FORM orderform FROM "order"

        DISPLAY FORM orderform

        CALL get_order()

        MESSAGE "End program."

        SLEEP 3

        CLEAR SCREEN

END MAIN
```

```
FUNCTION get_order()

        DEFINE
                query_1, s1     CHAR(800),
                exist           SMALLINT,
                answer          CHAR(1)

        MESSAGE "Enter search criteria for one or more orders."

        SLEEP 3

        MESSAGE ""

        CONSTRUCT query_1 ON customer.customer_num, fname, lname,
                  address1, address2, city, state, zipcode, phone,
                  orders.order_num, order_date, ship_date,
                  items.stock_num, items.manu_code, quantity, total_price
                  FROM customer.*, orders.*, stock_num,
                  manu_code, quantity, total_price

        LET s1 = "SELECT UNIQUE customer.customer_num, fname, lname, ",
                 "address1, address2, city, state, zipcode, phone, ",
                 "orders.order_num, order_date, ship_date ",
                 "FROM customer, orders, items ",
                 "WHERE customer.customer_num = orders.customer_num AND ",
                 "orders.order_num = items.order_num AND ",
                 query_1 CLIPPED,
                 "  ORDER BY customer.customer_num, orders.order_num"

        PREPARE s_1 FROM s1

        DECLARE q_curs CURSOR FOR s_1

        LET exist = 0

        FOREACH q_curs INTO p_customer.*, p_orders.*

                LET exist = 1

                CLEAR FORM

                DISPLAY BY NAME p_customer.*

                DISPLAY BY NAME p_orders.*

                PROMPT "Do you want to view the items for this order (y/n) :
                        FOR answer

                IF answer = "y" THEN
                        CALL get_items()
                END IF

                PROMPT "Do you want to see the next order (y/n) : "
                        FOR answer

                IF answer = "n" THEN
                        EXIT FOREACH
                END IF


        END FOREACH

        IF exist = 0 THEN
                MESSAGE "No rows satisfy the search criteria."

        ELSE

                IF answer = "y" THEN
                        MESSAGE "No more rows satisfy the search criteria."
                END IF

        END IF

        SLEEP 3

END FUNCTION
```

```
FUNCTION get_items()

        DEFINE counter      SMALLINT

        DECLARE my_curs CURSOR FOR
                SELECT item_num, items.stock_num,
                   items.manu_code, description,
                   quantity, unit_price, total_price
                FROM items, stock
                WHERE order_num = p_orders.order_num
                   AND items.stock_num = stock.stock_num
                   AND items.manu_code = stock.manu_code
                ORDER BY item_num

        LET counter = 1

        FOREACH my_curs INTO p_items[counter].*

                LET counter = counter + 1

        END FOREACH


        CALL set_count(counter - 1)

        MESSAGE "Press ESC when you finish viewing the items on order."

        DISPLAY ARRAY p_items TO s_items.*

        MESSAGE ""

END FUNCTION
```

As you can see, this program contains a function called **get_order** that performs the following tasks:

1. Displays a message that prompts the user to query for one or more orders.

2. Uses the CONSTRUCT statement to let the user specify search criteria in all display fields on the **order** form except **item_num** and **description**. This means, for example, that the user can query for orders placed by a given customer, orders placed before a given date, orders that contain a given item, or any combination of these. When the user presses **ESCAPE** after entering his query, **INFORMIX-4GL** constructs the character variable **query_1** that contains the conditions for the WHERE clause of a SELECT statement.

3. Concatenates **query_1** to the character strings that contain the rest of the SELECT statement and assigns the result to the character variable **s1**. Since the user can specify search criteria in display fields associated with the **customer**, **orders**, and **items** tables, the FROM and WHERE clauses of this SELECT statement make reference to all three tables, even though the SELECT clause only

includes columns from **customer** and **orders**. The program displays only customer and order information, unless the user requests to see the items associated with a given order.

4. Prepares an executable statement called **s__1** from the SELECT statement stored in **s1**.

5. Declares a cursor for the prepared SELECT statement.

6. Uses a FOREACH loop to display the orders on the screen. This FOREACH loop is very similar to the one in the previous example except for the inclusion of a new prompt and a conditional call to the function **get__items**. If the user enters y in response to the prompt, INFORMIX-4GL invokes the **get__items** function that performs the following operations:

   a. Declares a cursor for a SELECT statement that retrieves the items for the given order.

   b. Uses a FOREACH statement to store the items for the given order in the program array **p__items**.

   c. Calls the **set__count** function to set the initial value of **arr__count** to the number of rows currently stored in the program array.

   d. Uses the DISPLAY ARRAY statement to let the user view the items for the given order.

7. Displays a message if no orders satisfied the search criteria (**exist** = 0).

8. Displays a message if the user wanted to see another order (**answer** = y) and INFORMIX-4GL could not find one.

Chapter Summary

- You can write programs that retrieve information based on search criteria that the user enters on a screen form.
- The user can use relational, range, and alternation operators, as well as wildcard characters, to perform a query by example.
- You can use the CONSTRUCT statement to create a character variable that contains the conditions for the WHERE clause of a SELECT statement based on the search criteria entered by the user.
- You can create a SELECT statement by concatenating the character variable that contains the WHERE conditions to the character strings that contain the rest of the SELECT statement.
- You can use the PREPARE statement to create an executable statement from the SELECT statement that you construct.
- You can use a cursor with the DECLARE and FOREACH statements to run a prepared SELECT statement that retrieves one or more rows from the database.

Chapter 13

Windows

Chapter 13 Table of Contents

Chapter Overview 5

Window Management 6

Demonstration Programs Containing Windows 9

Using the Order Entry Program 10

Including a Window in a Program 12
- Naming a Window 12
- Establishing the Location of a Window 13
- Specifying the Size of a Window 14
  - Providing Explicit Window Dimensions 14
  - Automatically Sizing a Window to a Form 15
  - Opening a Form in a Window 18
- Indicating Window Attributes 19
  - Default Settings 20
  - Requesting Borders 21
  - Setting the Prompt and Message Lines 23
  - Setting the Form Line 25

Closing a Window 26

Displaying Information in a Window 27
- Displaying a Message 27
- Prompting for Input 29
- Displaying a Form in a Window 30
- Displaying a Menu in a Window 31

Clearing a Window of Text 33

Examining the Order Entry Program 35
- The **wind1.4gl** Program 36
- The **enter_order** Function 44
- The **get_cust** Function 46
- The **show_cust** Function 47
- The **stock_help** Function 51
- The **get_stock** Function 51

Using a Program with Multiple Windows 56

Working with Multiple Windows 60
Indicating the Current Window 60
More About the CLEAR WINDOW Statement 63
More About the CLOSE WINDOW Statement 65

Examining a Program with Multiple Windows 68
The **wind2.4gl** Program 69
The GLOBALS Statement 75
The MAIN Statement 75
The **show_menu** Function 76
The **set_curs** Function 77
The **get_cust** Function 79
The **view_cust** Function 79
The **get_ord** Function 80
The **view_ord** Function 80
The **det_cust** Function 81
The **det_ord** Function 82
The **change_wind** Function 83
The **clear_menu** Function 83
The **mess** Function 83

Chapter Summary 84

Chapter Overview

This chapter explains how to build windows into your INFORMIX-4GL programs and describes the window management statements available in INFORMIX-4GL. The following topics are discussed:

- Including a window in an INFORMIX-4GL program
- Establishing the location, size, and attributes of a window
- Placing text, forms, and menus in a window
- Clearing a window of text
- Closing a window
- Indicating the current window
- Working with multiple windows

The examples in this chapter are based on the following programs and forms included with the demonstration database:

wind1.4gl
wind2.4gl
order.per
cust.per
stock1.per
custcur.per
ordcur.per

For complete information about the commands and statements used to work with windows, see Chapter 7, "INFORMIX-4GL Statement Syntax," in Volume 2 of the *INFORMIX-SQL Reference Manual.*

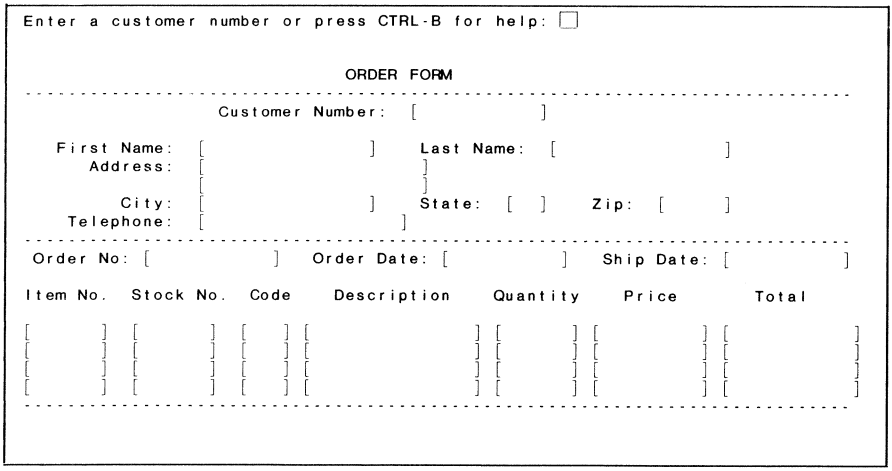
Window Management

The terminal screen can be considered the *window* in which a program displays information. In your application programs you can devote different rectangular parts of the terminal screen to different activities. This flexibility lets you display each INFORMIX-4GL activity in a different window.

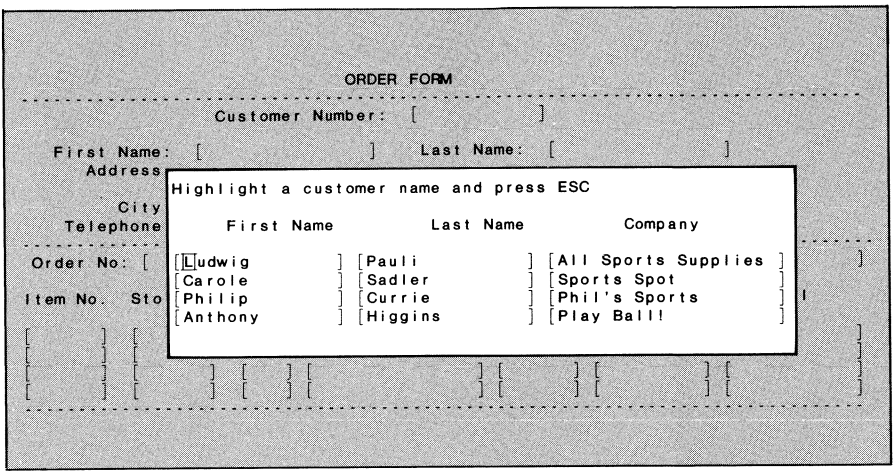
Until now the INFORMIX-4GL programs with which you have worked have used the entire screen. This chapter describes how to include one or more windows in your programs.

You can create applications that include windows for screen forms, displays, prompts, menus, or report output. For example, the following five screens are produced by an order entry program that uses two windows to display helpful information. The program that generates these windows is included with the demonstration application and is discussed later in this chapter.

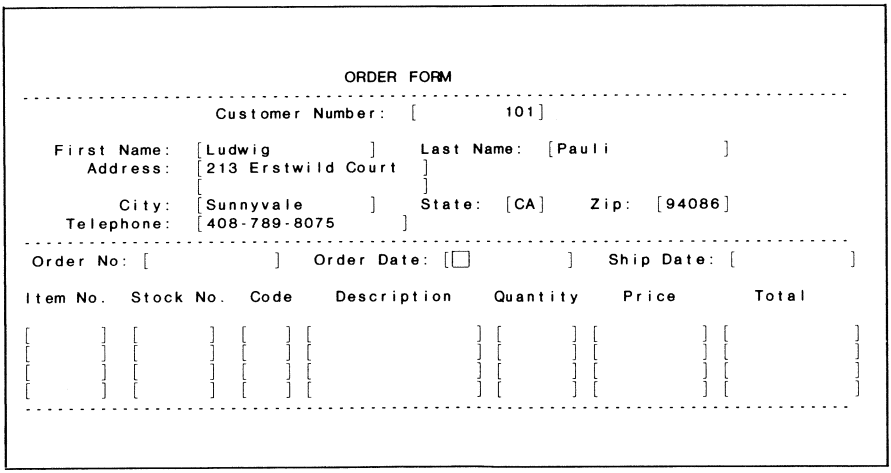
The following screen shows the **order** form as it first appears to the user.



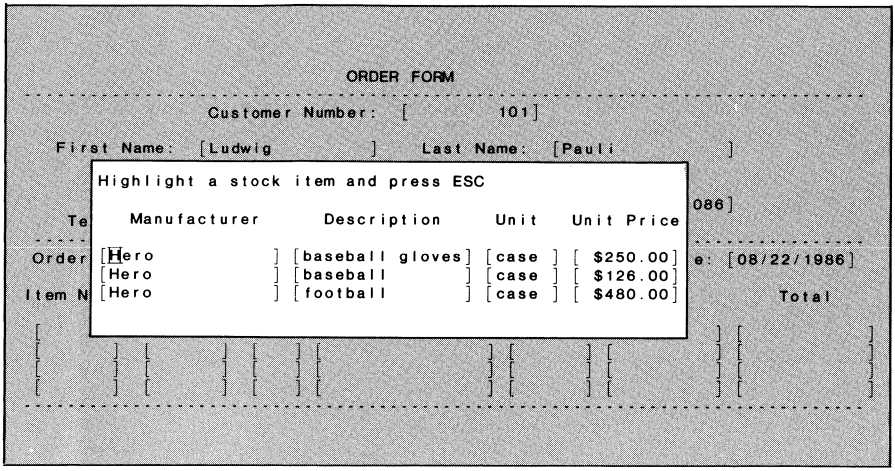
If the user presses **CONTROL-B** when the cursor is in the Customer Number field, a window appears in the middle of the screen. The program automatically selects all rows in the **customer** table and displays them in a screen array in the window.



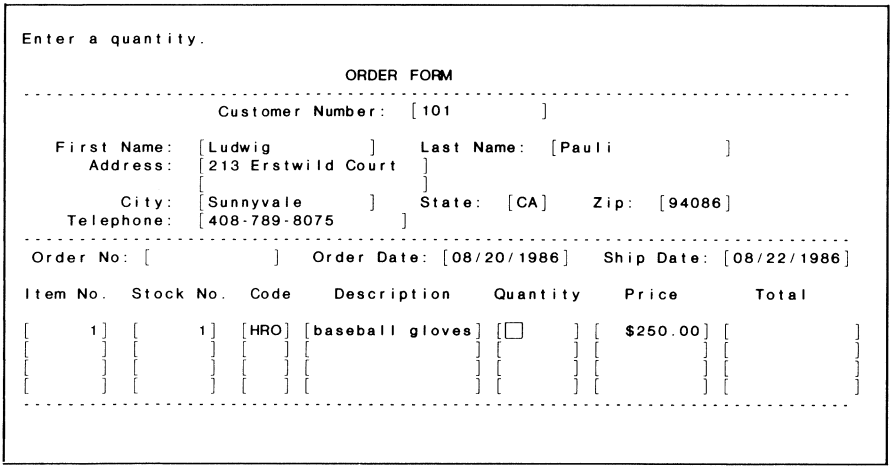
The user selects the desired customer by positioning the cursor on the row and pressing the **ESCAPE** key. Customer information is then displayed on the **order** form in the customer information fields.



A window is also available to help with entering the items that make up a customer's order.



The user performs a query by example to select the rows displayed in this screen array. The user selects the desired item by positioning the cursor on the row and pressing the **ESCAPE** key. The item information then appears on the order entry form in the next available row of the screen array.



Demonstration Programs Containing Windows

The demonstration application supplied with INFORMIX-4GL includes two programs that use windows.

wind1.4gl An order entry program. The **order** screens displayed in the previous section are produced by this program.

wind2.4gl A multiple-window program that demonstrates how to work with two windows.

Each program is examined in later sections in this chapter.

Using the Order Entry Program

If you have not already done so, you should compile and use the **wind1.4gl** program. Consider how the user accesses and uses the two windows as you work with the program.

Several of the characteristics of the program are noted here. The statements required to write a program with windows are described in the next section.

- The user can select a customer in two ways:

  1. Enter the customer's unique number at the prompt. With the selection, the necessary customer information is automatically entered into the appropriate fields on the form.

  2. Press CONTROL-B to display the customer information window. The user selects a customer row by highlighting a customer name and pressing the ESCAPE key. Once the user selects a customer, the window is closed and disappears from the screen. With the selection, the necessary customer information is automatically entered into the appropriate fields on the form.

- The screen array at the bottom portion of the form is used to enter the items that make up an order. Again, the user can enter an item in one of two ways:

  1. Enter a stock number and manufacturer code. INFORMIX-4GL automatically displays the description and unit price for the item identified by the stock number and manufacturer code.

  2. Press CONTROL-B to display the stock information window. The user performs a query by example by entering search criteria in the display fields in the first row of the screen array. INFORMIX-4GL then displays items that satisfy the search criteria in the screen array. The user scrolls through the array until the desired item appears, and selects an item by highlighting the row and pressing the ESCAPE key. Once the user selects an item, the window closes. INFORMIX-4GL inserts the stock number, manufacturer code, description, and unit price for the item into the appropriate fields in the item array.

- The user enters a quantity and the program computes and displays the total price for that item. The cursor moves to the next row of the screen array.

- When the user presses the ESCAPE key, the order is entered into the database, the order number appears in the appropriate field on the form, and the program ends.

The next sections describe the statements you can use to create an INFORMIX-4GL program that displays windows. Many of the examples refer to the **wind1.4gl** and **wind2.4gl** programs included with the demonstration database.

Including a Window in a Program

You use the OPEN WINDOW statement to open a window in an **INFORMIX-4GL** program. You indicate the name, location, and dimensions of a window in the OPEN WINDOW statement. In addition, you can optionally display a border around the window, display the window in reverse video or color, and change the values for the prompt, message, form, and comment lines by including an ATTRIBUTE clause in the OPEN WINDOW statement.

The complete syntax of an OPEN WINDOW statement follows:

OPEN WINDOW *window-name* AT *row, column*
WITH {*integer* ROWS, *integer* COLUMNS |
FORM "*form-file*"}
[ATTRIBUTE (*attribute-list*)]

The structure of the OPEN WINDOW statement is described in the following sections of this chapter.

Naming a Window

The OPEN WINDOW statement attaches a name to the window. For example, the following statement is part of the first OPEN WINDOW statement appearing in the **wind1.4gl** program. This statement opens the **cwindo** (customer information) window:

```
OPEN WINDOW cwindo              # partial statement
```

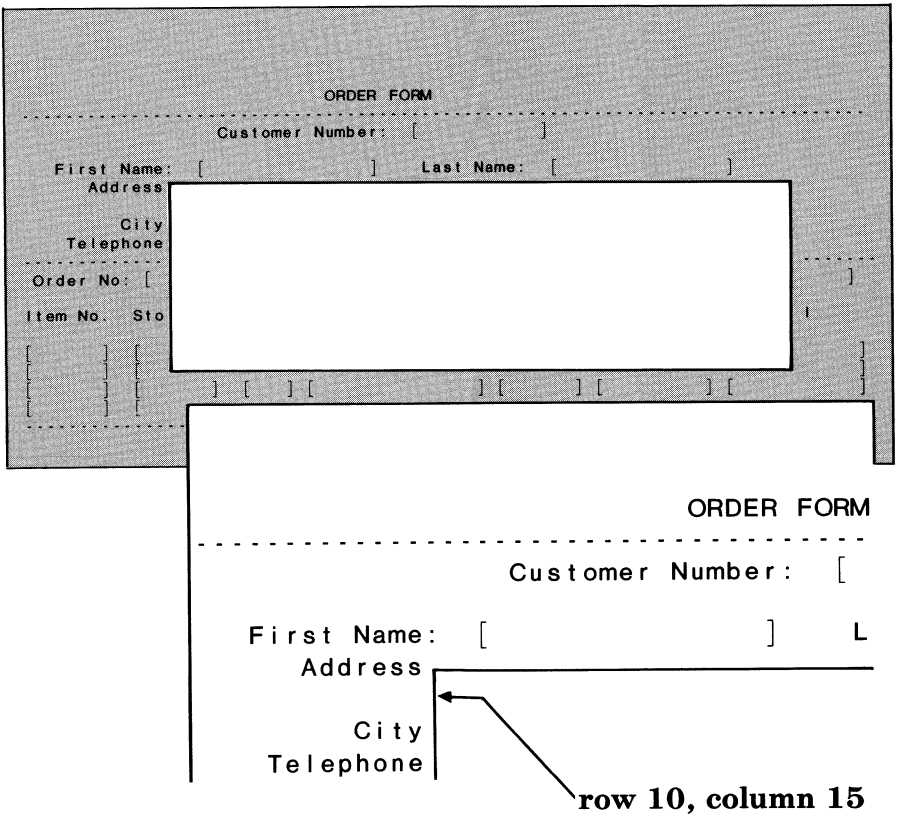
A window name must be a valid **INFORMIX-4GL** identifier and must be unique within a program.

Establishing the Location of a Window

The OPEN WINDOW statement includes an AT clause indicating the originating location for the window. For example, the OPEN WINDOW statement used to open the **cwindo** window is

```
OPEN WINDOW cwindo
   AT 10, 15                    # partial statement
```

The first row of the **cwindo** window appears at row 10 on the full screen, and the first column of the window appears at column 15. The following screen illustrates how the window is positioned on the screen.



The location of a window is always relative to the full screen. If you open a second window while displaying a first, the coordinates used to locate the second window are relative to the screen and not the first window. See the section "Working with Multiple Windows" for more information about programs that display more than one window.

Specifying the Size of a Window

You can include explicit dimensions for the window in the OPEN WINDOW statement, or let INFORMIX-4GL automatically determine the required width and length of a window sized to fit a form. You can also open a window with explicit dimensions and open a form within the window.

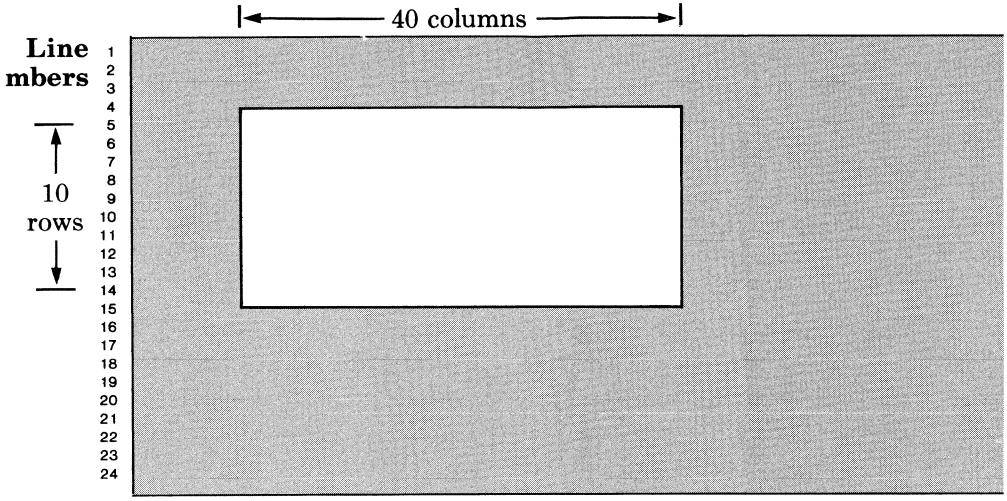
Providing Explicit Window Dimensions

You specify the dimensions of a window by including a WITH *integer* ROWS, *integer* COLUMNS clause in the OPEN WINDOW statement.

For example, the following OPEN WINDOW statement opens the **w1** window at screen location 5, 10 and creates a bordered window that is 10 rows long and 40 columns wide.

```
OPEN WINDOW w1 AT 5, 10
   WITH 10 ROWS, 40 COLUMNS
   ATTRIBUTE (BORDER)
```

The window opened with this OPEN WINDOW statement appears in the next screen. (The ATTRIBUTE clause and the BORDER keyword are described in the section "Indicating Window Attributes" later in this chapter).



Automatically Sizing a Window to a Form

Your OPEN WINDOW statement can include a WITH FORM clause in place of a WITH *integer* ROWS, *integer* COLUMNS clause.

For example, the **wind1.4gl** program automatically opens a window large enough to display the **cust** form.

```
OPEN WINDOW cwindo AT 10, 15
   WITH FORM "cust"
   ATTRIBUTE (BORDER)
```

When you open a window WITH FORM *form-name*, INFORMIX-4GL also opens and displays the form. You cannot include OPEN FORM, DISPLAY FORM, and CLOSE FORM statements that refer to the form used in this OPEN WINDOW statement. INFORMIX-4GL closes the form automatically when the window is closed. See the section "Closing a Window" later in this chapter for more information.

In determining the window size for a form, INFORMIX-4GL performs the following actions:

- Truncates any trailing blank spaces appearing at the right of the form. INFORMIX-4GL draws the right margin of the window at the first column following the last character on the form.

- Draws the left margin of the window before any leading blank spaces. If the left margin of the form includes one or more blank spaces, these spaces appear in the window.

- Draws the top line of the window two lines above the top line of the form. By default, the first two lines of the window are reserved for prompt and message text produced by your INFORMIX-4GL program. (The section "Indicating Window Attributes" includes information about adjusting these settings.)

- Draws the bottom line of the window one line below the last line of the form. By default, the last line of the window is reserved for comment text produced by the COMMENTS attribute. (The section "Indicating Window Attributes" includes information about adjusting this setting.)

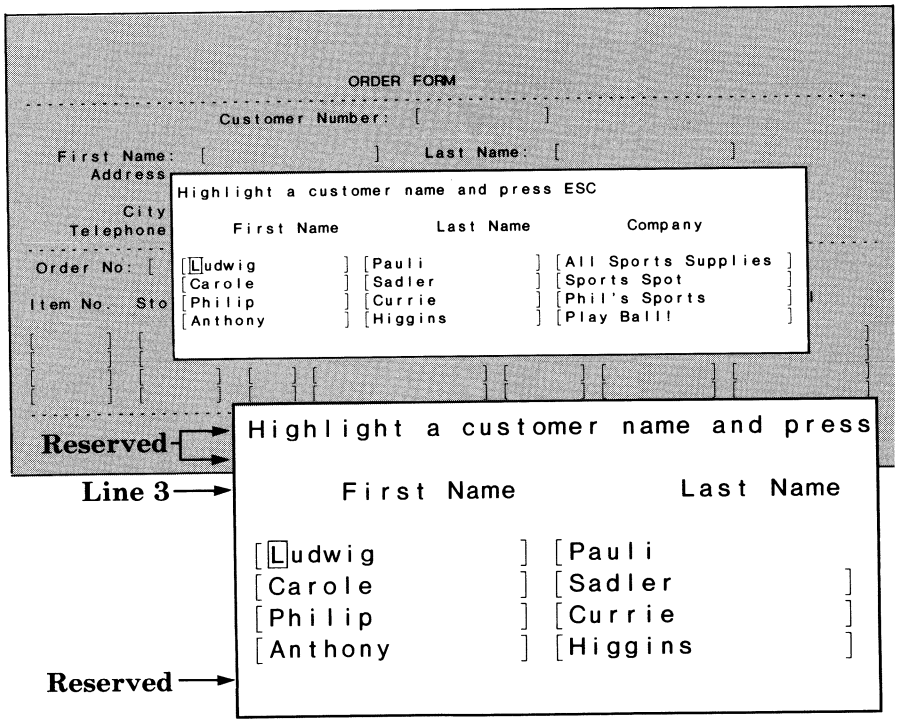
The next two screens illustrate how INFORMIX-4GL calculates the size of the **cwindo** window and displays the **cust** form.

The screen section of the **cust** form specification appears below:

```
screen
{
     First Name          Last Name          Company

[f001            ] [f002            ] [f003                ]
[f001            ] [f002            ] [f003                ]
[f001            ] [f002            ] [f003                ]
[f001            ] [f002            ] [f003                ]
}
```

The next screen illustrates how INFORMIX-4GL sizes the **cwindo** window to accommodate the **cust** form.



Opening a Form in a Window

If you want a window larger than the one that **INFORMIX-4GL** automatically creates for a form, or want to display a series of forms in the window, you can include a WITH *integer* ROWS, *integer* COLUMNS clause in the OPEN WINDOW statement. You would then include OPEN FORM, DISPLAY FORM, and CLOSE FORM statements after the OPEN WINDOW statement in your program.

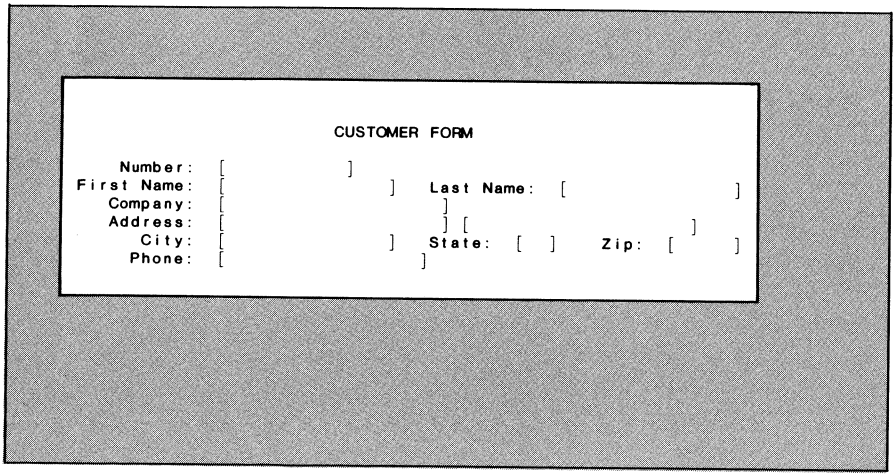
The following example shows a portion of an **INFORMIX-4GL** program that opens a window with defined dimensions. The window displays the **custcur** form.

```
. . .
OPEN WINDOW cw1 AT 5, 5
   WITH 11 ROWS, 63 COLUMNS
   ATTRIBUTE (BORDER)

OPEN FORM custcur FROM "custcur"

DISPLAY FORM custcur
. . .
```

The next screen appears when **INFORMIX-4GL** executes the above statements.



INFORMIX-4GL displays a runtime error if you do not allow sufficient rows and columns in your window to display the form. By default, INFORMIX-4GL reserves three rows (one prompt line, one message line, and one comment line) in addition to the number of rows required to display the form. The default form line is line three of the window.

You can reduce the number of rows INFORMIX-4GL uses by defining the form line as line one or two and changing the prompt and message lines accordingly. The minimum number of rows required to display a form in a window is the number of rows in the form plus an additional line below the form. The next section describes how to make changes like these.

Indicating Window Attributes

You can specify a border, reverse video, or foreground color for a window, and change the values for the prompt, message, comment, and form lines, by including an ATTRIBUTE clause in the OPEN WINDOW statement.

This chapter describes the use of the BORDER, PROMPT LINE, MESSAGE LINE, and FORM LINE keywords in the ATTRIBUTE clause. Specifying color attributes, reverse video display, and an alternate comment line is described in Chapter 7, "INFORMIX-4GL Statement Syntax," of the *INFORMIX-4GL Reference Manual.*

Default Settings

Following are the attributes that you can include in an ATTRIBUTE clause along with their default values:

Attribute	Default
BORDER	no border
color	The default foreground color on your terminal
REVERSE	no reverse video
PROMPT LINE *line-value*	FIRST
MESSAGE LINE *line-value*	FIRST + 1
FORM LINE *line-value*	FIRST + 2
COMMENT LINE *line-value*	LAST - 1 (for the screen) LAST (for all windows except the screen)

For the prompt, message, and comment lines, *line-value* can be an integer, a program variable that evaluates to an integer, FIRST plus an optional integer, or LAST minus an optional integer. For the form line, *line-value* can be an integer, a program variable that evaluates to an integer, or FIRST plus an optional integer.

INFORMIX-4GL uses the values specified in the most recently executed OPTIONS statement to establish the prompt, message, comment, and form lines. You can change any of these values in the ATTRIBUTE clause of the OPEN WINDOW statement. If you change the line values in an ATTRIBUTE clause, they apply only to the newly opened window.

Requesting Borders

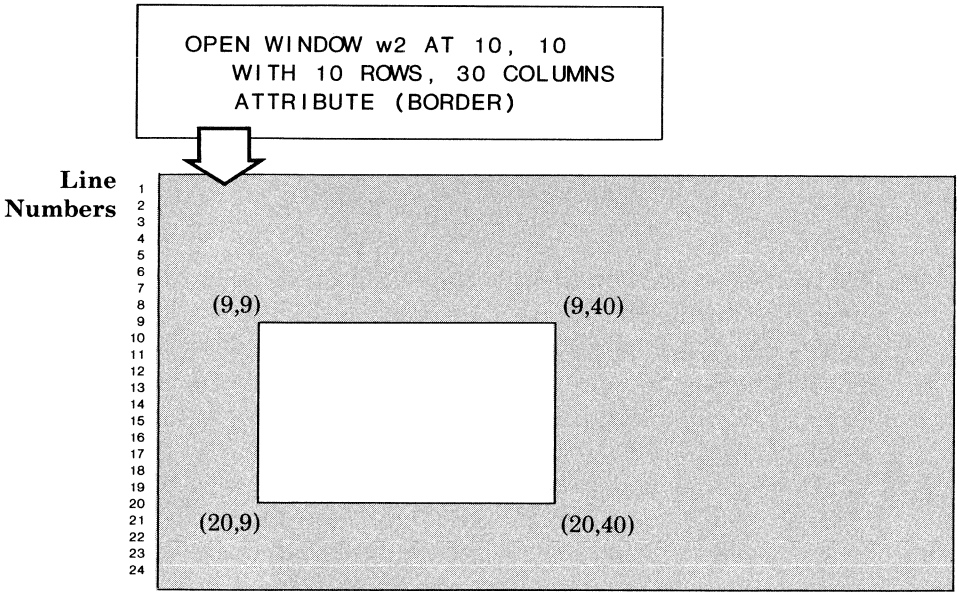
Use the BORDER attribute if you want INFORMIX-4GL to draw a border around your window. In most instances you will want a window to appear with a border, so that the text displayed in the window is visually distinct from the background text.

The following OPEN WINDOW statement opens the bordered **w2** window at row position 10, column position 10 on the screen. The **w2** window is 10 rows long and 30 columns wide.

```
OPEN WINDOW w2 AT 10, 10
    WITH 10 ROWS, 30 COLUMNS
    ATTRIBUTE (BORDER)
```

On DOS systems, INFORMIX-4GL uses graphics characters to draw the border. On UNIX systems, INFORMIX-4GL uses characters defined in the system **termcap** file to draw the border. If you choose, you can specify alternate border characters in this file. If no characters are defined, INFORMIX-4GL uses the hyphen (-) for horizontal lines, the vertical bar (|) for vertical lines, and the plus sign (+) for corners. See Appendix N, "Termcap Changes for INFORMIX-4GL," in the *INFORMIX-4GL Reference Manual* and the manual that comes with your terminal for information about making changes to your **termcap** file.

INFORMIX-4GL draws the border outside the window area you specify in the WITH clause of the OPEN WINDOW statement. The following screen displays the window created with the previous OPEN WINDOW statement. The coordinates of the bordered window are indicated in the screen, along with the OPEN WINDOW statement that creates the window.



INFORMIX-4GL uses one column to the left and one column to the right of the window to display the vertical lines, and a row above the top line and below the bottom line of the window to display the horizontal lines.

Note: The **termcap** entry for some terminals includes an sg#1 setting. On these terminals INFORMIX-4GL reserves an additional column to the left and to the right of the window. These columns are reserved whether or not you specify a border. INFORMIX-4GL uses a total of four extra columns for bordered windows on these terminals: two columns to the left of the window and two columns to the right of the window.

If a window and its border (if any) exceed the physical limits of the screen, INFORMIX-4GL generates a runtime error.

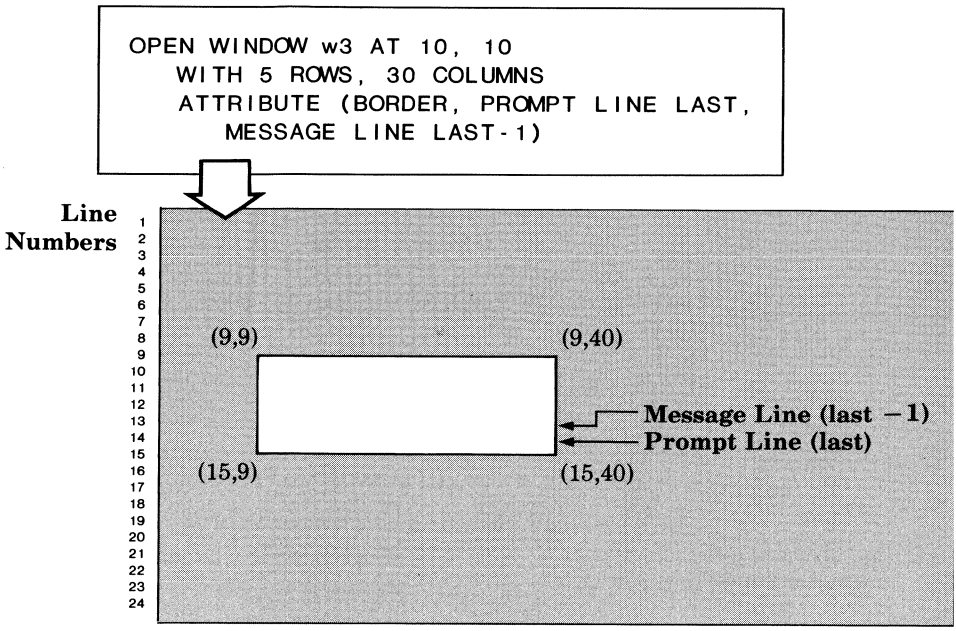
Setting the Prompt and Message Lines

You use the OPTIONS statement in your INFORMIX-4GL program to make global settings that remain in effect throughout a program. Chapter 8, "Menus, Options, and Help," describes how to use the OPTIONS statement to change the default positions of the message and prompt lines on the full screen. You use the ATTRIBUTE clause in the OPEN WINDOW statement to change settings only within a single window.

For example, the following OPEN WINDOW statement opens a bordered window and assigns the prompt line to LAST and the message line to LAST - 1.

```
OPEN WINDOW w3 at 10, 10
    WITH 5 ROWS, 30 COLUMNS
    ATTRIBUTE (BORDER, PROMPT LINE LAST,
        MESSAGE LINE LAST-1)
```

The **w3** window appears (with annotation) in the next screen.



Settings made in an ATTRIBUTE clause are not global to the INFORMIX-4GL program, nor are they passed to subsequent windows. The location of all other message, prompt, comment, and form lines is governed by the settings made in the most recently executed OPTIONS statement.

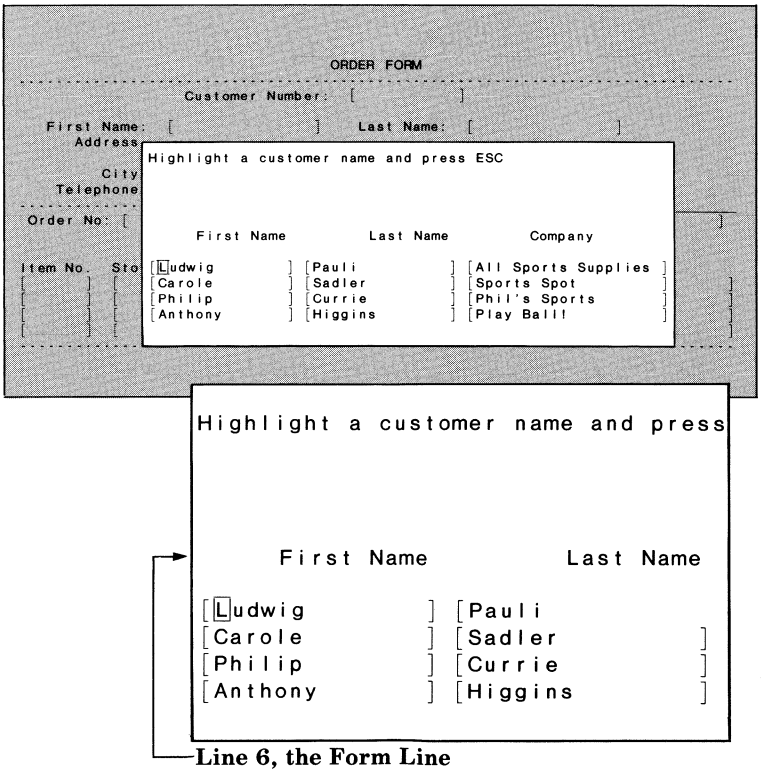
Note: Since the error line is relative to the screen rather than the current window, you cannot change its location in the ATTRIBUTE clause of an OPEN WINDOW statement. Changing the error line is described in Chapter 7, "INFORMIX-4GL Statement Syntax," of the *INFORMIX-4GL Reference Manual.*

Setting the Form Line

By default, the top row of a form appears on line 3 of the window. You change this line by including the FORM LINE *n* keywords in the ATTRIBUTE clause of the OPEN WINDOW statement.

For example, the following OPEN WINDOW statement opens the bordered **cwindo1** window and displays the top row of the **cust** form on line 6 of the window. The **cwindo1** window appears in the next screen.

```
OPEN WINDOW cwindo1 AT 10, 15
    WITH FORM "cust"
    ATTRIBUTE (BORDER, FORM LINE 6,
        MESSAGE LINE FIRST)
```



Closing a Window

You use the CLOSE WINDOW statement to close a window. For example, the following statement appears in the **wind1.4gl** program and closes the **cwindo** window:

```
CLOSE WINDOW cwindo
```

When **INFORMIX-4GL** executes the CLOSE WINDOW statement it removes the **cwindo** window from the screen and restores the underlying display.

```
                              ORDER FORM
--------------------------------------------------------------------------------
                    Customer Number:  [         101]

    First Name:  [Ludwig          ]     Last Name:  [Pauli           ]
       Address:  [213 Erstwild Court  ]
                 [                     ]
          City:  [Sunnyvale       ]     State:  [CA]     Zip:  [94086]
     Telephone:  [408-789-8075       ]
--------------------------------------------------------------------------------
 Order No: [           ]   Order Date: [           ]   Ship Date: [           ]

 Item No.   Stock No.   Code     Description     Quantity     Price        Total

 [       ]  [       ]  [    ]  [                ]  [       ]  [          ]  [          ]
 [       ]  [       ]  [    ]  [                ]  [       ]  [          ]  [          ]
 [       ]  [       ]  [    ]  [                ]  [       ]  [          ]  [          ]
 [       ]  [       ]  [    ]  [                ]  [       ]  [          ]  [          ]
--------------------------------------------------------------------------------
```

When you close a window, **INFORMIX-4GL** frees all resources used by the window, including forms, and restores the underlying display. Closing the window does not affect the value of variables set while the window was open.

Displaying Information in a Window

Your INFORMIX-4GL applications can include windows that display screen forms, help information, prompts, menus, or report output. The two windows in the **wind1.4gl** program display forms that are used to scroll through arrays of customer and stock information. The windows in the **wind2.4gl** program display both forms and menus.

Displaying a Message

You may want to use a window to display a screen message. The following example demonstrates how to display a reminder in a window:

```
. . .
INPUT p_orders.order_date, p_orders.ship_date
  FROM orders.order_date, orders.ship_date

  AFTER FIELD ship_date

      OPEN WINDOW w4 AT 10, 10
          WITH 3 ROWS, 45 COLUMNS
          ATTRIBUTE (BORDER)

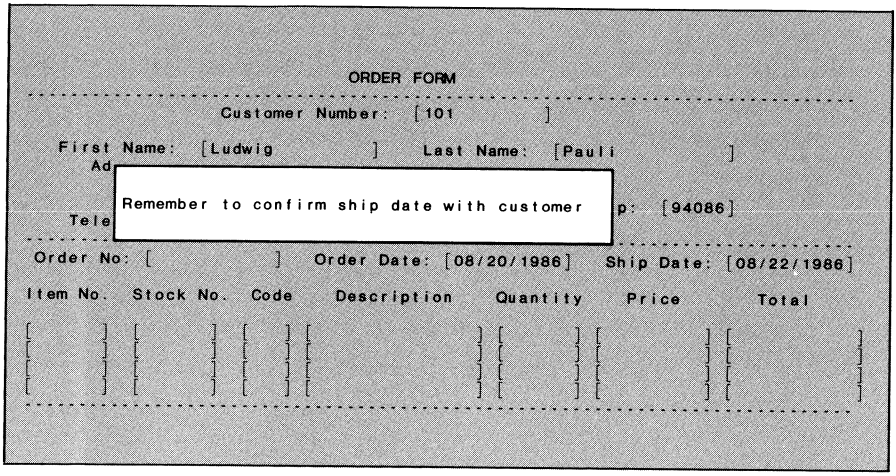
      MESSAGE "Remember to confirm ship date ",
          "with customer"

      SLEEP 3

      CLOSE WINDOW w4

END INPUT
. . .
```

The bordered **w4** window appears on the screen at the designated coordinates when the cursor leaves the **ship_date** field:



The window appears for three seconds, disappears, and the underlying screen is restored.

Prompting for Input

This example shows how to prompt for input in a bordered window.

```
. . .
DISPLAY BY NAME p_customer.*

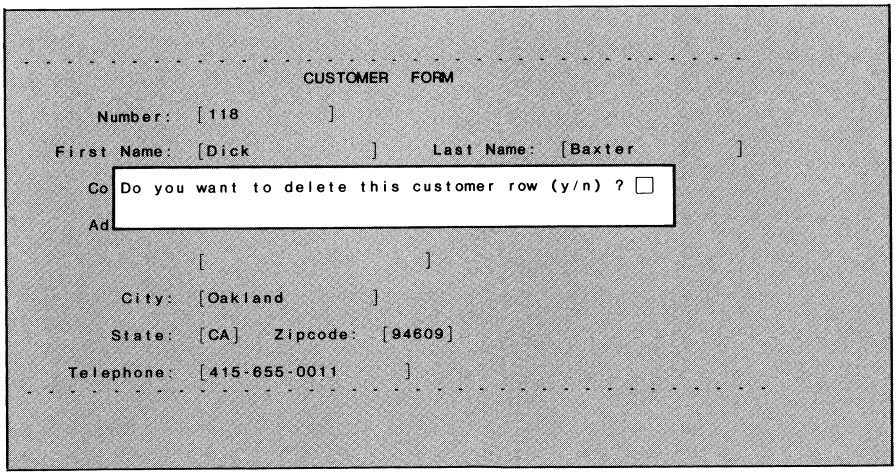
OPEN WINDOW w5 AT 10, 10
    WITH 2 ROWS, 50 COLUMNS
    ATTRIBUTE (BORDER)

PROMPT "Do you want to delete ",
       "this customer row (y/n) ? "
    FOR answer

CLOSE WINDOW w5

. . .
```

The **w5** window appears in the following screen.



Displaying a Form in a Window

The next example shows how to let the user enter customer data on a screen form displayed in a bordered window.

```
. . .
OPEN WINDOW w6 AT 3, 5
     WITH FORM "custcur"
     ATTRIBUTE (BORDER, MESSAGE LINE FIRST)

MESSAGE "Enter information about a customer"

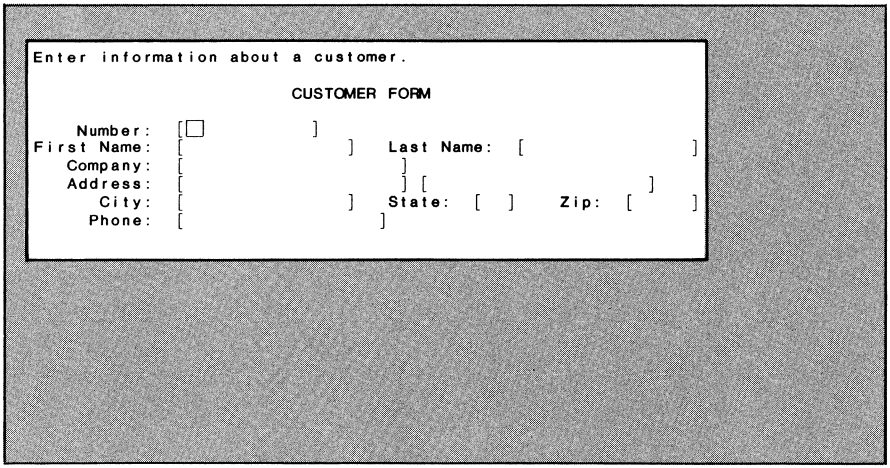
INPUT BY NAME p_customer.*

LET p_customer.customer_num = 0

INSERT INTO customer VALUES (p_customer.*)

CLOSE WINDOW w6
. . .
```

The bordered **w6** window appears in the following screen.



Displaying a Menu in a Window

Displaying a menu in an INFORMIX-4GL window is easily accomplished. This example shows how to open a bordered window that displays a menu. The program takes the user's menu choice and assigns the appropriate value to the **response** variable.

```
. . .
OPEN WINDOW w7 AT 14, 6
    WITH 2 ROWS, 45 COLUMNS
    ATTRIBUTE (BORDER)

MENU "Employee Type"

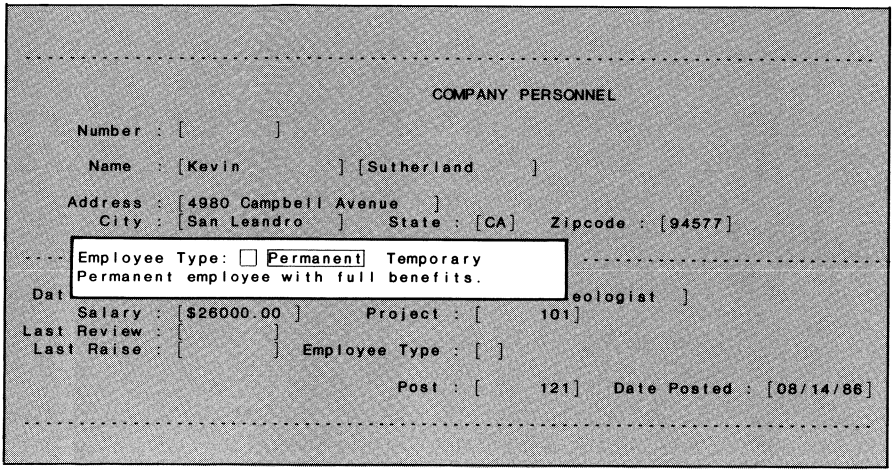
    COMMAND "Permanent"
        "Permanent employee with full benefits."
        LET response = "p"
        EXIT MENU

    COMMAND "Temporary"
        "Temporary employee with no benefits."
        LET response = "t"
        EXIT MENU

END MENU

CLOSE WINDOW w7
. . .
```

The **w7** window appears in the next screen.

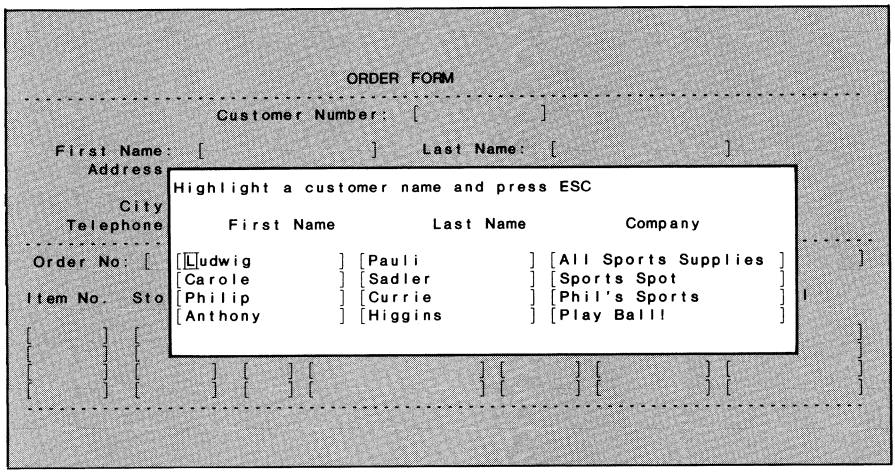


More sophisticated window menus appear in the **wind2.4gl** program. The **wind2.4gl** program is described in a later section in this chapter.

Clearing a Window of Text

You can use the CLEAR statement with the WINDOW keyword to clear the display within a window. The CLEAR WINDOW statement does not affect the value of variables; it simply clears the window of all text, retaining any borders.

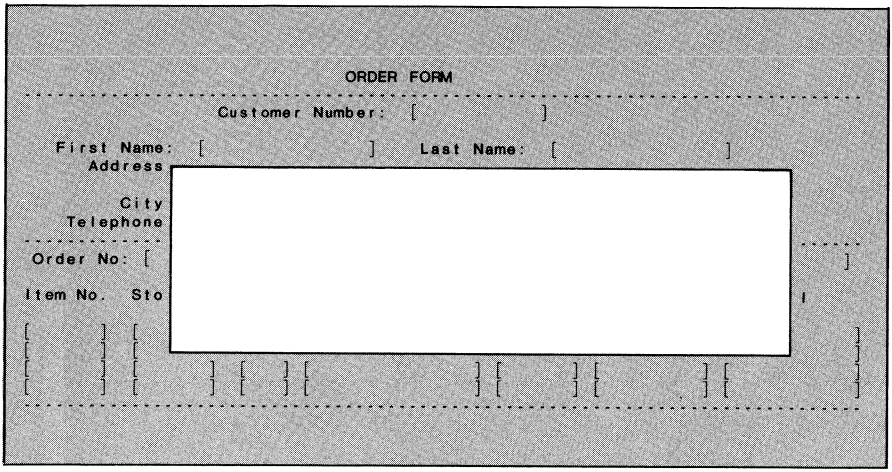
The examples in this section use the screen display from the **wind1.4gl** program. The **cwindo** window appears in the following screen.



You execute the following statement to clear the **cwindo** window of text:

```
CLEAR WINDOW cwindo
```

The next screen illustrates the effect of the CLEAR WINDOW statement.



Examining the Order Entry Program

You now have most of the tools needed to write programs that display information in windows. This section describes the **wind1.4gl** program that is used to enter new customer orders into the database.

The **wind1.4gl** program is a more sophisticated version of the order entry portion of the **ordmenu** program described in Chapter 11. With little work you could combine the two programs into a more elegant program for entering and updating customer orders.

The functions containing the window statements are described in this section. These functions are as follows:

enter_order	get_cust
stock_help	show_cust
get_stock	

The following functions and program blocks are not discussed here because they are very similar to material in the **ordmenu** program. Most of these functions are described in Chapter 11.

get_item	renum_items
get_total	insert_items
get_item_num	mess

```
DATABASE stores

#       The GLOBALS statement defines two program records
#       and a program array used throughout the program.

GLOBALS

        DEFINE

        p_customer RECORD
                    customer_num LIKE customer.customer_num,
                    fname LIKE customer.fname,
                    lname LIKE customer.lname,
                    address1 LIKE customer.address1,
                    address2 LIKE customer.address2,
                    city LIKE customer.city,
                    state LIKE customer.state,
                    zipcode LIKE customer.zipcode,
                    phone LIKE customer.phone

                    END RECORD,

        p_orders RECORD

              order_num LIKE orders.order_num,
              order_date LIKE orders.order_date,
              ship_date LIKE orders.ship_date

              END RECORD,

        p_items ARRAY[10] OF RECORD

              item_num LIKE items.item_num,
              stock_num LIKE items.stock_num,
              manu_code LIKE items.manu_code,
              description LIKE stock.description,
              quantity LIKE items.quantity,
              unit_price LIKE stock.unit_price,
              total_price LIKE items.total_price

              END RECORD

END GLOBALS

#       The MAIN statement opens the order form and displays
#       the form on the screen.  It calls the enter_order function
#       that allows the user to enter information about a
#       customer order into the database.

MAIN

        OPEN FORM orderform FROM "order"

        DISPLAY FORM orderform

        CALL enter_order()

        CALL mess("End program.")

        CLEAR SCREEN

END MAIN
```

```
#       The enter_order function allows the user to enter an
#       order for a customer on the order form, inserts the
#       order into the stores database, and displays the
#       order number for the new order.

FUNCTION enter_order()

        DEFINE pa_curr,sc_curr, stock  SMALLINT,
               m_code CHAR(3)

        CALL get_cust()

        CALL mess("Enter an order date and a ship date.")

        INPUT p_orders.order_date, p_orders.ship_date
              FROM order_date, ship_date

        CALL mess("Enter up to 10 items.")

        INPUT ARRAY p_items FROM s_items.*

              BEFORE FIELD stock_num
              MESSAGE "Enter a stock number or press CTRL-B for help."

              BEFORE FIELD manu_code
              MESSAGE "Enter the manufacturer code or ",
                    "press CTRL-B for help."

              BEFORE FIELD quantity
                    MESSAGE "Enter a quantity."

              ON KEY (CONTROL-B)
                    IF infield(stock_num) OR infield(manu_code) THEN
                          CALL stock_help()
                          NEXT FIELD quantity
                    END IF

              AFTER FIELD stock_num, manu_code
                    MESSAGE ""

                    LET pa_curr = arr_curr()

                    IF    p_items[pa_curr].stock_num IS NOT NULL
                          AND p_items[pa_curr].manu_code IS NOT NULL
                    THEN
                          CALL get_item()

                          IF p_items[pa_curr].quantity IS NOT NULL THEN

                             CALL get_total()

                          END IF

                    END IF

              AFTER FIELD quantity
                    MESSAGE ""

                    LET pa_curr = arr_curr()

                    IF p_items[pa_curr].stock_num IS NOT NULL
                          AND p_items[pa_curr].manu_code IS NOT NULL
                          AND p_items[pa_curr].quantity IS NOT NULL
                    THEN
                          CALL get_total()

                    END IF
```

```
            BEFORE INSERT
                  CALL get_item_num()

            AFTER INSERT
                  CALL renum_items()

            AFTER DELETE
                  CALL renum_items()

      END INPUT

      INSERT INTO orders (order_num, order_date,
            customer_num, ship_date)
      VALUES (0, p_orders.order_date,
            p_customer.customer_num,
            p_orders.ship_date)

      LET p_orders.order_num = SQLCA.SQLERRD[2]

      DISPLAY p_orders.order_num TO order_num

      CALL insert_items()

      CALL mess("Order added.")

END FUNCTION

#     The get_cust function retrieves customer information from the
#     stores database and displays it on the screen form.

FUNCTION get_cust()

      WHILE TRUE

            PROMPT "Enter a customer number or press CTRL-B for help:  "
                  FOR p_customer.customer_num
                  ON KEY (CONTROL-B)
                        LET p_customer.customer_num = show_cust()
            END PROMPT

            SELECT fname, lname, address1, address2,
                  city, state, zipcode, phone
                  INTO p_customer.fname THRU p_customer.phone
                  FROM customer
                  WHERE customer_num = p_customer.customer_num

            IF status = 0 THEN

                  EXIT WHILE

            ELSE
                  CALL mess("No customer with that customer number.")

            END IF

      END WHILE

      DISPLAY p_customer.* TO customer.*

END FUNCTION
```

```
#         The show_cust function opens a window, allows the user to
#         select a customer, and returns the customer number to the
#         get_cust function.

FUNCTION show_cust()

        DEFINE p_cust ARRAY[25] OF RECORD
               fname LIKE customer.fname,
               lname LIKE customer.lname,
               company LIKE customer.company
               END RECORD,
               p_custnum ARRAY[25] OF integer,
               counter SMALLINT

        OPEN WINDOW cwindo AT 10,15
               WITH FORM "cust"
               ATTRIBUTE (BORDER, MESSAGE LINE FIRST)

        DECLARE cust_list CURSOR FOR
               SELECT fname, lname, company, customer_num
                      FROM customer

        LET counter = 1

        FOREACH cust_list
               INTO p_cust[counter].*, p_custnum[counter]
               LET counter = counter + 1
               IF counter > 25 THEN
                      EXIT FOREACH
               END IF
        END FOREACH

        CALL set_count(counter -1)

        MESSAGE "Highlight a customer name and press ESC"

        DISPLAY ARRAY p_cust TO s_cust.*

        LET counter = arr_curr()

        CLOSE WINDOW cwindo

        RETURN p_custnum[counter]

END FUNCTION



#         The stock_help function is called when the cursor is in the
#         stock_num or manu_code field and the user presses CONTROL-B.

FUNCTION stock_help()

        DEFINE pa_curr, sc_curr SMALLINT
        LET sc_curr = scr_line()
        LET pa_curr = arr_curr()
        CALL get_stock() RETURNING
               p_items[pa_curr].stock_num,
               p_items[pa_curr].manu_code,
               p_items[pa_curr].description,
               p_items[pa_curr].unit_price
        DISPLAY p_items[pa_curr].stock_num,
               p_items[pa_curr].manu_code,
               p_items[pa_curr].description,
               p_items[pa_curr].unit_price
        TO s_items[sc_curr].stock_num,
               s_items[sc_curr].manu_code,
               s_items[sc_curr].description,
               s_items[sc_curr].unit_price

END FUNCTION
```

```
#         The get_stock function opens a window, allows the user to
#         query the database for information about stock items, and
#         returns information about a selected item to the stock_help function.

FUNCTION get_stock()

        DEFINE p_stock ARRAY[25] OF RECORD
               manu_name LIKE manufact.manu_name,
               description LIKE stock.description,
               unit LIKE stock.unit,
               unit_price LIKE stock.unit_price
               END RECORD,

               p_sn_mc ARRAY[25] OF RECORD
               stock_num LIKE stock.stock_num,
               manu_code LIKE stock.manu_code
               END RECORD,

               counter SMALLINT,
               query_1, sel_stmt CHAR(500)

        OPEN WINDOW w AT 10,8
               WITH FORM "stock1"
               ATTRIBUTE (BORDER, MESSAGE LINE FIRST)




        WHILE TRUE

               MESSAGE "Enter search criteria for one or more items."

               CONSTRUCT query_1 ON
                  manu_name, description, unit, unit_price
                  FROM s_stock[1].*

               LET sel_stmt =
                  "SELECT manu_name, description, unit, unit_price, ",
                     "stock_num, stock.manu_code, ",
                     "FROM stock, manufact ",
                     "WHERE stock.manu_code = manufact.manu_code AND ",
                        query_1 CLIPPED

               PREPARE s1 FROM sel_stmt

               DECLARE stock_list CURSOR FOR s1

               LET counter = 1

               FOREACH stock_list
                      INTO p_stock[counter].*, p_sn_mc[counter].*
                      LET counter = counter + 1
                      IF counter > 25 THEN
                            EXIT FOREACH
                      END IF
               END FOREACH

               IF counter = 1 THEN

                     MESSAGE "No items satisfy the search criteria."
                     SLEEP 3
               ELSE
                     EXIT WHILE

               END IF

        END WHILE
```

```
        CALL set_count(counter -1)

        MESSAGE "Highlight a stock item and press ESC."

        DISPLAY ARRAY p_stock TO s_stock.*

        LET counter = arr_curr()

        CLOSE WINDOW w

        RETURN p_sn_mc[counter].stock_num,
               p_sn_mc[counter].manu_code,
               p_stock[counter].description,
               p_stock[counter].unit_price

END FUNCTION



#       The get_item function selects the description and unit price
#       for an item from the database, enters this information into
#       the program array, and displays the information on the
#       screen form.

FUNCTION get_item()

        DEFINE pa_curr, sc_curr SMALLINT

        LET pa_curr = arr_curr()
        LET sc_curr = scr_line()

        SELECT description, unit_price
               INTO  p_items[pa_curr].description,
                     p_items[pa_curr].unit_price
               FROM  stock
               WHERE   stock.stock_num = p_items[pa_curr].stock_num
                     AND stock.manu_code = p_items[pa_curr].manu_code

        DISPLAY p_items[pa_curr].description,
                     p_items[pa_curr].unit_price
               TO s_items[sc_curr].description,
                     s_items[sc_curr].unit_price

END FUNCTION


#       The get_total function computes the total price for an item
#       and displays this value on the form.

FUNCTION get_total()

        DEFINE pa_curr, sc_curr SMALLINT

        LET pa_curr = arr_curr()

        LET sc_curr = scr_line()

        LET p_items[pa_curr].total_price =
              p_items[pa_curr].quantity * p_items[pa_curr].unit_price

        DISPLAY p_items[pa_curr].total_price
              TO s_items[sc_curr].total_price

END FUNCTION
```

```
#         The get_item_num function displays the item number of the
#         current program array row before the user begins entering
#         information.

FUNCTION get_item_num()

         DEFINE pa_curr, sc_curr SMALLINT

         LET pa_curr = arr_curr()

         LET sc_curr = scr_line()

         LET p_items[pa_curr].item_num = pa_curr

         DISPLAY pa_curr TO s_items[sc_curr].item_num

END FUNCTION


#         The renum_items function renumbers the items in the
#         screen array and program array after the user inserts
#         or deletes a row.

FUNCTION renum_items()

         DEFINE pa_curr,
                pa_total,
                sc_curr,
                sc_total,
                k       SMALLINT

         LET pa_curr = arr_curr()

         LET pa_total = arr_count()

         LET sc_curr = scr_line()

         LET sc_total = 4

         FOR k = pa_curr TO pa_total

               LET p_items[k].item_num = k

               IF sc_curr <= sc_total THEN

                     DISPLAY k TO s_items[sc_curr].item_num

                     LET sc_curr = sc_curr + 1

               END IF

         END FOR

END FUNCTION
```

```
#         The insert_items function inserts the items in the program array
#         into the database.

FUNCTION insert_items()

         DEFINE counter SMALLINT

         FOR counter = 1 TO arr_count()

               INSERT INTO items
                     VALUES (p_items[counter].item_num,
                           p_orders.order_num,
                           p_items[counter].stock_num,
                           p_items[counter].manu_code,
                           p_items[counter].quantity,
                           p_items[counter].total_price)

         END FOR

END FUNCTION

#         The mess function displays a message on line 23 of the screen.
#         Other functions in the program pass character strings
#         to the mess function.

FUNCTION mess(str)

         DEFINE str CHAR(50)

         DISPLAY str CLIPPED AT 23,1

         SLEEP 3

         DISPLAY "" AT 23, 1

END FUNCTION
```

The **enter_order** *Function*

The **enter_order** function in the **wind1.4gl** program performs many of the same tasks as the function of the same name in the **ordmenu** program. This function allows the user to enter an order for a customer on the **order** form, inserts the order into the **stores** database, and displays the order number for the new order. This **enter_order** function differs from the previous one in that it allows the user to call functions that display windows with customer and item information.

The **enter_order** function performs the following operations:

1. Defines four local variables for use in the function.

2. Calls the **get_cust** function (described in the next section). The **get_cust** function enters customer information on the form.

3. Displays the following message on the screen:

   ```
   Enter an order date and a ship date
   ```

4. Allows the user to enter order and shipping dates into the form and stores these values in the **p_orders.order_date** and **p_orders.ship_date** program variables.

5. Uses an INPUT ARRAY statement to assign values to the **p_items** program array from data entered into the **s_items** screen array. Comments on selected clauses in the INPUT ARRAY statement appear next:

   - Three BEFORE FIELD clauses display messages on the screen.

   - An ON KEY clause calls the **stock_help** function when the user presses CONTROL-B and the cursor is in fields **stock_num** or **manu_code**.

The **stock_help** function displays items in a screen array that appears in a window. The user selects an item displayed in the screen array and information about this item is entered on the form.

The **infield** function appears in the ON KEY clause. **infield** is an INFORMIX-4GL function that tracks the movement of the cursor on a form and executes a command when the cursor appears in the designated field.

When the **stock_help** function is completed the cursor moves to the NEXT FIELD (**quantity**).

The NEXT FIELD statement must appear in the ON KEY clause so that values displayed in the **stock_num** and **manu_code** fields and assigned to the corresponding program variables during the ON KEY clause are not lost. Whenever a user triggers an ON KEY clause, INFORMIX-4GL preserves the values in fields listed in the INPUT or INPUT ARRAY statement in an input buffer. The values in the input buffer are normally restored after the program executes all statements in the ON KEY clause. If you direct the cursor to a new field with the NEXT FIELD statement, you prevent the INPUT ARRAY statement from ignoring the new value(s) assigned as the program executes the ON KEY clause.

- The program uses two AFTER FIELD clauses to look up description and unit price information, and to calculate the total price of a given item. These and the remaining clauses are described in more detail in Chapter 11.

6. Inserts a new row into the **orders** table using the values in the appropriate program variables.

7. Assigns the value in SQLCA.SQLERRD[2] to **p_orders.order_num.** (SQLCA.SQLERRD[2] contains the serial number of the row that INFORMIX-4GL just inserted into the **orders** table. See Chapter 2 of the

INFORMIX-4GL Reference Manual for information about the SQLCA record.)

8. Displays this number in the **order_num** field on the screen form.

9. Calls the **insert_items** function that inserts the items in the program array into the **items** table.

10. Displays the message Order added on the screen.

The **get_cust** *Function*

The **get_cust** function retrieves customer information from the **stores** database and displays it on the screen form. The **get_cust** function performs the following operations:

1. Uses a WHILE loop to select a customer based on a customer number. The number can be entered by the user or returned by the **show_cust** function described in the next section. The WHILE loop performs the following operations:

   a. Prompts the user for a customer number and enters the number into the **p_customer.customer_num** program variable. The prompt message indicates that the user can also . . . press CTRL-B for help.

      When the user presses CONTROL-B, the program sets the value for the **customer_num** field to the value returned by the **show_cust** function.

   b. Performs a SELECT and attempts to retrieve a customer row from the database using the value in the **p_customer.customer_num** variable to identify a unique row. Values returned from the SELECT are stored in the **p_customer** record.

c. Tests whether the SELECT returned a row. If a row is returned, the program exits the WHILE loop. If no row is returned, the following message appears on the screen:

```
No customer with that customer number.
```

Steps a through c are repeated.

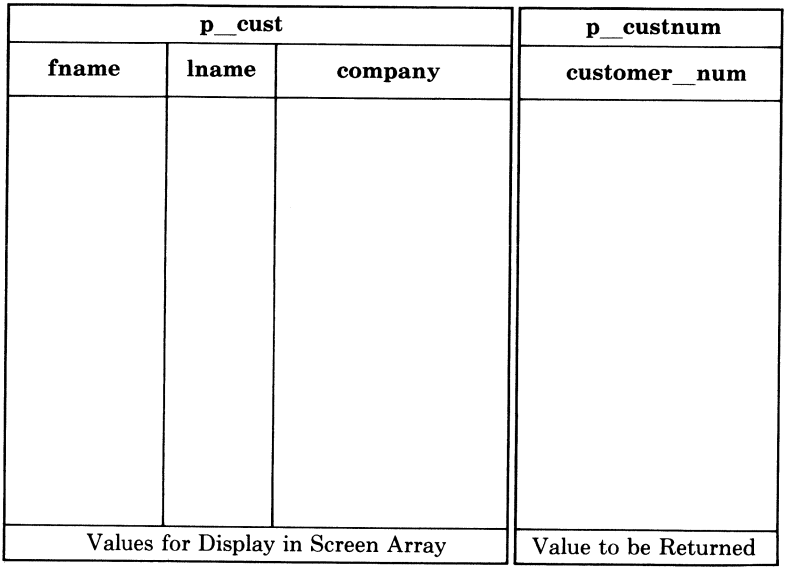
2. Uses a DISPLAY statement to display the values in the **p_customer** record to the appropriate fields in the screen form.

The **show_cust** *Function*

The **show_cust** function opens a window, allows the user to select a customer, and returns the customer number to the **get_cust** function. The **show_cust** function performs the following operations:

1. DEFINEs the **p_cust** and **p_custnum** program arrays. Each array holds customer information. The SELECT statement (described below) returns the values stored in each array. The **p_cust** array stores the **fname**, **lname**, and **company** values returned by the query. The program displays these values in the screen array. The **p_custnum** array stores the **customer_num** values returned by the query. These values are not displayed in the screen array. This array is required so that a single **customer_num** value can be returned to the **get_cust** function.

The following figure illustrates the structure of the two arrays.



2. OPENs the **cwindo** window and displays the **cust** screen form in the newly opened window.

3. DECLAREs a cursor for a SELECT statement that retrieves rows with first and last name, company name, and customer number information from the **customer** table.

4. LETs the value of the local variable **counter** equal 1. The **counter** variable serves as an index to the rows in the two program arrays.

5. Uses a FOREACH statement to fetch rows from the **customer** table. The **p_cust** array stores the **fname**, **lname**, and **company** values; the **p_custnum** array stores the **customer_num** values.

The following figure illustrates how the two program arrays store the first six rows.

p_cust			p_custnum
fname	**lname**	**company**	**customer_num**
Ludwig	Pauli	All Sports Supplies	101
Carole	Sadler	Sports Spot	102
Philip	Currie	Phil's Sports	103
Anthony	Higgins	Play Ball!	104
Raymond	Vector	Los Altos Sports	105
George	Watson	Watson & Son	106
Values for Display in Screen Array			Value to be Returned

The program increments the **counter** variable as each row is fetched. The program arrays were defined to hold a maximum of 25 rows; the program detects if a 26th row is fetched and exits the FOREACH loop before attempting to store the 26th row in the program arrays.

6. Calls the INFORMIX-4GL **set_count** function. This function sets the value in the **arr_count** function to the total number of rows currently stored in the program array (the value of the **counter** variable less 1). This step is required before executing a DISPLAY ARRAY statement.

7. Displays the following message on the screen:

   ```
   Highlight a customer name and press ESC
   ```

8. Uses a DISPLAY ARRAY statement to let the user view the list of customers in the **p_cust** program array.

9. LETs the value of the local variable **counter** equal the value in the **arr_curr** variable. **arr_curr** returns the number of the current row in both program arrays.

10. Closes the **cwindo** window.

11. RETURNs a customer number to the calling program, using the value in the **counter** variable to identify the appropriate row (the current row) in the **p_custnum** program array.

    For example, if the user highlights the third row of the **p_cust** program array, the function returns the customer number in the third row of the **p_custnum** program array. The following figure illustrates how the **show_cust** function returns the appropriate value to the calling program (the **get_cust** function).



p_cust			p_custnum
fname	**lname**	**company**	**customer_num**
Ludwig	Pauli	All Sports Supplies	101
Carole	Sadler	Sports Spot	102
Philip	Currie	Phil's Sports	103
Anthony	Higgins	Play Ball!	104
Raymond	Vector	Los Altos Sports	105
George	Watson	Watson & Son	106
Values for Display in Screen Array			Value to be Returned

The **stock__help** *Function*

The program calls the **stock__help** function when the cursor is in the **stock__num** or **manu__code** fields and the user presses CONTROL-B. The **stock__help** function performs the following operations:

1. Defines and assigns values to two local variables used to keep track of the current lines in both the program and screen arrays.

2. Calls the **get__stock** function. This function returns values that are assigned to the **stock__num**, **manu__code**, **description**, and **unit__price** components in the current row of the **p__items** program array. (The next section describes the **get__stock** function.)

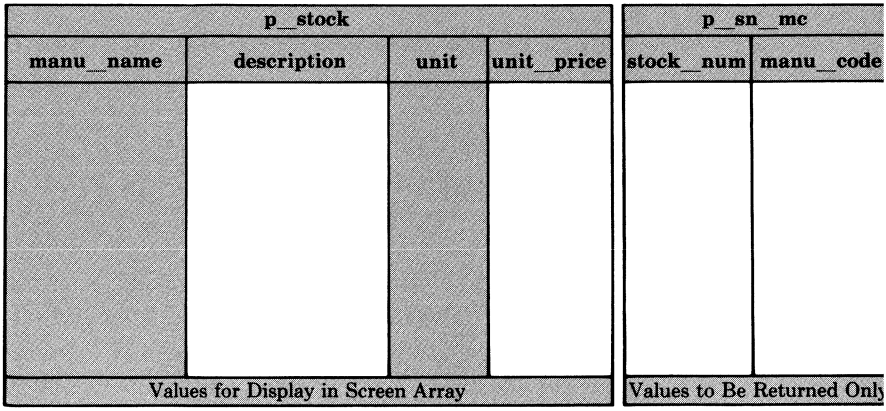
3. Displays the values in the current row of the **p__items** program array in the **s__items** screen array at the bottom of the form.

The **get__stock** *Function*

The **get__stock** function opens a window, allows the user to query the database for information about stock items, and returns information about a selected item to the **stock__help** function. The **get__stock** function performs the following operations:

1. Defines two program arrays. The program uses these arrays in a manner similar to the two program arrays created in the **show__cust** function. The **p__stock** and **p__sn__mc** arrays hold the rows returned from a database query of stock items. The **p__stock** array stores the **manu__name**, **description**, **unit**, and **unit__price** values. The **p__sn__mc** array stores the **stock__num** and **manu__code** values.

The following tables illustrate the structure of the two program arrays.



(Set of Values to Be Returned)

2. Opens the **w** window and displays the **stock1** screen form.

3. Uses a WHILE loop to allow the user to enter search parameters on the **stock1** screen form. The WHILE loop lets the user perform several searches, if necessary, to locate information about the desired item(s). The WHILE loop performs the following operations:

   a. Prompts the user to enter search criteria for a query of the **stock** and **manufact** tables.

   b. Uses a CONSTRUCT statement to let the user specify search criteria for a manufacturer name, item description, unit, and unit price. When the user presses the ESCAPE key, INFORMIX-4GL constructs the **query_1** variable. **query_1** contains part of the conditions for the WHERE clause of the SELECT statement that follows.

c. Uses a LET statement to concatenate a character string containing part of the SELECT statement for retrieving item information to the character variable **query_1**. Note that the SELECT clause includes six columns; four will be stored in the **p_stock** array for display on the screen, and two will be stored in the **p_sn_mc** array.

d. PREPAREs the SELECT statement and DECLAREs a cursor.

e. Sets the initial value of the **counter** variable to 1. The **counter** variable serves as an index to the rows in the two program arrays.

f. Uses a FOREACH loop to fetch SELECTed rows from the **stock** and **manufact** tables and places the row(s) into the two program arrays. The **p_stock** array stores the **manu_name**, **description**, **unit**, and **unit_price** values; the **p_sn_mc** array stores the **stock_num** and **manu_code** values.

The following figure illustrates how the two program arrays store the first five rows that satisfy a query by example.

p_stock				p_sn_mc	
manu_name	description	unit	unit_price	stock_num	manu_code
Hero	baseball gloves	case	250.00	1	HRO
Hero	baseball	case	126.00	2	HRO
Husky	baseball gloves	case	800.00	1	HSK
Husky	baseball bat	case	240.00	1	HSK
Smith	baseball gloves	case	450.00	1	SMT
Values for Display in Screen Array				Values to Be Returned Only	

(Set of Values to Be Returned)

The program increments the **counter** variable as each row is fetched. The program arrays were defined to hold a maximum of 25 rows; the program detects if a 26th row is fetched and exits the FOREACH loop before attempting to store that row in the program arrays.

g. Tests whether a row was fetched and displays the following message if no row is found:

```
No items satisfy the search criteria.
```

The program repeats steps "a" through "g" if no row was fetched.

4. Calls the **set_count** function and sets the value in the **arr_count** function to the total number of rows currently stored in the program array. This step is required before executing a DISPLAY ARRAY statement.

5. Displays the following message on the screen:

```
Highlight a stock item and press ESC
```

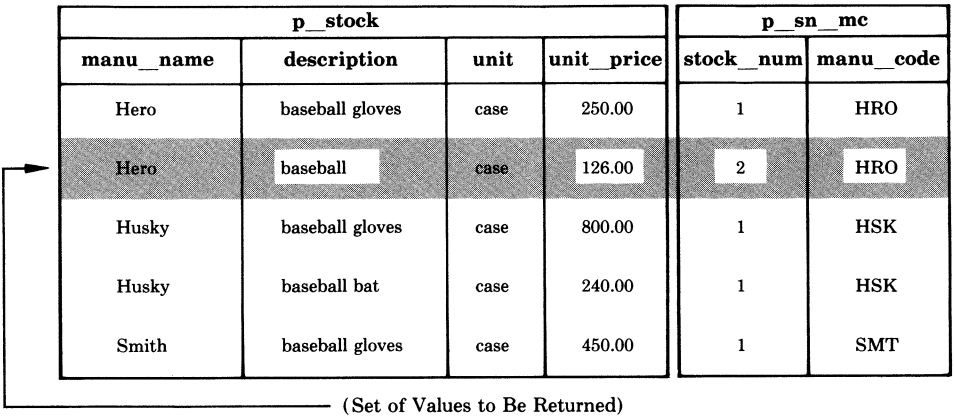
6. Uses a DISPLAY ARRAY statement to let the user view items in the **p_stock** array.

7. LETs the value in the **counter** variable equal the value returned by the **arr_curr** function. The **arr_curr** function returns the number of the current row in both program arrays.

8. Closes the **w** window.

9. RETURNs a stock number, manufacturer code, description, and unit price to the calling function, using the value in the **counter** program to identify the appropriate row.

For example, if the user highlights the second row of the **p_stock** program array, the function returns the stock number and manufacturer code in the second row of the **p_sn_mc** program array as well as the description and unit price in the second row of the **p_stock** program array. The following figure illustrates how **INFORMIX-4GL** returns the appropriate values to the calling program (the **stock_help** function).



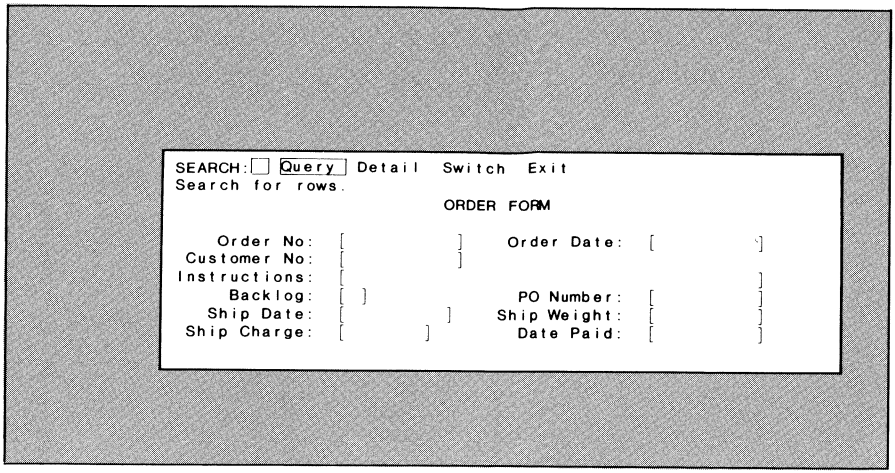
Using a Program with Multiple Windows

The **wind1.4gl** program demonstrates the actions of the OPEN WINDOW and CLOSE WINDOW statements. The two windows in the program appear on the screen at different times. The program does not have to manage the display of multiple windows. You may want your INFORMIX-4GL programs to display more than one window on the screen at the same time.

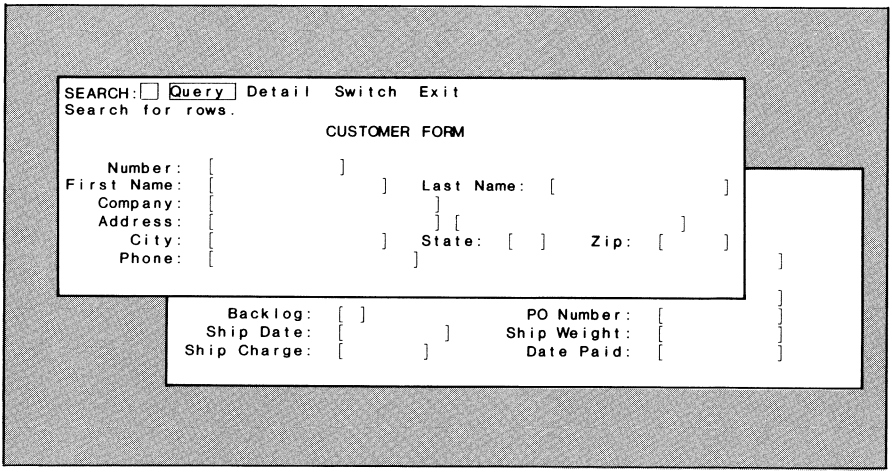
This section describes the window management statements used in multi-window programs. The **wind2.4gl** program in the demonstration application is an example of an INFORMIX-4GL program containing multiple windows. If you have not already done so, you should compile and use the **wind2.4gl** program. As you work with the program, consider how the two windows remain on the screen for the duration of the program, how the user moves from one window to the other, and how each window overwrites the other at the appropriate time.

Selected characteristics of the program are noted here. The final section of this chapter contains a detailed description of the **wind2.4gl** program.

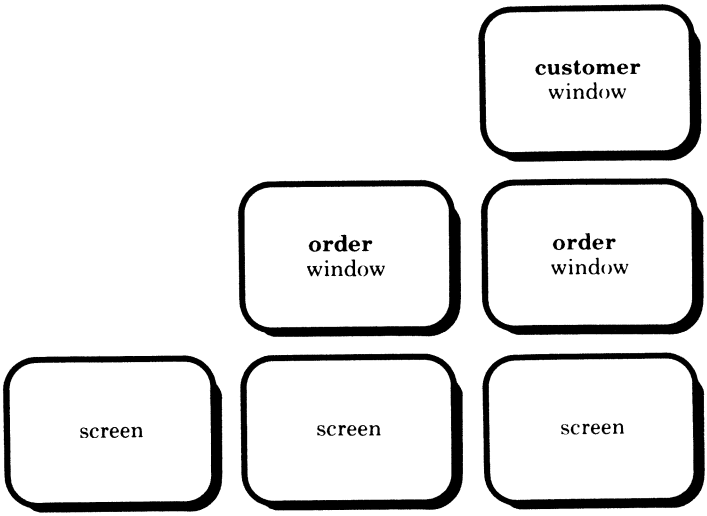
- The program opens and displays two windows during the entire program: one window displays an order form, while the second window displays a customer form. The order window appears first on the screen.



The customer window next appears and overwrites a portion of the order window.



The "stacking" of the two windows is illustrated in the following display:



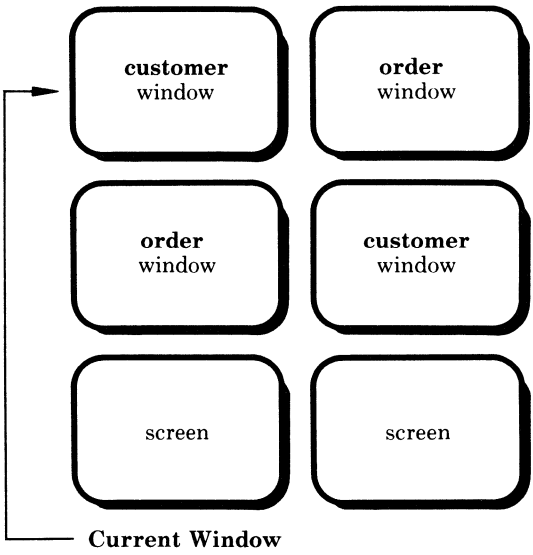
The orders window is the *current window* when the program begins. The window is completely visible for a brief moment. The program then opens the customer window and this window becomes the current window. The customer window is completely visible and overwrites a portion of the orders window.

- The **Query** option on the SEARCH Menu allows the user to perform a query by example on the form displayed in the current window. The user can query for customer information (when the program displays the customer window) or order information (when the program displays the order window).

- The **Detail** option changes the current window and performs a query using the value in the **customer_num** program variable to select customer or order information.

  Once the user finishes viewing the row(s) retrieved by the query, the program restores the original window.

- The **Switch** option on the SEARCH Menu selects a new current window. The shifting of the window stack is illustrated in the following display:



- The current window is always fully visible. The user interacts only with the current window. The program applies the options on the SEARCH Menu only to the form displayed in the current window. For example, the **Query** option queries the **customer** table when the customer window is the current window. The **Query** option queries the **orders** table when the orders window is the current window.

The next section describes the INFORMIX-4GL window management statements used to write a program containing multiple windows.

Working with Multiple Windows

When working with multiple windows, INFORMIX-4GL maintains a list or stack of all open windows. Whenever you OPEN a new window, INFORMIX-4GL makes it the current window by adding it to the top of the window list. When you CLOSE a window, INFORMIX-4GL removes it from its window list. The next window in its list becomes the current window.

This section describes how the INFORMIX-4GL window management statements are used in a multi-window environment.

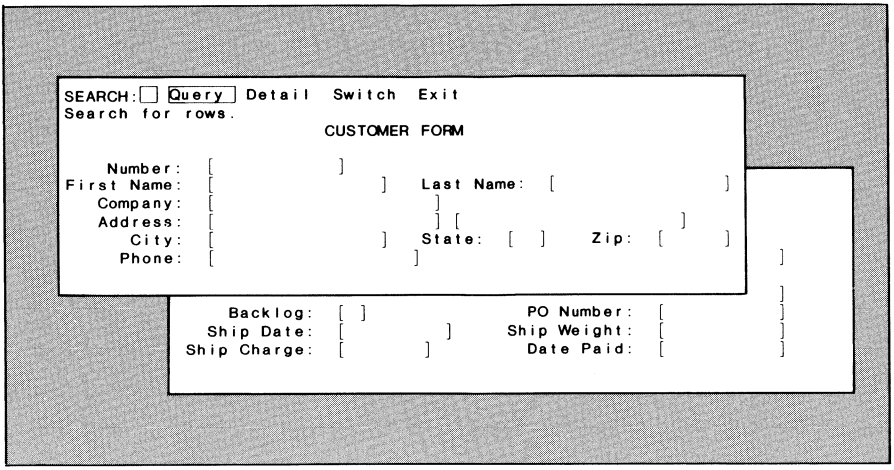
Indicating the Current Window

In programs with multiple windows you may need to select a different current window so that input and output occurs in the appropriate window. You use the CURRENT WINDOW statement to make a window the current and topmost window.

```
CURRENT WINDOW IS window
```

For example, the **wind2.4gl** program displays two windows. The **custcur** form appears in the **cw1** window, and the **ordcur** form appears in the **ow2** window.

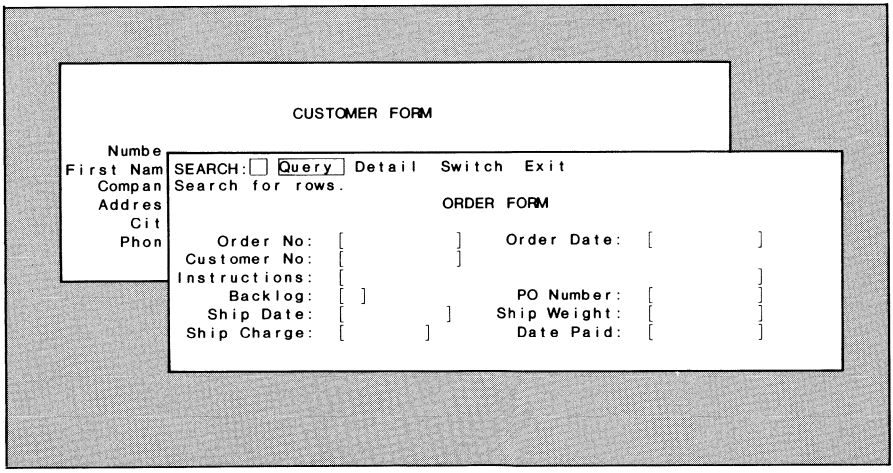
The **cw1** window is the current window and is completely visible in the following screen.



The following statement designates the **ow2** window as the current window.

```
CURRENT WINDOW IS ow2
```

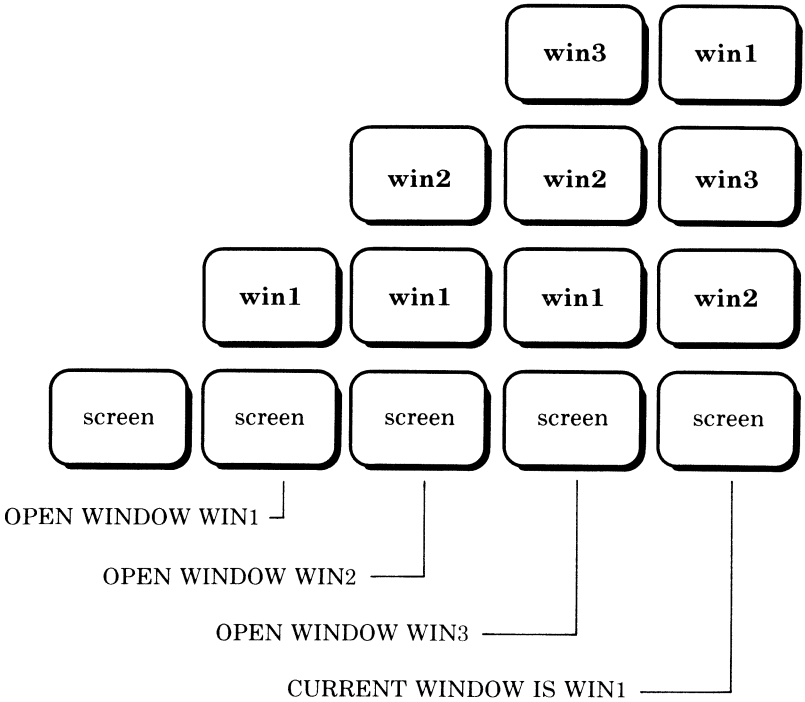
As the current window, the full **ow2** window is completely visible on the screen. The **ow2** window overwrites much of the text of the **cw1** window.



The terminal screen is the current window when a program starts; the name of this window is *screen*.

When working with multiple windows, **INFORMIX-4GL** keeps a record of the order in which the windows are displayed on the screen. When you specify a current window, **INFORMIX-4GL** adjusts its window list by moving the new current window to the top of the list; the other windows in the list retain their relative positions in the list.

The shifting of the window stack as a result of executing OPEN WINDOW and CURRENT WINDOW statements is illustrated in the following display:



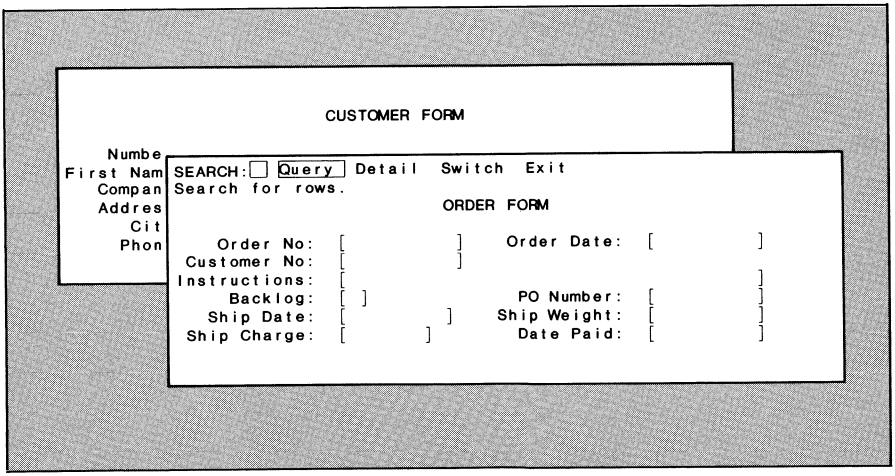
Statements that require input, such as INPUT or MENU, run in the current window. When you change the current window while one of these statements is active and then resume the statement, the original window is restored as the current window.

For example, you can use an ON KEY clause in an INPUT statement to allow the user to open a new window by pressing a specific key during input. When the user presses the designated key, INFORMIX-4GL executes the statements in the ON KEY clause. At the completion of the statements in the ON KEY clause, INFORMIX-4GL resumes input from the window that was current before the ON KEY break.

The context of each window includes the values for the prompt, message, form, and comment lines. When a window becomes the current window, these values are restored.

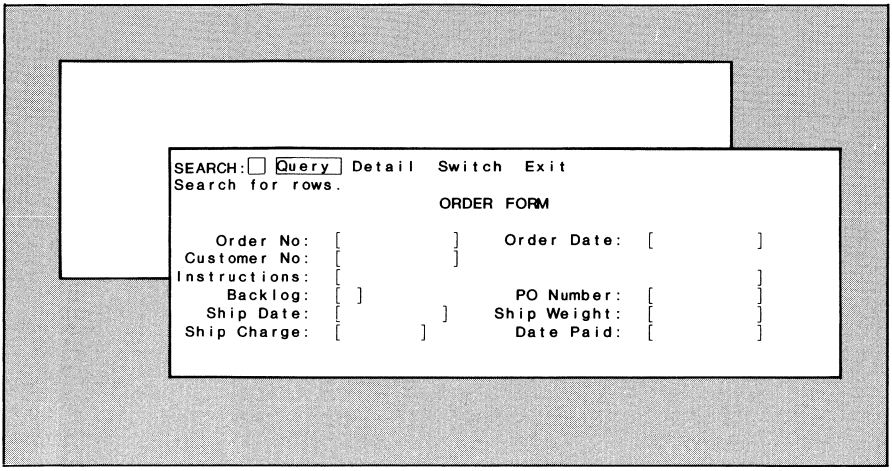
More About the CLEAR WINDOW Statement

The current window need not be the window specified in a CLEAR WINDOW statement. For example, in the following screen, the **ow2** window is the current window:



The following statement clears the **cw1** window of all text:

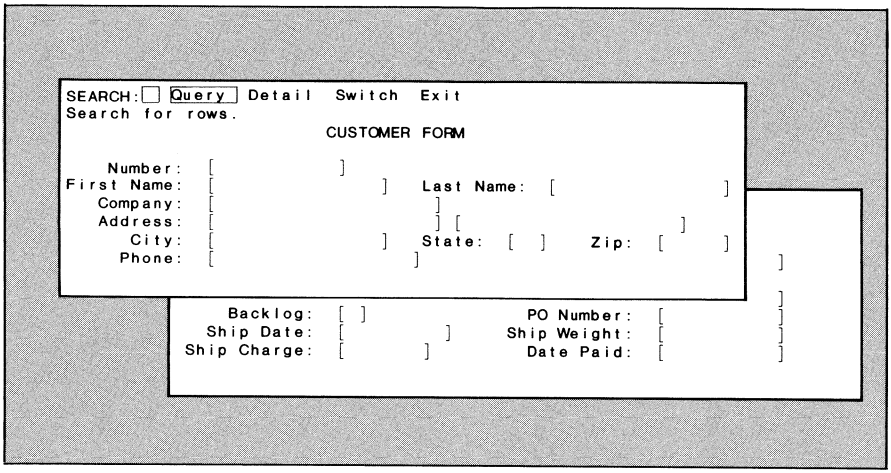
```
CLEAR WINDOW cw1
```



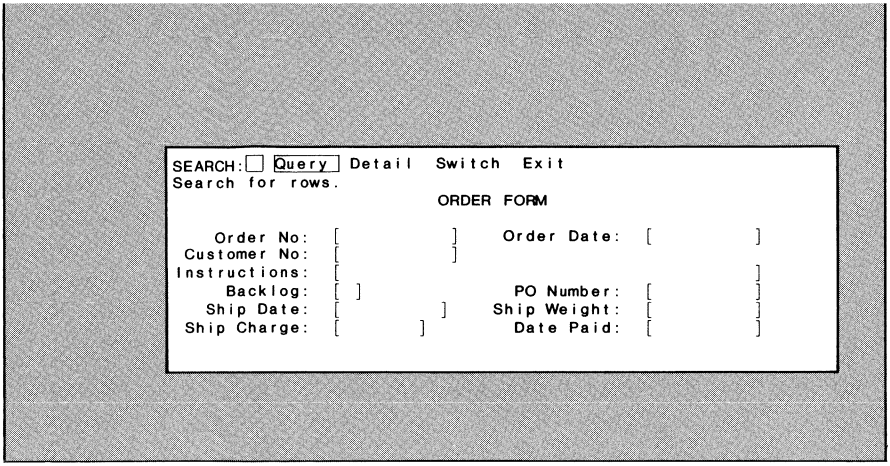
Note: The CLEAR WINDOW *screen* statement compares with the CLEAR SCREEN statement in the following respect: the CLEAR SCREEN statement makes the screen the current window and clears it, while the CLEAR WINDOW *screen* statement clears the screen except for the area occupied by the window(s) you have opened. The CLEAR WINDOW statement does not change the current window.

More About the CLOSE WINDOW Statement

When you execute a CLOSE WINDOW statement, **INFORMIX-4GL** removes the window from the screen and redisplays any text overwritten by the window. The next screen is from the **wind2.4gl** program. The **cw1** window is the current window.

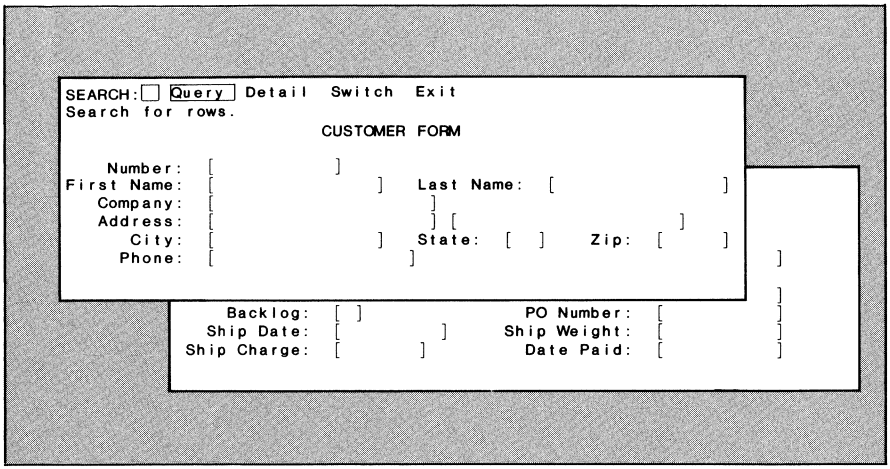


If you close the **cw1** window, **INFORMIX-4GL** erases the window from the screen and redisplays the entire **ow2** window. The **ow2** window is now the current window.



When you close the current window, **INFORMIX-4GL** removes it from the screen; the previous window in its list becomes the current window. The current window is not affected when you close a window other than the current window. **INFORMIX-4GL** simply clears the window from the screen, restores any display overwritten by the window, and removes the window from its list.

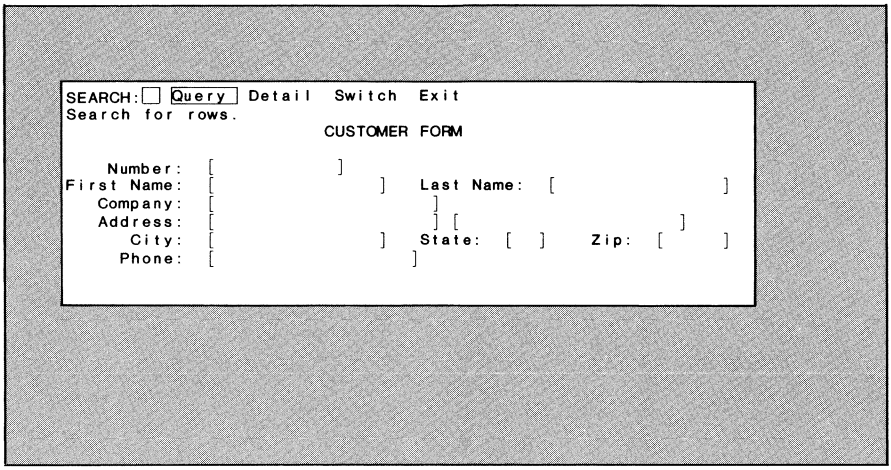
For example, both the **cw1** window and the **ow2** window appear on the following screen. The **cw1** window is the current window.



The next statement closes the **ow2** window and removes it from the screen.

```
CLOSE WINDOW ow2
```

The **cw1** window is the current window and remains completely visible on the screen.



If you close a window that is currently being used for input, INFORMIX-4GL generates a runtime error. For example, you cannot close a window in the middle of an INPUT or INPUT ARRAY statement. You also cannot close the *screen* window.

Examining a Program with Multiple Windows

An earlier section of this chapter describes the **wind2.4gl** program from the user's perspective. In the program, two windows appear on the screen; each window displays a different form with the SEARCH Menu. The user displays a different current window by selecting the **Switch** or **Detail** option on the menu.

The **wind2.4gl** program demonstrates how to use the INFORMIX-4GL window management statements in a multi-window program. This section describes the program blocks and functions that make up the **wind2.4gl** program. The functions are as follows:

```
show_menu        det_cust
set_curs         det_ord
get_cust         change_wind
view_cust        clear_menu
get_ord          mess
view_ord
```

This section also describes the GLOBALS and MAIN statements.

The complete **wind2.4gl** program appears on the next few pages.

```
DATABASE stores

#       The GLOBALS statement defines two program records and
#       a SMALLINT variable used throughout the program.

GLOBALS

        DEFINE
            p_customer    RECORD LIKE customer.*,
            p_orders      RECORD LIKE orders.*,
            cur           SMALLINT

END GLOBALS

#       The MAIN statement opens two windows and displays
#       forms in each window. It sets the global variable cur
#       to 1, and calls the show_menu function.  Finally, it
#       closes the forms and windows.

MAIN

        OPEN WINDOW ow2 AT 10,15
            WITH 11 ROWS,63 COLUMNS
            ATTRIBUTE (BORDER)

        OPEN FORM ordcur FROM "ordcur"

        DISPLAY FORM ordcur

        OPEN WINDOW cw1 AT 5,5
            WITH 11 ROWS,63 COLUMNS
            ATTRIBUTE (BORDER)

        OPEN FORM custcur FROM "custcur"

        DISPLAY FORM custcur

        LET cur = 1

        CALL show_menu()

        CLOSE FORM custcur

        CLOSE FORM ordcur

        CLOSE WINDOW cw1

        CLOSE WINDOW ow2

END MAIN
```

```
#         The show_menu function displays the SEARCH Menu.
#         The SEARCH Menu is the ''main'' menu for the program
#         and appears at the top of the current window.

FUNCTION show_menu()

         DEFINE win_flag SMALLINT

         LET win_flag = 1

         WHILE win_flag = 1

               LET win_flag = 0

               MENU "SEARCH"

                     COMMAND "Query" "Search for rows."
                           IF cur = 1 THEN
                                CALL get_cust()
                           ELSE
                                CALL get_ord()
                           END IF

                     COMMAND "Detail" "Get details."
                           IF cur = 1 THEN
                                CALL det_ord()
                           ELSE
                                CALL det_cust()
                           END IF

                     COMMAND "Switch" "Change the window."
                           LET win_flag = 1
                           EXIT MENU

                     COMMAND "Exit" "Leave the program."
                           EXIT MENU

               END MENU

               IF win_flag = 1 THEN
                     CALL change_wind()
               END IF

         END WHILE

END FUNCTION
```

```
#         The set_curs function uses a CASE statement to test
#         the value of the argument passed to it by other functions
#         in the program.  It prepares a SELECT statement, declares
#         a scroll cursor for the SELECT statement, and opens the cursor.

FUNCTION set_curs(flag)

         DEFINE flag SMALLINT,
                query_1, query_str CHAR(300)

         CASE flag

               WHEN 1    # function called by get_cust
                     CONSTRUCT query_1 ON customer.* FROM customer.*
                     LET query_str = "SELECT * FROM customer WHERE ",
                           query_1 CLIPPED

               WHEN 2    # function called by get_ord
                     CONSTRUCT query_1 ON orders.* FROM orders.*
                     LET query_str = "SELECT * FROM orders WHERE ",
                           query_1 CLIPPED

               WHEN 3    # function called by det_ord
                     LET query_str = "SELECT * FROM orders WHERE ",
                           "orders.customer_num = ",
                           p_customer.customer_num USING "###"

         END CASE

         PREPARE s_1 FROM query_str

         DECLARE q_curs SCROLL CURSOR FOR s_1

         OPEN q_curs

END FUNCTION


#         The get_cust function allows the user to perform a
#         query by example on the cust_cur form displayed in the
#         cw1 window.

FUNCTION get_cust()

         CALL clear_menu()

         CALL mess("Enter search criteria for one or more customers.")

         CALL set_curs(1)

         CALL view_cust()

END FUNCTION
```

```
#         The view_cust function performs a fetch first and tests
#         whether a customer row is returned.  If the active set is
#         empty, the function ends.  If the active set contains at
#         least one row, the function displays the row along with a
#         menu that allows the user to browse through the
#         row(s) in the active set.

FUNCTION view_cust()

         FETCH FIRST q_curs INTO p_customer.*

         IF status = NOTFOUND THEN
               CALL mess("No customers found.")
               CLOSE q_curs
               RETURN
         ELSE
               DISPLAY BY NAME p_customer.*
         END IF

         MENU "BROWSE"

               COMMAND "Next" "View the next customer in the list."
                     FETCH NEXT cust_curs INTO p_customer.*
                     IF status = NOTFOUND THEN
                           CALL mess("No more customers in this direction.")
                           FETCH LAST cust_curs INTO p_customer.*
                     END IF
                     DISPLAY BY NAME p_customer.*

               COMMAND "Previous" "View the previous customer in the list."
                     FETCH PREVIOUS cust_curs INTO p_customer.*
                     IF status = NOTFOUND THEN
                           CALL mess("No more customers in this direction.")
                           FETCH FIRST cust_curs INTO p_customer.*
                     END IF
                     DISPLAY BY NAME p_customer.*

               COMMAND "First" "View the first customer in the list."
                     FETCH FIRST cust_curs INTO p_customer.*
                     DISPLAY BY NAME p_customer.*

               COMMAND "Last" "View the last customer in the list."
                     FETCH LAST cust_curs INTO p_customer.*
                     DISPLAY BY NAME p_customer.*

               COMMAND "Select-and-exit" "Select the current customer."
                     CALL clear_menu()
                     EXIT MENU

         END MENU

         CLOSE q_curs

END FUNCTION



#         The get_ord function allows the user to perform a
#         query by example on the ord_cur form displayed
#         in the ow2 window.

FUNCTION get_ord()

         CALL clear_menu()

         CALL mess("Enter search criteria for one or more orders.")

         CALL set_curs(2)

         CALL view_ord()

END FUNCTION
```

```
#       The view_ord function performs a fetch first and tests
#       whether an order row is returned.  If the active set
#       is empty, the function ends.  If the active set contains
#       at least one row, the function displays the row along
#       with a menu that allows the user to browse through
#       the row(s) in the active set.

FUNCTION view_ord()

        FETCH FIRST q_curs INTO p_orders.*

        IF status = NOTFOUND THEN
              CALL mess("No orders found.")
              CLOSE q_curs
              RETURN
        ELSE
              DISPLAY BY NAME p_orders.*
        END IF

        MENU "BROWSE"

              COMMAND "Next" "View the next order in the list."
                    FETCH NEXT order_curs INTO p_orders.*
                    IF status = NOTFOUND THEN
                          CALL mess("No more orders in this direction.")
                          FETCH LAST order_curs INTO p_orders.*
                    END IF
                    DISPLAY BY NAME p_orders.*

              COMMAND "Previous" "View the previous order in the list."
                    FETCH PREVIOUS order_curs INTO p_orders.*
                    IF status = NOTFOUND THEN
                          CALL mess("No more orders in this direction.")
                          FETCH FIRST order_curs INTO p_orders.*
                    END IF
                    DISPLAY BY NAME p_orders.*

              COMMAND "First" "View the first order in the list."
                    FETCH FIRST order_curs INTO p_orders.*
                    DISPLAY BY NAME p_orders.*

              COMMAND "Last" "View the last order in the list."
                    FETCH LAST order_curs INTO p_orders.*
                    DISPLAY BY NAME p_orders.*

              COMMAND "Select-and-exit" "Select the current order."
                    CALL clear_menu()
                    EXIT MENU

        END MENU

        CLOSE q_curs

END FUNCTION

#       The det_cust function is called when the user selects
#       the Detail option on the SEARCH Menu and ow2 is
#       the current window.  The det_cust function provides
#       information about the customer who placed the order
#       currently displayed in the ow2 window.

FUNCTION det_cust()

        DEFINE answer CHAR(1)

        CURRENT WINDOW IS cw1

        SELECT * INTO p_customer.* FROM customer
              WHERE customer_num = p_orders.customer_num

        DISPLAY BY NAME p_customer.*

        PROMPT "Press RETURN to return to the other window: "
              FOR answer

END FUNCTION
```

```
#         The det_ord function is called when the user selects
#         the Detail option on the SEARCH Menu and cw1 is
#         the current window.  The det_ord function provides
#         information about the orders placed by the customer
#         currently displayed in the cw1 window.

FUNCTION det_ord()

         DEFINE answer CHAR(1)

         CURRENT WINDOW IS ow2

         CALL set_curs(3)

         CALL view_ord()

         PROMPT "Press RETURN to return to the other window: "
               FOR answer

END FUNCTION



#         The change_wind function is called when the user
#         selects the Switch option on the SEARCH Menu.
#         The function changes the current window.

FUNCTION change_wind()

         CALL clear_menu()

         IF cur = 1 THEN
               CURRENT WINDOW IS ow2
               LET cur = 2
         ELSE
               CURRENT WINDOW IS cw1
               LET cur = 1
         END IF

END FUNCTION



#         The clear_menu function clears the top two
#         lines in the current window of text.

FUNCTION clear_menu()

         DISPLAY "" AT 1, 1
         DISPLAY "" AT 2, 1

END FUNCTION



#         The mess function is passed a character string as an argument
#         and displays the characters on the 11th (last) line of the
#         current window.

FUNCTION mess(str)

         DEFINE str CHAR(50)

         DISPLAY str CLIPPED AT 11,1

         SLEEP 3

         DISPLAY "" AT 11,1

END FUNCTION
```

The GLOBALS Statement

The GLOBALS statement defines the **p_customer** and **p_orders** program records LIKE the **customer** and **orders** tables. These global records are used throughout the program.

The statement also defines a global variable called **cur** that is used to keep track of the current window.

The MAIN Statement

The MAIN statement opens two windows and displays a form in each window. The MAIN statement performs the following operations:

1. Displays the **ordcur** form in the bordered **ow2** window.

2. Displays the **custcur** form in the bordered **cw1** window. **cw1** is the current window, since it is opened after the **ow2** window.

   Explicit dimensions are provided for each window; the windows are not opened WITH FORM. The program uses OPEN FORM and DISPLAY FORM statements to open and display each form in the appropriate window.

3. Sets the value of the **cur** variable to 1, indicating that the **cw1** window is the current window.

4. Calls the **show_menu** function.

5. Closes the **custcur** and **ordcur** forms.

6. Closes the **cw1** and **ow2** windows.

The **show_menu** *Function*

The **show_menu** function displays the SEARCH Menu. This menu is the "main" menu for the program. Options on this menu allow the user to perform a query by example on the form displayed in the current window, to perform a query on the form displayed in the alternate window, to switch windows, and to exit the program.

The **show_menu** function performs the following operations:

1. Defines the local variable **win_flag** and sets its value to 1.

2. Uses a WHILE loop to display the SEARCH Menu in the current window as long as the value of **win_flag** is 1. The WHILE loop performs the following operations:

   a. Sets the value of **win_flag** to 0.

   b. Uses a MENU statement to create the SEARCH Menu. The options on the SEARCH Menu are described below:

      Query Calls the **get_cust** function if the value of **cur** is 1. (The program sets the value of **cur** to 1 whenever the **cw1** window is the current window.) The **get_cust** function allows the user to query the **customer** table. If the value of **cur** is not 1 (indicating that **ow2** window is the current window), the program calls the **get_ord** function. The **get_ord** function allows the user to query the **orders** table.

      Detail Calls the **det_ord** function if the value of **cur** is 1. The **det_ord** function finds the orders for the current customer by querying the **orders** table based upon the value of the **p_customer.customer_num** program variable.

If the value of **cur** is not 1, the program calls the **det_cust** function. The **det_cust** function finds the customer who placed the current order by querying the **customer** table based upon the value of the **p_orders.customer_num** program variable.

Switch Sets the value of the **win_flag** variable to 1 and exits the menu.

Exit Exits the menu.

c. Uses an IF statement to test the value of **win_flag**. If **win_flag** equals 1 (indicating that the user has selected the **Switch** option), the program calls the **change_wind** function. The **change_wind** function changes the current window and resets the value of the **cur** variable.

d. Repeats Steps a through c as long as the user selects the **Switch** option to leave the SEARCH Menu. (If the user selects the **Exit** option, the value of **wind_flag** remains 0 and the WHILE loop ends.)

The **set_curs** *Function*

The **set_curs** function is called by the **get_cust**, **get_ord**, and **det_ord** functions. **set_curs** uses the value passed by the calling function to determine which of three SELECT statements to prepare.

The **set_curs** function performs the following actions:

1. Defines three local variables:

   - **flag** receives its value from the calling function and is used in the CASE statement to determine the string that is assigned to **query_str**.

- **query_1** stores the conditions for the WHERE clause of a SELECT statement.

- **query_str** stores the SELECT statement that will be prepared.

2. Uses a CASE statement to determine the value assigned to the **query_str** variable.

   - When **flag** is 1 (indicating that the function was called by **get_cust**), INFORMIX-4GL executes a CONSTRUCT statement that allows the user to perform a query by example on the **custcur** form. INFORMIX-4GL then concatenates the string containing the search criteria to the string that contains the rest of the SELECT statement and assigns the result to **query_str**.

   - When **flag** is 2 (indicating that the function was called by **get_ord**), INFORMIX-4GL executes a CONSTRUCT statement that allows the user to perform a query by example on the **ordcur** form. INFORMIX-4GL then concatenates the string containing the search criteria to the string that contains the rest of the SELECT statement and assigns the result to **query_str**.

   - When **flag** is 3 (indicating that the function was called by **det_ord**), INFORMIX-4GL concatenates the value in **p_customer.customer_num** to the string that contains the rest of the SELECT statement and assigns the result to **query_str**.

   (For more information about the CASE statement, see Chapter 7 of the *INFORMIX-4GL Reference Manual*).

3. Prepares an executable SELECT statement from **query_str**.

4. Declares a scrolling cursor for the prepared SELECT statement. (The scrolling cursor allows you to retrieve rows in the active set in sequential or random order.)

5. Opens the cursor.

The **get__cust** *Function*

The **get__cust** function allows the user to perform a query by example on the **custcur** form displayed in the **cw1** window.

The **get__cust** function performs the following operations:

1. Calls the **clear__menu** function that clears the first two lines of the current window.

2. Calls the **mess** function that displays the following message in the current window:

   Enter search criteria for one or more customers.

3. Passes the value 1 to the **set__curs** function. When **set__curs** receives this value, it allows the user to perform a query by example on the **custcur** form and prepares the appropriate SELECT statement.

4. Calls the **view__cust** function that allows the user to view the row(s) returned by the SELECT statement prepared in the **set__curs** function.

The **view__cust** *Function*

The **view__cust** function allows the user to view the customer rows retrieved by a SELECT statement prepared in the **set__curs** function. The **view__cust** function performs the following operations:

1. Attempts to retrieve the first row from the active set.

2. Tests the **status** variable to determine whether a row is retrieved. If no row is retrieved, the function displays the message No customers found, closes the cursor, and returns. If a row is retrieved, the function displays the values on the **custcur** form.

3. Displays the BROWSE menu that allows the user to view the row(s) in the active set. (The BROWSE menu is very similar to the BROWSE menu in the **c_menu2** program described in Chapter 8.)

4. Closes the cursor.

The **get_ord** *Function*

The **get_ord** function allows the user to perform a query by example on the **ordcur** form displayed in the **ow2** window.

The actions in the **get_ord** function parallel those in the **get_cust** function.

The **view_ord** *Function*

The **view_ord** function is called by the **get_ord** and **det_ord** functions and displays the orders retrieved by a SELECT statement prepared in the **set_curs** function. The **view_ord** function performs the following operations:

1. Attempts to retrieve the first row from the active set.

2. Tests the **status** variable to determine whether a row is retrieved. If no row is retrieved, the function displays the message No orders found in the window, closes the cursor, and returns. If a row is retrieved, the function displays the values on the **ordcur** form.

3. Displays the BROWSE menu that allows the user to view the order(s) in the active set.

4. Closes the cursor.

The **det_cust** *Function*

The program calls the **det_cust** function when the user selects the **Detail** option on the SEARCH Menu and **ow2** is the current window. The **det_cust** function provides information about the customer who placed the order currently displayed in the **ow2** window.

The **det_cust** function performs the following operations:

1. Defines a local variable that is used in a PROMPT statement.

2. Makes **cw1** the current window.

3. Selects a row from the **customer** table based on the value in the **p_orders.customer_num** program variable.

   No cursor is declared because the SELECT statement returns only one row. (Only one customer can place an order.)

4. Displays the customer row on the **custcur** form.

5. Prompts the user to press **RETURN** to return to the **ow2** window.

   Note: If a menu is running in the **ow2** window and the user selects an option that changes the current window to **cw1**, **INFORMIX-4GL** automatically restores **ow2** as the current window after it executes the last statement for the option. In this case, the SEARCH menu is displayed in **ow2** and the user selects the **Detail** option that changes the current window to **cw1**. **INFORMIX-4GL** automatically restores **ow2** as the current window after it executes the last statement for the Detail option and redisplays the SEARCH menu.

The **det_ord** *Function*

The program calls the **det_ord** function when the user selects the **Detail** option on the SEARCH Menu and **cw1** is the current window. The **det_ord** function provides information on the orders placed by the customer currently displayed in the **cw1** window.

The **det_ord** function performs the following operations:

1. Defines the **answer** local variable.

2. Makes **ow2** the current window.

3. Passes the value 3 to the **set_curs** function. (The **set_curs** function uses this value to determine which SELECT statement to prepare.)

4. Calls the **view_ord** function that allows the user to view the row(s) in the active set.

5. Prompts the user to press **RETURN** to return to the **cw1** window.

   Note: In this case, the SEARCH menu is displayed in the **cw1** window and the user selects the **Detail** option that changes the current window to **ow2**. **INFORMIX-4GL** automatically restores **cw1** as the current window after it executes the last statement for the Detail option and redisplays the SEARCH menu.

The **change_wind** *Function*

The program calls the **change_wind** function when the user selects the **Switch** option on the SEARCH Menu. The **change_wind** function performs the following operations:

1. Calls the **clear_menu** function that clears all text from the top two lines of the current window.

2. Tests the value of the **cur** variable to determine the current window. The program makes the alternate window the current window, and sets **cur** to the value corresponding to the new current window.

The **clear_menu** *Function*

The **clear_menu** function uses two DISPLAY statements to clear text from the top two window lines.

The **mess** *Function*

The **mess** function accepts a character string as an argument and displays the string on line 11 in the current window. The function then sleeps for three seconds and clears the line of text.

Chapter Summary

- You can display separate INFORMIX-4GL activities in different windows.
- You use the OPEN WINDOW statement to open a window in an INFORMIX-4GL program. You indicate the name, location, and dimensions of the window in the statement.
- You can optionally display a border around the window, display the window in reverse video or in color, and change the values for the prompt, message, form, and comments lines by including an ATTRIBUTE clause in the OPEN WINDOW statement.
- You use the CLOSE WINDOW statement to close a window. The current window need not be the window specified in the statement.
- Your INFORMIX-4GL applications can include windows that display screen forms, help information, prompts, menus, or report output.
- You use the CLEAR statement with the WINDOW keyword to clear the display within a window. The current window need not be the window specified in the statement.
- You use the CURRENT WINDOW statement to make a window the current or topmost window.
- The terminal screen is the current window when a program starts.

Chapter 14

Learning More About INFORMIX-4GL

Chapter 14 Table of Contents

Chapter Overview 5

Access Privileges 6
Granting and Revoking Database Privileges 6
The CONNECT Privilege 6
The RESOURCE Privilege 7
The DBA Privilege 7
Granting and Revoking Table Privileges 8

Modifying a Database 9
Changing the Structure of a Database 9
Removing a Table 10
Removing an Index 10
Renaming a Table 11
Removing a Database 11
Changing the Structure of a Table 12
Renaming a Column 12
The ALTER TABLE Statement 12
Combining Keywords in the ALTER TABLE Statement 14

Transactions 15
Creating a Database with Transactions 15
Specifying Transactions 16
Starting a New Transaction Log 17
Recovering a Database 17

Audit Trails 18
Comparing Audit Trails and the Transaction Log 18

Views 19
Creating and Dropping Views 19

Input and Display Attributes 20
The **upscol** Utility 21

Chapter Summary 22

Chapter Overview

This chapter briefly describes important features of INFORMIX-4GL that were not covered in previous chapters and shows you where to find more information about these topics. The discussion includes

- How to grant and revoke access to a database or table
- How to change the structure of a database or table
- How to use transactions and audit trails to protect the integrity of your database
- How to use views to create dynamic "windows" on your database
- How to specify input and display attributes for one or more form specifications using the **upscol** utility

For complete information about the statements used in this chapter, see the *INFORMIX-4GL Reference Manual.*

Access Privileges

In Chapter 2 you learned how to let others have access to your database by using the GRANT CONNECT TO PUBLIC statement. This section briefly describes table and database privileges you can grant and revoke using the GRANT and REVOKE statements. For complete information about GRANT and REVOKE, refer to Chapter 7, "INFORMIX-4GL Statement Syntax," of the *INFORMIX-4GL Reference Manual.*

Granting and Revoking Database Privileges

With **INFORMIX-4GL** you can use the GRANT and REVOKE statements to control a user's access to a database independently of any operating system privileges that the user might have. The following sections describe the CONNECT, RESOURCE, and DBA privileges that you can use with the GRANT and REVOKE statements to control access to your database.

The CONNECT Privilege

You must grant the CONNECT privilege if you want one or more users to have access to your database. With the CONNECT privilege, a user may specify your database in a DATABASE statement, but cannot create or drop tables and indexes. The formats for GRANT CONNECT and REVOKE CONNECT are

```
GRANT CONNECT TO PUBLIC
REVOKE CONNECT FROM PUBLIC

GRANT CONNECT TO user-list
REVOKE CONNECT FROM user-list
```

where PUBLIC refers to all users and *user-list* is a list of login names that corresponds to specific users.

Once you have granted the CONNECT privilege to one or more users, they also have default table privileges unless someone has explicitly changed them (see the section "Granting and Revoking Table Privileges" for more information).

The RESOURCE Privilege

With the RESOURCE privilege, users have the CONNECT privilege as well as the ability to create or drop their own tables and indexes in the database. The formats for GRANT RESOURCE and REVOKE RESOURCE are

```
GRANT RESOURCE TO PUBLIC
REVOKE RESOURCE FROM PUBLIC

GRANT RESOURCE TO user-list
REVOKE RESOURCE FROM user-list
```

The DBA Privilege

If you create a database, you automatically have the DBA (Database Administrator) privilege. If you grant the DBA privilege to another user, that user has the RESOURCE privilege as well as the ability to

- Create, alter, or drop any tables and indexes in the database, including the system catalogs
- Drop, start, and roll forward the database (see the sections "Modifying a Database" and "Transactions" for more information about these operations)
- Grant and revoke CONNECT, RESOURCE, and DBA privileges

The only restriction placed on users with DBA status is the inability to revoke the DBA privilege from themselves. However, a user with DBA status may grant the privilege to another user, who may then revoke it from the grantor.

The formats for GRANT DBA and REVOKE DBA are

GRANT DBA TO *user-list*
REVOKE DBA FROM *user-list*

In most cases, you will want only one user to have the DBA privilege.

Granting and Revoking Table Privileges

You can use the GRANT and REVOKE statements to specify the operations you want to let a user perform on a table you have created. The formats are

GRANT *table-privilege* ON *table-name* TO PUBLIC
REVOKE *table-privilege* ON *table-name* FROM PUBLIC

GRANT *table-privilege* ON *table-name* TO *user-list*
REVOKE *table-privilege* ON *table-name* FROM *user-list*

where *table-name* is the name of a table, and *table-privilege* is a list of one or more of the privileges listed below:

ALTER	adds a column, delete a column, or change the data type of a column (see the section "Changing the Structure of a Table" for more information).
DELETE	deletes rows from a table.
INDEX	creates indexes for a table.
INSERT	inserts rows in a table.
SELECT	selects information from one or more columns in a table.
UPDATE	updates information from one or more columns in a table.
ALL	performs any or all of the operations listed above.

By default a user has all privileges except ALTER for tables in the database unless someone has explicitly changed them. This means that the user may insert, retrieve, update, and delete information in your database tables. The user may also create indexes for those tables if the user has the RESOURCE privilege.

Modifying a Database

In Chapter 2 you learned how to create a database with the CREATE DATABASE, CREATE TABLE, and CREATE INDEX statements. This section briefly explains how to change the structure of a database or table using the following commands:

DROP TABLE	CLOSE DATABASE
DROP INDEX	RENAME COLUMN
RENAME TABLE	ALTER TABLE
DROP DATABASE	

For complete information about these statements, refer to Chapter 7, "INFORMIX-4GL Statement Syntax," of the *INFORMIX-4GL Reference Manual.*

Changing the Structure of a Database

You can change the structure of a database by

- Removing a table
- Removing an index
- Renaming a table
- Removing a database

You can make these changes only to databases, tables, and indexes that you created, unless you have DBA status.

Before you rename a table or remove a table or index, be sure the database containing the table or index is the current database (that is, the database you have specified in the DATABASE statement of a program).

Removing a Table

Use the DROP TABLE statement to remove a table from a database. The general format is

DROP TABLE *table-name*

where *table-name* is the name of the table you want to remove.

Caution! Be careful with the DROP TABLE statement. When you use it, INFORMIX-4GL deletes the table you specify, *along with all the data it contains.* If you accidentally delete the wrong table, you will have to recreate the table and restore all the data from a backup copy. If you do not have a backup, you will have to re-enter the data.

Removing an Index

Use the DROP INDEX statement to remove an index. The general format is

DROP INDEX *index-name*

where *index-name* is the name of the index you want to remove.

You can drop only those indexes that you have created. If you accidentally remove the wrong index, you can easily recreate it with the CREATE INDEX statement.

Renaming a Table

Use the RENAME TABLE statement to change the name of an existing table. The general format is

RENAME TABLE *oldname* TO *newname*

where *oldname* is the current name of the table and *newname* is the name you want to assign to it.

If you rename a table, change all references to the table in your form specifications and programs and recompile them.

Removing a Database

Use the DROP DATABASE statement to remove an entire database. The general format is

DROP DATABASE *database-name*

where *database-name* is the name of the database you want to delete.

If you want to drop the current database, you must close it first. The general format to close a database is

CLOSE DATABASE *database-name*

where *database-name* is the name of the database you want to close. Then you can use the DROP DATABASE statement to drop the database you closed.

Caution! Be very careful with the DROP DATABASE statement. When you use it, INFORMIX-4GL deletes all tables, indexes, and data from the database. The DROP DATABASE statement also removes the database directory, if all the files in the directory belong to the database. If the database directory contains files that do not belong to the database, the DROP DATABASE statement removes only the database files and leaves the rest of the directory intact.

Changing the Structure of a Table

You can change the structure of a table at any time, even if you have already stored data in the table. You can

- Rename a column
- Add a column
- Delete a column
- Change the data type of a column

You cannot change the structure of a table created by another person unless you have the ALTER privilege or DBA status.

You can only change the structure of tables that belong to the current database. If you try to change a table that does not belong to the current database, **INFORMIX-4GL** displays an error message.

Renaming a Column

Use the RENAME COLUMN statement to change the name of a column. The general format is

RENAME COLUMN *table.old-column* TO *new-column*

where *table.old-column* is the column you want to rename and *new-column* is the new name you want to assign to the column.

When you rename a column, you will have to change references to the column in all form specifications and programs and recompile them.

The ALTER TABLE Statement

You can use the ALTER TABLE statement to add a column to a table, delete a column from a table, or change the data type of a column.

Adding a Column. Use the ALTER TABLE statement with the ADD keyword to add one or more new columns to a table at any time, even when the table already contains data. The format is

```
ALTER TABLE table-name
   ADD (new-column-name data-type [,...] )
```

where *table-name* is the name of the table you want to alter, *new-column-name* is the name of the column you want to add, and *data-type* is the data type of the new column. When **INFORMIX-4GL** encounters this statement, it appends the new column to the list of columns in the table. You can add as many columns to the end of the table as you want, as long as each column name is unique.

When you add columns to a table, **INFORMIX-4GL** assigns a NULL to each new column.

Inserting a New Column in a Table. You can use a BEFORE clause with the ADD keyword of the ALTER TABLE statement to insert a new column before an existing column. The format is

```
ALTER TABLE table-name
      ADD (new-column-name data-type
      BEFORE old-column-name [,...] )
```

where *old-column-name* is the column before which you want to insert the new column.

You can insert as many columns in the table as you want. If you add a column, you may have to add references to the column in your form specifications or programs and recompile them. Be especially careful of places in form specifications or programs where you have specified *table.** or used the THRU keyword.

You can only add a column once. If you accidentally add a column in the wrong location, you will have to delete it first (see "Dropping a Column from a Table," below) and then add it again in the correct location.

Dropping a Column from a Table. Use the ALTER TABLE statement with the DROP keyword to drop a column and any data it contains. The format is

```
ALTER TABLE table-name
    DROP (old-column-name [,...] )
```

As with other ALTER TABLE keywords, you can work with more than one column at a time. Use the DROP keyword when you are sure you have no need for the data in a column or when you have just added a new column in the wrong location.

If you drop a column, you will have to drop references to the column in all form specifications and programs and recompile them. Be especially careful of places in form specifications or programs where you have specified *table.** or used the THRU keyword.

Changing the Data Type of a Column. You can use the MODIFY keyword of the ALTER TABLE statement to change the data type of a column.

You normally change the data type of a column when you need to store larger values than the current data type can accommodate. The format is

```
ALTER TABLE table-name
    MODIFY (old-column-name new-data-type [,...] )
```

When **INFORMIX-4GL** encounters an ALTER TABLE statement with the MODIFY keyword, it converts all values from the old data type to the new data type.

Combining Keywords in the ALTER TABLE Statement

You can use two or more of the keywords (ADD, DROP, MODIFY) in a single statement. The keywords can appear in any order, separated by commas, and are run sequentially. If **INFORMIX-4GL** cannot run part of an ALTER TABLE

statement, it ignores the entire statement and leaves the table unchanged. See the *INFORMIX-4GL Reference Manual* for more information.

Transactions

A transaction is a series of operations on a database that you want INFORMIX-4GL to complete entirely or not at all. If you are writing an accounting program, for example, you will often want INFORMIX-4GL to perform several operations as a unit to ensure that the books remain in balance. INFORMIX-4GL provides several statements that give you the ability to protect the integrity of your database by treating several operations as a single transaction:

```
CREATE DATABASE WITH LOG
START DATABASE
BEGIN WORK
COMMIT WORK
ROLLBACK WORK
ROLLFORWARD DATABASE
```

For complete information about these statements, refer to Chapter 7 of the *INFORMIX-4GL Reference Manual.*

Creating a Database with Transactions

In order to use transactions with a database, you must have a transaction log file in which INFORMIX-4GL records all modifications to the database. A database with a transaction log is described as a database "with transactions."

Use the CREATE DATABASE statement and the WITH LOG IN clause to create a database and specify a transaction log. The format is

CREATE DATABASE *database-name* WITH LOG IN "*pathname*"

where *database-name* is the name of the database you want to create and *pathname* is the full pathname of the transaction log

file. This log is equivalent to an audit trail and is used by INFORMIX-4GL to process transactions or bring archive copies of the database up to date.

If you have created a database without transactions, you can add transactions by using the START DATABASE statement described in the *INFORMIX-4GL Reference Manual.* The START DATABASE statement also allows you to change the name of a transaction log file.

Specifying Transactions

When you want INFORMIX-4GL to execute statements that access the database as a group, you should use BEGIN WORK before the first statement in the group and COMMIT WORK or ROLLBACK WORK after the last statement in the group. To use transactions efficiently, you should keep the number of statements in a transaction small.

If you are satisfied that the group of statements has produced the desired result, you can terminate the transaction with COMMIT WORK. If you are not happy with the result, you should terminate the transaction with ROLLBACK WORK. This statement restores the database to the state that existed before you issued BEGIN WORK with an important exception. All statements that modify tables, columns, and indexes are committed if they were executed successfully.

Note: If you create a database with transactions, but do not execute the BEGIN WORK statement, INFORMIX-4GL treats each statement as a single transaction. This means that INFORMIX-4GL commits each statement if INFORMIX-4GL executes it successfully. If the statement fails, INFORMIX-4GL rolls the database back to the state before the statement was executed.

Starting a New Transaction Log

Since a transaction log file can become quite large, you will periodically want to make a back-up copy of your database and purge the transaction log file. See the *INFORMIX-4GL Reference Manual* for details.

Recovering a Database

If your database becomes corrupt, you can recover the database by restoring the back-up copy and executing the ROLLFORWARD DATABASE statement. The format is

ROLLFORWARD DATABASE *database-name*

where *database-name* is the name of the database you want to recover.

Audit Trails

An audit trail is a file that contains a history of all additions, deletions, updates, and manipulations to a single table. Should the table become corrupted, you can use the audit trail to restore the table.

Comparing Audit Trails and the Transaction Log

Audit trails compare with transactions and a transaction log in the following respects:

- The transaction log contains a coordinated record of all database modifications, including updates, insertions, and deletions that affect multiple tables. The audit trail records modifications to a single table.

- Transactions guarantee that **RDSQL** statements are either successfully completed or are completely cancelled. An audit trail does not protect against partial completion of an **RDSQL** statement.

- You can use the transaction log to recover an entire database. You can use an audit trail file to recover only the table for which it is created.

- With audit trails you can record modifications to a single important table without incurring the system expense of maintaining a transaction log on the entire database.

For complete information about the use of audit trails, refer to Chapter 2, "Using RDSQL," in the *INFORMIX-4GL Reference Manual.*

Views

In some cases, you may want a user to work with only part of a database, especially if it contains information that you want to keep confidential. With **INFORMIX-4GL**, you can control what the user sees by creating a dynamic window on the database called a *view*. **INFORMIX-4GL** provides two statements for handling views:

```
CREATE VIEW
DROP VIEW
```

For complete information about these statements, refer to Chapter 7 of the *INFORMIX-4GL Reference Manual.*

Creating and Dropping Views

You can use the CREATE VIEW statement to select the rows, columns, and aggregates from the database that you want the user to see. The CREATE VIEW statement has the following format

```
CREATE VIEW view-name [(column-list)]
    AS SELECT-statement [WITH CHECK OPTION]
```

where *view-name* is the name of the view, *column-list* is a list of names for the selected columns and aggregates, and *SELECT-statement* is an **INFORMIX-4GL** SELECT statement.

Since a view has a name and resembles a table in many ways, you can substitute a view name for a table name in most **INFORMIX-4GL** statements. This means, for example, that you can allow the user to insert, select, update, and delete information through a view. Moreover, if you have created a view with the WITH CHECK OPTION, **INFORMIX-4GL** makes sure that any data entered or updated by the user satisfies the conditions in the WHERE clause of the SELECT statement that defines the view. For example, if you create a view with the WITH CHECK OPTION that allows a user to work only with customers in California, the user will not be able to enter a

customer who resides in Oregon. This way, you can prevent a user from entering information that will not be accessible through the view.

You can delete a view using the DROP VIEW statement. The format is:

DROP VIEW *view-name*

where *view-name* is the name of a view. See the *INFORMIX-4GL Reference Manual* for more information about views.

Input and Display Attributes

In Chapter 6 you learned about input and display attributes such as DEFAULT and REVERSE that you can include after the field names in the ATTRIBUTES section of a form specification. With **INFORMIX-4GL**, you can also create two database tables called **syscolval** and **syscolatt** to store the attributes for columns you plan to name in one or more form specifications.

When you compile a form specification, **FORM4GL** will search **syscolval** and **syscolatt** for the input and display attributes for each column listed in the ATTRIBUTES section of the form specification. **FORM4GL** then adds the attributes (if any) to those that were explicitly mentioned in the form specification file so that **INFORMIX-4GL** can enforce them during INPUT and DISPLAY statements. (In case of conflict, the attributes listed in the form specification have precedence.) This feature is especially useful if you want to use the same input and display attributes for a column given as a field name in several different form specifications.

The **upscol** *Utility*

The **upscol** utility is a menu-oriented program that creates the **syscolval** and **syscolatt** tables and allows you to set input and display attributes for any database columns that you specify.

With the **upscol** program, you can set the input attributes listed below:

INCLUDE
DEFAULT
PICTURE
COMMENTS
SHIFT (UPSHIFT and DOWNSHIFT)
VERIFY
AUTONEXT

You can also set the following display attributes:

Attribute	Description
COLOR	Intensity for monochrome terminals or color for color terminals.
REVERSE	Reverse video.
UNDER	Underline.
BLINK	Blinking.
LEFT	Left justification in numeric fields.
FORMAT	Format in DECIMAL, SMALLFLOAT, FLOAT, and DATE fields. (If you specify a format, it will be applied unconditionally.)
WHERE	Conditions where all of the previous display attributes except FORMAT are in effect.

Note: You can specify several attributes in the **syscolatt** table that you cannot use in form specification files. These include COLOR, BLINKING, UNDER(line), LEFT, and WHERE.

For more information on input and display attributes, as well as the **upscol** program, refer to Chapter 3 and Appendix E of the *INFORMIX-4GL Reference Manual.*

Chapter Summary

- You can use the GRANT and REVOKE statements to control a user's access to the database independently of any operating-system privileges that the user might have.

- You can grant or revoke the database privileges CONNECT, RESOURCE, and DBA, as well as the table privileges ALTER, DELETE, INDEX, INSERT, SELECT, and UPDATE.

- You can change the structure of a database with the DROP TABLE, DROP INDEX, RENAME TABLE, DROP DATABASE, and CLOSE DATABASE statements.

- You can use the RENAME COLUMN and ALTER TABLE statements to change the structure of a table.

- Use transactions if you want **INFORMIX-4GL** to complete a series of operations on a database entirely or not at all.

- You can use the CREATE DATABASE statement and the WITH LOG IN clause to create a database that handles transactions.

- You can add transactions to an existing database or change the name of the transaction log file with the START DATABASE statement.

- You can specify a transaction by using BEGIN WORK before the first statement in a group and COMMIT WORK or ROLLBACK WORK after the last statement in the group.

- You can use the ROLLFORWARD DATABASE statement to recover a database from a back-up copy.

- You can use an audit trail to record modifications to a single important table without incurring the system expense of maintaining a transaction log on the entire database.

- You can create a dynamic window called a view that restricts the user's access to information in a database.

- You can use the CREATE VIEW statement to select the rows, columns, and aggregates from the database that you want a user to work with. You can also drop a view with the DROP VIEW statement.

- You can use the **upscol** program to create two database tables called **syscolval** and **syscolatt** to store input and display attributes for columns you plan to name in the ATTRIBUTES section of one or more form specifications.

List of Appendixes

Appendix A. The *stores* Database

Appendix B. Environment Variables

Appendix C. Reserved Words

Appendix D. Sample Reports

Appendix A

The *stores* Database

The *stores* Database

This appendix contains four sections.

- The first section describes the structure of each table in the **stores** database, presents the RDSQL statement used to create each table, and notes any indexes on columns in each table.

- The second section presents a map of the **stores** database with potential join columns identified.

- The third section discusses the join columns that link the six tables in the database and presents figures illustrating these relationships.

- The final section shows the data contained in each table in the **stores** database.

The **stores** database contains information about a fictitious sporting goods distributor that services stores in the Western United States. The database is made up of the following six tables:

- customer
- orders
- items
- stock
- manufact
- state

customer

The **customer** table contains information about 18 stores that order sporting goods from the distributor. This information includes the name of the store representative and the store name, address, and telephone number. The columns of the **customer** table are as follows:

customer_num	serial(101)
fname	char(15)
lname	char(15)
company	char(20)
address1	char(20)
address2	char(20)
city	char(15)
state	char(2)
zipcode	char(5)
phone	char(18)

The **customer_num** column is indexed as unique. The **zipcode** column is indexed to allow duplicate values.

orders

The **orders** table contains information about orders placed by the distributor's customers. This information includes the order number, date the order was made, the customer number, shipping instructions, whether a backlog exists, the customer's purchase order number, date the order was shipped, weight of the order, shipment charge, and the date the customer paid for the order. The columns of **orders** are as follows:

order_num	serial(1001)
order_date	date
customer_num	integer
ship_instruct	char(40)
backlog	char(1)
po_num	char(10)
ship_date	date
ship_weight	decimal(8,2)
ship_charge	money(6)
paid_date	date

The **order_num** column is indexed and must contain unique values; the **customer_num** column is indexed but allows duplicates.

items

The **items** table keeps track of individual items in an order. For example, some orders only contain one item, while other orders contain as many as five items. Information in the **items** table includes the item number, order number, stock number, manufacturer code, quantity, and the total price for each item that has been ordered. The columns of the **items** table are as follows:

item_num	smallint
order_num	integer
stock_num	smallint
manu_code	char(3)
quantity	smallint
total_price	money(8)

The **order_num** column is indexed and allows duplicate values. A multiple-column index for the **stock_num** and **manu_code** columns also permits duplicate values.

stock

The distributor offers customers 15 different types of sporting goods. For example, the distributor offers baseball gloves from three different manufacturers and basketballs from one manufacturer.

The **stock** table is a catalog of the items sold by the distributor. For each item in stock, the **stock** table lists a stock number identifying the type of item, a code identifying the manufacturer, a description of the item, its unit price, the unit by which the item must be ordered, and a description of the unit (for example, a case containing 10 baseballs).

The **stock** table contains the following columns:

stock_num	smallint
manu_code	char(3)
description	char(15)
unit_price	money(6)
unit	char(4)
unit_descr	char(15)

A multiple-column index for the **stock_num** and **manu_code** columns allows only unique values.

manufact

The distributor handles sporting goods from five manufacturers. Information about these manufacturers is kept in the **manufact** table. This information consists of an identification code and the manufacturer's name. The columns in the **manufact** table are as follows:

manu_code	char(3)
manu_name	char(15)

The **manu_code** column is indexed and must contain unique values.

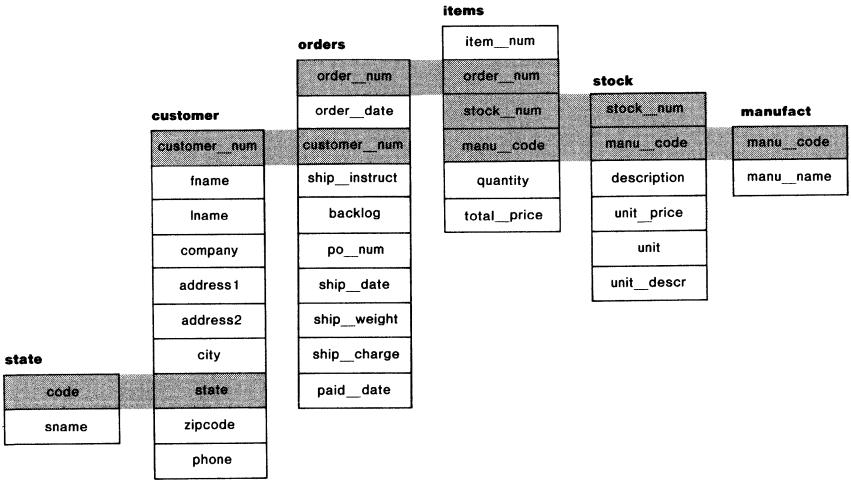
state

The **state** table contains the names and abbreviations of the states. It includes the following columns:

code	char(2)
sname	char(15)

The **code** column is indexed and must contain unique values.

Map of the **stores** *Database*



Join Columns That Link the Database

The six tables of the **stores** database are linked together by join columns. This section identifies these columns and describes how you can use them to retrieve data from several tables at once and display it as if it were stored in a single table. Figures show the relationships among the tables and how information stored in one table supplements information in other tables.

Join Columns in the *customer* and *orders* Tables

The **customer** table and the **orders** table are joined by the **customer_num** column, as shown in the following example:

customer table (detail)

customer_num	fname	lname
101	Ludwig	Pauli
102	Carole	Sadler
103	Philip	Currie
104	Anthony	Higgins

orders table (detail)

order_num	order_date	customer_num
1001	01/20/1986	104
1002	06/01/1986	101
1003	10/12/1986	104
1004	04/12/1986	106

Figure A-1. Tables Joined by the **customer_num** Column

For example, the **customer** table contains a **customer_num** column that holds a number identifying the customer, along with columns for name, company, address, and telephone number. The row with information about Anthony Higgins contains the number 104 in the **customer_num** column. The **orders** table also contains a **customer_num** column that stores the number of the customer who placed a particular order. According to Figure A-1, customer 104 (Anthony Higgins) has placed two orders, since his customer number appears in two rows of the **orders** table.

The join relationship lets you select information from both tables. This means you can retrieve Anthony Higgins's name and address and information about his orders at the same time.

Join Columns in the *orders* and *items* Tables

The **orders** and **items** tables are linked by an **order_num** column that contains an identification number for each order. If a particular order includes several items, the same order number appears in several rows of the **items** table. Figure A-2 shows this relationship.

orders table (detail)

order_num	order_date	customer_num
1001	01/20/1986	104
1002	06/01/1986	101
1003	10/12/1988	104

items table (detail)

item_num	order_num	stock_num	manu_code
1	1001	1	HRO
1	1002	4	HSK
2	1002	3	HSK
1	1003	9	ANZ
2	1003	8	ANZ
3	1003	5	ANZ

Figure A-2. Tables Joined by the **order_num** Column

Join Columns in the *items* and *stock* Tables

The **items** table and the **stock** table are joined by two columns: the **stock_num** column stores a stock number for an item, and the **manu_code** column stores a code that identifies the manufacturer. You need both the stock number and the manufacturer code to uniquely identify an item. For example, the item with the stock number 1 and the manufacturer code HRO is a Hero baseball glove, while the item with the stock number 1 and the manufacturer code HSK is a Husky baseball glove.

The same stock number and manufacturer code may appear in more than one row of the **items** table if the same item belongs to separate orders, as shown in Figure A-3.

items table (detail)

item_num	order_num	stock_num	manu_code
1	1001	1	HRO
1	1002	4	HSK
2	1002	3	HSK
1	1003	9	ANZ
2	1003	8	ANZ
3	1003	5	ANZ
1	1004	1	HRO

stock table (detail)

stock_num	manu_code	description
1	HRO	baseball glove
1	HSK	baseball glove
1	SMT	baseball glove

Figure A-3. Tables Joined by Two Columns

Join Columns in the *stock* and *manufact* Tables

The **stock** table and the **manufact** table are joined by the **manu_code** column. The same manufacturer code may appear in more than one row of the **stock** table if the manufacturer produces more than one piece of equipment. This relationship is illustrated in Figure A-4.

stock table (detail)

stock_num	manu_code	description
1	HRO	baseball glove
1	HSK	baseball glove
1	SMT	baseball glove
2	HRO	baseball

manufact table (detail)

manu_code	manu_name
NRG	Norge
HSK	Husky
HRO	Hero

Figure A-4. Tables Joined by the **manu_code** Column

Join Columns in the *state* and *customer* Tables

The **state** table and the **customer** table are joined by a column that contains the state code. This column is called **code** in the **state** table and **state** in the **customer** table. If several customers live in the same state, the same state code will appear in several rows of the **customer** table, as shown in Figure A-5.

customer
(detail)

customer_num	fname	lname	...	state
101	Ludwig	Pauli	...	CA
102	Carole	Sadler	...	CA
103	Philip	Currie	...	CA

code	sname
AK	Alaska
AL	Alabama
AR	Arkansas
AZ	Arizona
CA	California

state table
(detail)

Figure A-5. Tables Joined by the State-Code Column

Joining lets you rearrange your view of a database whenever you want. It provides flexibility that lets you create new relationships between tables without redesigning the database. You can easily expand the scope of a database by adding new tables that join the existing tables. As you read through this manual, you will see programs that set up the join relationships across tables of the **stores** database. You can refer to the preceding figures whenever you need to review these relationships.

Data in the **stores** *Database*

The data in the **stores** database is displayed in the tables that follow.

customer Table

customer_num	fname	lname	company	address1	address2	city	state	zipcode	phone
101	Ludwig	Pauli	All Sports Supplies	213 Erstwild Court		Sunnyvale	CA	94086	408-789-8075
102	Carole	Sadler	Sports Spot	785 Geary St		San Francisco	CA	94117	415-822-1289
103	Philip	Currie	Phil's Sports	654 Poplar	P. O. Box 3498	Palo Alto	CA	94303	415-328-4543
104	Anthony	Higgins	Play Ball!	East Shopping Cntr.	422 Bay Road	Redwood City	CA	94026	415-368-1100
105	Raymond	Vector	Los Altos Sports	1899 La Loma Drive		Los Altos	CA	94022	415-776-3249
106	George	Watson	Watson & Son	1143 Carver Place		Mountain View	CA	94063	415-389-8789
107	Charles	Ream	Athletic Supplies	41 Jordan Avenue		Palo Alto	CA	94304	415-356-9876
108	Donald	Quinn	Quinn's Sports	587 Alvarado		Redwood City	CA	94063	415-544-8729
109	Jane	Miller	Sport Stuff	Mayfair Mart	7345 Ross Blvd.	Sunnyvale	CA	94086	408-723-8789
110	Roy	Jaeger	AA Athletics	520 Topaz Way		Redwood City	CA	94062	415-743-3611
111	Frances	Keyes	Sports Center	3199 Sterling Court		Sunnyvale	CA	94085	408-277-7245
112	Margaret	Lawson	Runners & Others	234 Wyandotte Way		Los Altos	CA	94022	415-887-7235
113	Lana	Beatty	Sportstown	654 Oak Grove		Menlo Park	CA	94025	415-356-9982
114	Frank	Albertson	Sporting Place	947 Waverly Place		Redwood City	CA	94062	415-886-6677
115	Alfred	Grant	Gold Medal Sports	776 Gary Avenue		Menlo Park	CA	94025	415-356-1123
116	Jean	Parmelee	Olympic City	1104 Spinosa Drive		Mountain View	CA	94040	415-534-8822
117	Arnold	Sipes	Kids Korner	850 Lytton Court		Redwood City	CA	94063	415-245-4578
118	Dick	Baxter	Blue Ribbon Sports	5427 College		Oakland	CA	94609	415-655-0011

orders Table

order_num	order_date	customer_num	ship_instruct	backlog	po_num	ship_date	ship_weight	ship_charge	paid_date
1001	01/20/86	104	ups	n	B77836	02/01/86	20.40	10.00	03/22/86
1002	06/01/86	101	po on box; deliver back door only	n	9270	06/06/86	50.60	15.30	07/03/86
1003	10/12/86	104	via ups	n	B77890	10/13/86	35.60	10.80	11/04/86
1004	04/12/86	106	ring bell twice	y	8006	04/30/86	95.80	19.20	
1005	12/04/86	116	call before delivering	n	2865	12/19/86	80.80	16.20	12/30/86
1006	09/19/86	112	after 10 am	y	Q13557		70.80	14.20	
1007	03/25/86	117		n	278693	04/23/86	125.90	25.20	
1008	11/17/86	110	closed Monday	y	LZ230	12/06/86	45.60	13.80	12/21/86
1009	02/14/86	111	door next to supersaver	n	4745	03/04/86	20.40	10.00	04/21/86
1010	05/29/86	115	deliver 776 Gary if no answer	n	429Q	06/08/86	40.60	12.30	07/22/86
1011	03/23/86	104	ups	n	B77897	04/13/86	10.40	5.00	06/01/86
1012	06/05/86	117		n	278701	06/09/86	70.80	14.20	
1013	09/01/86	104	via ups	n	B77930	09/18/86	60.80	12.20	10/10/86
1014	05/01/86	106	ring bell, kick door loudly	n	8052	05/10/86	40.60	12.30	07/18/86
1015	07/10/86	110	closed Mon	n	MA003	08/01/86	20.60	6.30	08/31/86

items Table

item_num	order_num	stock_num	manu_code	quantity	total_price
1	1001	1	HRO	1	250.0
1	1002	4	HSK	1	960.0
2	1002	3	HSK	1	240.0
1	1003	9	ANZ	1	20.0
2	1003	8	ANZ	1	840.0
3	1003	5	ANZ	5	99.0
1	1004	1	HRO	1	960.0
2	1004	2	HRO	1	126.0
3	1004	3	HSK	1	240.0
4	1004	1	HSK	1	800.0
1	1005	5	NRG	10	280.0
2	1005	5	ANZ	10	198.0
3	1005	6	SMT	1	36.0
4	1005	6	ANZ	1	48.0
1	1006	5	SMT	5	125.0
2	1006	5	NRG	5	190.0
3	1006	5	ANZ	5	99.0
4	1006	6	SMT	1	36.0
5	1006	6	ANZ	1	48.0
1	1007	1	HRO	1	250.0
2	1007	2	HRO	1	126.0
3	1007	3	HSK	1	240.0
4	1007	4	HRO	1	480.0
5	1007	7	HRO	1	600.0
1	1008	8	ANZ	1	840.0
2	1008	9	ANZ	5	100.0
1	1009	1	SMT	1	450.0
1	1010	6	SMT	1	36.0
2	1010	6	ANZ	1	48.0
1	1011	5	ANZ	5	99.0
1	1012	8	ANZ	1	840.0
2	1012	9	ANZ	10	200.0
1	1013	5	ANZ	1	19.8
2	1013	6	SMT	1	36.0
3	1013	6	ANZ	1	48.0
4	1013	9	ANZ	2	40.0
1	1014	4	HSK	1	960.0
2	1014	4	HRO	1	480.0
1	1015	1	SMT	1	450.0

stock table

stock_num	manu_code	description	unit_price	unit	unit_descr
1	HRO	baseball gloves	250.00	case	10 gloves/case
1	HSK	baseball gloves	800.00	case	10 gloves/case
1	SMT	baseball gloves	450.00	case	10 gloves/case
2	HRO	baseball	126.00	case	24/case
3	HSK	baseball bat	240.00	case	12/case
4	HSK	football	960.00	case	24/case
4	HRO	football	480.00	case	24/case
5	NRG	tennis racquet	28.00	each	each
5	SMT	tennis racquet	25.00	each	each
5	ANZ	tennis racquet	19.80	each	each
6	SMT	tennis ball	36.00	case	24 cans/case
6	ANZ	tennis ball	48.00	case	24 cans/case
7	HRO	basketball	600.00	case	24/case
8	ANZ	volleyball	840.00	case	24/case
9	ANZ	volleyball net	20.00	each	each

manufact Table

manu_code	manu_name
ANZ	Anza
HSK	Husky
HRO	Hero
NRG	Norge
SMT	Smith

state Table

code	sname	code	sname
AK	Alaska	MT	Montana
AL	Alabama	NB	Nebraska
AR	Arkansas	NC	North Carolina
AZ	Arizona	ND	North Dakota
CA	California	NH	New Hampshire
CT	Connecticut	NJ	New Jersey
CO	Colorado	NM	New Mexico
DE	Delaware	NV	Nevada
FL	Florida	NY	New York
GA	Georgia	OH	Ohio
HI	Hawaii	OK	Oklahoma
IA	Iowa	OR	Oregon
ID	Idaho	PA	Pennsylvania
IL	Illinois	RI	Rhode Island
IN	Indiana	SC	South Carolina
KS	Kansas	SD	South Dakota
KY	Kentucky	TN	Tennessee
LA	Louisiana	TX	Texas
MA	Massachusetts	UT	Utah
MD	Maryland	VA	Virginia
ME	Maine	VT	Vermont
MI	Michigan	WA	Washington
MN	Minnesota	WI	Wisconsin
MO	Missouri	WV	West Virginia
MS	Mississippi	WY	Wyoming

Appendix B

Environment Variables

Environment Variables

INFORMIX-4GL makes the following assumptions about the user's environment:

- The editor used is the predominant editor for the operating system (usually **vi** for UNIX systems and **edlin** for DOS systems).
- The database worked with is in the current directory.
- If the computer is running UNIX, the program that sends files to the printer is **lp**. If the computer is running DOS, the name of the printer device is **lpt1**.
- On UNIX systems, you use **/tmp** to store temporary files. On DOS systems, you use the current directory to store temporary files.
- The INFORMIX-4GL program and its associated files are located in **/usr/informix** (on a UNIX system) or **\informix** (on a DOS system).

Setting Environment Variables

You can change any of the assumptions by setting one or more of the environment variables recognized by INFORMIX-4GL. These environment variables are listed by operating system in later sections of this appendix.

Setting Environment Variables on UNIX Systems

You may set environment variables at the system prompt or in your **.profile** (Bourne shell) or **.login** (C shell) file. If you set an environment variable at the system prompt, you will have to assign it again the next time you log onto the system. If you set a variable in your **.profile** (Bourne shell) or **.login** (C shell) file, UNIX will assign it automatically every time you log onto the system.

If you are using the Bourne shell, enter the following two commands to set the **ABCD** environment variable to *value*:

```
ABCD=value
export ABCD
```

If you are using the C shell, enter the following command to set the **ABCD** environment variable to *value*:

```
setenv ABCD value
```

Setting Environment Variables on DOS Systems

You may set environment variables at the system prompt or in your **AUTOEXEC.BAT** file. If you set an environment variable at the system prompt, you will have to assign it again the next time you start the system. If you set a variable in your **AUTOEXEC.BAT** file, DOS will assign it automatically every time you start the system.

Enter the following command to set the **ABCD** environment variable to *value*:

```
set ABCD=value
```

UNIX Environment Variables

The environment variables recognized by **INFORMIX-4GL** are as follows:

DBDELIMITER specifies the field delimiter used by the **dbload** utility in unloaded data files. The vertical bar (| = ASCII 124) is the default.

For example, if you are using the C shell, enter the following command to set the field delimiter to a plus sign:

```
setenv DBDELIMITER +
```

DBDATE specifies the format you want to use for DATE values. Through DBDATE, you can specify

- The order of the month, day, and year in a date
- Whether the year should be printed with two digits (Y2) or four digits (Y4)
- The separator between the month, day, and year

The default value for DBDATE is

```
MDY4/
```

where M represents the month, D represents the day, Y4 represents a four-digit year, and the separator is a (/) slash (mm/dd/yyyy).

Other acceptable values for the separator are a hyphen (-) or a period (.). The slash appears if you attempt to use any values other than the hyphen or period.

The separator value must always be specified last. The number of digits specified for the year must always follow the Y. To make the date appears as mm.dd.yy, you would set the DBDATE environment variable for the C shell as follows:

```
setenv DBDATE MDY2.
```

For the Bourne shell, you would use

```
DBDATE=MDY2.
export DBDATE
```

Suppose you wanted the date to appear in European format (dd-mm-yyyy). For the C shell, you would set the DBDATE environment variable as follows:

```
setenv DBDATE DMY4-
```

For the Bourne shell, you would use

```
DBDATE=DMY4-
export DBDATE
```

DBEDIT — names the text editor you want to use to create program files, form specification files, and command files from within the INFORMIX-4GL Programmer's Environment. For most systems, the default is **vi**.

DBLANG — specifies the subdirectory of **$INFORMIXDIR** that contains the message (**.iem**) files used by your program. The default is **$INFORMIXDIR/msg.**

If you want to use a message directory other than **$INFORMIXDIR/msg**, perform the following steps:

1. Use the **mkdir** command to create the appropriate subdirectory in **$INFORMIXDIR**. Set the user and group to **informix** and the access permission for this directory to 755.

2. Set the DBLANG environment variable to the new subdirectory. If you are using the Bourne shell, enter

```
DBLANG=dirname
export DBLANG
```

   If you are using the C shell, enter

```
setenv DBLANG dirname
```

3. Copy the **.iem** files to the message directory specified by **$INFORMIXDIR/$DBLANG**. All files in the message directory should have user and group **informix** and 644 access permission.

4. Run your program.

DBMONEY applies to MONEY values. It has the format

```
[front]{. | ,}[back]
```

where *front* is the optional symbol that precedes the MONEY value, the comma or the period is the optional symbol that separates the integral from the fractional part of the MONEY value, and *back* is the optional symbol that follows the MONEY value. The *front* and *back* symbols may be up to seven characters long and may contain any characters except commas or periods.

The default value for DBMONEY is

```
$.
```

where the dollar sign precedes the MONEY value, and the period (.) separates the integral from the fractional part of the MONEY value. (No *back* symbol appears.)

Suppose you wanted to represent MONEY values in Deutsche Marks. For the C shell, you would set the DBMONEY environment variable as follows:

```
setenv DBMONEY DM,
```

For the Bourne shell, you would use

```
DBMONEY=DM,
export DBMONEY
```

where DM is the currency symbol, and the comma separates the integral from the fractional part of the MONEY value.

Whenever you make a change to the *back* symbol, you must also supply the *front* symbol and the value separator (comma or period). Similarly, if you change the the value separator from a comma to a period, you must also supply the *front* symbol.

DBPATH

specifies a list of directories (in addition to the current directory) for **INFORMIX-4GL** to search for databases and associated files. If you have not selected a database, **INFORMIX-4GL** looks to DBPATH as well as the current directory to determine the list of available databases.

Once you have selected a database, INFORMIX-4GL looks first in the current directory and then in the parent directory of the directory containing the database (recall that each database is contained in a separate directory) for the associated forms, reports, and command files.

For example, if you want INFORMIX-4GL to search for database files in Kevin's and Debby's directories, as well as in your current directory, you would specify

```
DBPATH=/usr/Kevin:/usr/Debby
export DBPATH
```

Make sure you enter a colon between the directory names.

DBPRINT — specifies the print program for your computer. For most UNIX systems, the default program is **lp**.

DBTEMP — specifies the directory into which INFORMIX-4GL should place its temporary files. The default is the **/tmp** directory.

INFORMIXDIR — specifies the directory containing the INFORMIX-4GL files. The default is the **/usr/informix** directory.

DOS Environment Variables

The DOS environment variables recognized by **INFORMIX-4GL** are as follows:

DBDELIMITER specifies the field delimiter used by the **dbload** utility for unloaded data files. The vertical bar (| = ASCII 124) is the default.

To change the field delimiter to a semicolon, for example, you can enter

```
set DBDELIMITER=;
```

DBDATE specifies the format you want to use for DATE values. Through DBDATE, you can specify

- The order of the month, day, and year in a date
- Whether the year should be printed with two digits (Y2) or four digits (Y4)
- The separator between the month, day, and year

The default value for DBDATE is

```
MDY4/
```

where M represents the month, D represents the day, Y4 represents a four-digit year, and the separator is a (/) slash (mm/dd/yyyy). Other acceptable values for the separator are a hyphen (-) or a period (.). The slash appears if you attempt to use any values other than the hyphen or period.

Always specify the separator value last. Always follow the Y with the number of digits specified for the year. To make the

date appear as mm.dd.yy, set the DBDATE environment variable as follows:

```
set DBDATE=MDY2.
```

Suppose you wanted the date to appear in European format (dd-mm-yyyy). You would set the DBDATE environment variable as follows:

```
set DBDATE=DMY4-
```

DBEDIT

names the text editor you want to use to create program files, form specification files, and command files from within the INFORMIX-4GL Programmer's Environment. The default editor is **edlin**. To change the default editor, you can enter

```
set DBEDIT=program-name
```

where *program-name* is the name of the new editor.

DBLANG

specifies the subdirectory of **$INFORMIXDIR** that contains the message (**.iem**) files used by your program. The default is **$INFORMIXDIR\msg**.

If you want to use a message directory other than $INFORMIXDIR\msg, perform the following steps:

1. Use the **mkdir** command to create the appropriate subdirectory in **$INFORMIXDIR**.

2. Set the DBLANG environment variable to the new subdirectory by using the following command:

```
set DBLANG=dirname
```

3. Copy the **.iem** files to the message directory specified by **$INFORMIXDIR\$DBLANG**. All files in the message directory should have user and group **informix** and 644 access permission.

4. Run your program.

DBMONEY applies to MONEY values. It has the format

[*front*]{. | ,}[*back*]

where *front* is the optional symbol that precedes the MONEY value, the comma or the period is the optional symbol that separates the integral from the fractional part of the MONEY value, and *back* is the optional symbol that follows the MONEY value. The *front* and *back* symbols may be up to seven characters long and may contain any characters except commas or periods.

The default value for DBMONEY is

$.

where the dollar sign precedes the MONEY value, and the period separates the integral from the fractional part of the MONEY value.

If you want to represent MONEY values in deutsche marks, for example, you would set the DBMONEY variable as follows

```
set DBMONEY=DM,
```

where DM is the currency symbol, and the comma separates the integral from the fractional part of the MONEY value.

If you have specified both symbols and you make a change to either one, you must respecify the other symbol and the value separator (comma or period). Similarly, if you change the value separator from a comma to a period, you must respecify the *front* symbol as well as the *back* symbol (if you had previously specified a *back* symbol).

DBPATH

specifies a list of directories for **INFORMIX-4GL** to search for databases and associated files. If you have not selected a database, **INFORMIX-4GL** looks to DBPATH to determine the list of available databases. If you have selected a database, **INFORMIX-4GL** looks first in the current directory and then in the directory where your database is located for associated forms, reports, and command files.

For example, if you want **INFORMIX-4GL** to search for database files in Kevin's directory on C: and Debby's directory on D:, you can enter

```
set DBPATH=c:\kevin;d:\debby
```

(When you set the DBPATH variable, make sure you enter a semicolon between directory names.)

Make sure you set the DBPATH variable before you execute the **startsql** command.

DBPRINT — specifies the name of the printer device. The default is **lpt1**. To change the default to **lpt2**, for example, you can enter

```
set DBPRINT=lpt2
```

DBTEMP — specifies the directory where INFORMIX-4GL should place its temporary files. The default is the current directory.

INFORMIXDIR — specifies the directory containing the INFORMIX-4GL files. The default is **\informix**.

Appendix C

Reserved Words

Reserved Words

The following words are **INFORMIX-4GL** reserved words and cannot be used as program identifiers or database column names:

ABSOLUTE
ACCEPT
ADD
AFTER
ALL
ALLOWING
ALTER
AND
ANY
ARG_VAL
ARRAY
ARR_COUNT
ARR_CURR
AS
ASC
ASCII
AT
ATTRIBUTE
ATTRIBUTES
AUDIT
AUTO
AVERAGE
AVG
BEFORE
BEGIN
BETWEEN
BLINK
BLUE
BOLD
BORDER
BOTTOM
BREAK
BY
CALL
CASE
CHAR
CHECK
CLEAR
CLIPPED
CLOSE
CLUSTER
COLUMN
COLUMNS
COMMAND
COMMENT
COMMIT
COMPOSITES
CONNECT
CONSTRUCT
CONTINUE
CONTROL
CORRECT
COUNT
CREATE
CURRENT
CURSOR
CYAN
DATABASE
DATE
DAY
DBA
DECIMAL
DECLARE
DEFAULT
DEFAULTS
DEFER
DEFINE
DELETE
DELIMITER
DESC
DESCRIBE
DIM
DISPLAY
DISPLAYONLY
DISTINCT
DOS
DOUBLE
DOWN
DOWNSHIFT
DROP
ELSE
END
ENTRY
ERROR
ERRORLOG
ERR_GET
ERR_PRINT
ERR_QUIT
ESC
ESCAPE
EVERY
EXCLUSIVE
EXECUTE
EXISTS
EXIT
EXITNOW
EXTERNAL
FALSE
FETCH
FIELD
FILE
FINISH

FIRST
FLOAT
FLUSH
FOR
FOREACH
FORM
FORM4GL
FORMAT
FROM
FUNCTION
GLOBALS
GOTO
GRANT
GREEN
GROUP
HAVING
HEADINGS
HELP
IF
IN
INCLUDE
INCORRECT
INDEX
INITIALIZE
INPUT
INSERT
INSTRUCTIONS
INTEGER
INTERRUPT
INTO
INVISIBLE
IS
JOINING
KEY
LABEL
LAST
LENGTH
LET
LEVEL
LIKE
LINE
LINENO
LINES
LOCK
LOG
LONG
MAGENTA
MAIN
MARGIN
MASTER
MATCHES
MAX
MDY
MENU
MESSAGE
MIN
MODE
MODIFY
MONEY
MONTH
NAME
NEED
NETAUF
NEXT
NEXTFIELD
NO
NOENTRY
NORMAL
NOT
NOTFOUND
NOUPDATE
NULL
NUM_ARGS
OF
ON
OPEN
OPTION
OPTIONS
OR
ORDER
OTHERWISE
OUTER
OUTPUT
PAGE
PAGENO
PAUSE
PERCENT
PICTURE
PIPE
PREPARE
PREVIOUS
PRINT
PRINTER
PRIOR
PRIVILEGES
PROGRAM
PROMPT
PUBLIC
PUT
QUERY
QUERYCLEAR
QUIT

RECOVER
RED
REGISTER
RELATIVE
REMOVE
RENAME
REPORT
REQUIRED
RESLOG
RESOURCE
RETURN
RETURNING
REVERSE
REVOKE
RIGHT
ROLLBACK
ROLLFORWARD
ROW
ROWID
ROWS
RUN
SCREEN
SCROLL
SCR_LINE
SELECT
SERIAL
SET
SET_COUNT
SHARE
SKIP
SLEEP
SMALLINT
SOME
SPACE
SPACES
SQLAWARN
SQLCA
SQLCODE
SQLERRD
START
STARTLOG
STATISTICS
STEP
STOP
SUM
SYNONYM
SYSTABLES
TABLE
TABLES
TEMP
THEN
THROUGH
THRU
TIME
TO
TODAY
TOP
TRAILER
TRUE
TYPE
TYPEDEF
UNDERLINE
UNIQUE
UNIX
UNLOAD
UNLOCK
UP
UPDATE
UPSHIFT
USER
USING
VALIDATE
VALUE
VALUES
VERIFY
VIEW
WAIT
WAITING
WARNING
WEEKDAY
WHEN
WHENEVER
WHERE
WHILE
WHITE
WINDOW
WITH
WITHOUT
WORK
WRAP
YEAR
YELLOW
ZEROFILL

Appendix D

Sample Reports

report1.4gl

```
DATABASE stores

MAIN

        DEFINE p_customer RECORD LIKE customer.*

        DECLARE q_curs CURSOR FOR
                SELECT * FROM customer

        START REPORT cust_list

        FOREACH q_curs INTO p_customer.*

                OUTPUT TO REPORT cust_list (p_customer.*)

        END FOREACH

        FINISH REPORT cust_list

END MAIN

REPORT cust_list (r_customer)

        DEFINE r_customer RECORD LIKE customer.*

        FORMAT EVERY ROW

END REPORT
```

report2.4gl

```
DATABASE stores

MAIN

        DEFINE p_customer RECORD LIKE customer.*

        DECLARE q_curs CURSOR FOR
                SELECT fname, lname, company
                        FROM customer

        START REPORT cust_list

        FOREACH q_curs INTO p_customer.fname,
                                p_customer.lname,
                                p_customer.company

                OUTPUT TO REPORT cust_list (p_customer.fname,
                                            p_customer.lname,
                                            p_customer.company)

        END FOREACH

        FINISH REPORT cust_list

END MAIN

REPORT cust_list (fname, lname, company)

        DEFINE fname, lname         CHAR(15),
               company  CHAR(20)

        FORMAT

                PAGE HEADER

                        PRINT COLUMN 25, "CUSTOMER LIST"
                        SKIP 1 LINE
                        PRINT COLUMN 24, "August 26, 1986"
                        SKIP 2 LINES
                        PRINT "Last Name",
                                COLUMN 26, "First Name",
                                COLUMN 50, "Company"
                        SKIP 1 LINE

                ON EVERY ROW
                        PRINT lname,
                                COLUMN 26, fname,
                                COLUMN 50, company


                PAGE TRAILER
                        PRINT "Confidential Information",
                                COLUMN 55, PAGENO USING "Page ##"

END REPORT
```

report3.4gl

```
DATABASE stores

MAIN

        DEFINE p_customer RECORD LIKE customer.*

        DECLARE q_curs CURSOR FOR
                SELECT lname, company, city, state
                        FROM customer
                        ORDER BY city

        START REPORT cust_list

        FOREACH q_curs INTO p_customer.lname,
                                p_customer.company,
                                p_customer.city,
                                p_customer.state

                OUTPUT TO REPORT cust_list (p_customer.lname,
                                        p_customer.company,
                                        p_customer.city,
                                        p_customer.state)

        END FOREACH

        FINISH REPORT cust_list

END MAIN

REPORT cust_list (lname, company, city, state)

        DEFINE lname      CHAR(15),
               company    CHAR(20),
               city       CHAR(15),
               state      CHAR(2)

        FORMAT

                ON EVERY ROW
                        PRINT lname,
                                COLUMN 18, company,
                                COLUMN 40, city,
                                COLUMN 60, state

                AFTER GROUP OF city
                        SKIP 2 LINES

END REPORT
```

report4.4gl

```
DATABASE stores

MAIN

        DEFINE p_customer RECORD LIKE customer.*

        DECLARE q_curs CURSOR FOR
                SELECT lname, company, city, state
                        FROM customer

        START REPORT cust_list

        FOREACH q_curs INTO p_customer.lname,
                                p_customer.company,
                                p_customer.city,
                                p_customer.state

                OUTPUT TO REPORT cust_list (p_customer.lname,
                                        p_customer.company,
                                        p_customer.city,
                                        p_customer.state)

        END FOREACH

        FINISH REPORT cust_list

END MAIN

REPORT cust_list (lname, company, city, state)

        DEFINE lname      CHAR(15),
               company    CHAR(20),
               city       CHAR(15),
               state      CHAR(2)

        ORDER BY city

        FORMAT

                PAGE HEADER

                        PRINT COLUMN 25, "CUSTOMER LIST"
                        SKIP 1 LINE
                        PRINT COLUMN 24, "August 26, 1986"
                        SKIP 2 LINES
                        PRINT "Last Name",
                                COLUMN 18, "Company",
                                COLUMN 40, "City",
                                COLUMN 60, "State"
                        SKIP 2 LINES

                ON EVERY ROW
                        PRINT lname,
                                COLUMN 18, company,
                                COLUMN 40, city,
                                COLUMN 60, state

                AFTER GROUP OF city
                        SKIP 2 LINES

                PAGE TRAILER
                        PRINT "Confidential Information",
                                COLUMN 55, PAGENO USING "Page ##"

END REPORT
```

report5.4gl

```
DATABASE stores

MAIN

        DEFINE p_stock RECORD LIKE stock.*

        DECLARE q_curs CURSOR FOR
                SELECT stock_num, manu_code,
                        description, unit, unit_price
                FROM stock

        START REPORT qty_rep

        FOREACH q_curs INTO p_stock.stock_num,
                                p_stock.manu_code,
                                p_stock.description,
                                p_stock.unit,
                                p_stock.unit_price

                OUTPUT TO REPORT qty_rep (p_stock.stock_num,
                                                p_stock.manu_code,
                                                p_stock.description,
                                                p_stock.unit,
                                                p_stock.unit_price)

        END FOREACH

        FINISH REPORT qty_rep

END MAIN


REPORT qty_rep (stock_num, manu_code, description, unit, unit_price)

        DEFINE stock_num        LIKE stock.stock_num,
                manu_code       LIKE stock.manu_code,
                description     LIKE stock.description,
                unit            LIKE stock.unit,
                unit_price      LIKE stock.unit_price


        FORMAT
                PAGE HEADER
                        PRINT "Stock Number",
                                column 15, "Manufacturer",
                                column 30, "Description",
                                column 50, "Unit",
                                column 57, "New Unit Price"
                        SKIP 2 lines

                ON EVERY ROW
                        PRINT stock_num,
                                column 15, manu_code,
                                column 30, description,
                                column 50, unit,
                                column 57, unit_price * 1.15 USING "$$$$$.$$"


END REPORT
```

report6.4gl

```
DATABASE stores

MAIN

        DEFINE p_items RECORD LIKE items.*,
               p_stock RECORD LIKE stock.*

        DECLARE q_curs CURSOR FOR
                SELECT items.stock_num, items.manu_code,
                       quantity, description, unit
                FROM items, stock
                WHERE items.stock_num = stock.stock_num
                       AND items.manu_code = stock.manu_code
                       AND items.stock_num = 5

        START REPORT qty_rep2

        FOREACH q_curs INTO p_items.stock_num,
                                p_items.manu_code,
                                p_items.quantity,
                                p_stock.description,
                                p_stock.unit

                OUTPUT TO REPORT qty_rep2 (p_items.stock_num,
                                                p_items.manu_code,
                                                p_items.quantity,
                                                p_stock.description,
                                                p_stock.unit)

        END FOREACH

        FINISH REPORT qty_rep2

END MAIN


REPORT qty_rep2 (stock_num, manu_code, quantity, description, unit)

        DEFINE stock_num        LIKE items.stock_num,
               manu_code        LIKE items.manu_code,
               quantity         LIKE items.quantity,
               description      LIKE stock.description,
               unit             LIKE stock.unit

        ORDER BY stock_num

        FORMAT
                PAGE HEADER
                        PRINT "Stock Number",
                                COLUMN 15, "Manufacturer",
                                COLUMN 30, "Description",
                                COLUMN 50, "Unit",
                                COLUMN 60, "Quantity"
                        SKIP 2 LINES

                ON EVERY ROW
                        PRINT stock_num,
                                COLUMN 15, manu_code,
                                COLUMN 30, description,
                                COLUMN 50, unit,
                                COLUMN 60, quantity

                ON LAST ROW
                        SKIP 1 LINE
                        PRINT COLUMN 20, "Total Quantity on Order: ",
                                COLUMN 55, SUM(quantity) USING "###"
                        SKIP 2 LINES
                        PRINT COLUMN 20, "Total Number of Orders: ",
                                COLUMN 55, COUNT(*) USING "###"

END REPORT
```

report7.4gl

```
DATABASE stores

MAIN

        DEFINE p_items RECORD LIKE items.*,
               p_stock RECORD LIKE stock.*

        DECLARE q_curs CURSOR FOR
               SELECT items.stock_num, items.manu_code,
                      quantity, description, unit
               FROM items, stock
               WHERE items.stock_num = stock.stock_num
                      AND items.manu_code = stock.manu_code
                      ORDER BY items.stock_num

        START REPORT qty_rep3

        FOREACH q_curs INTO p_items.stock_num,
                                p_items.manu_code,
                                p_items.quantity,
                                p_stock.description,
                                p_stock.unit

               OUTPUT TO REPORT qty_rep3 (p_items.stock_num,
                                                p_items.manu_code,
                                                p_items.quantity,
                                                p_stock.description,
                                                p_stock.unit)

        END FOREACH

        FINISH REPORT qty_rep3

END MAIN


REPORT qty_rep3 (stock_num, manu_code, quantity, description, unit)

        DEFINE stock_num        LIKE items.stock_num,
               manu_code        LIKE items.manu_code,
               quantity         LIKE items.quantity,
               description      LIKE stock.description,
               unit             LIKE stock.unit

        FORMAT
               PAGE HEADER
                      PRINT "Stock Number",
                              COLUMN 15, "Manufacturer",
                              COLUMN 30, "Description",
                              COLUMN 50, "Unit",
                              COLUMN 60, "Quantity"
                      SKIP 2 LINES

               ON EVERY ROW
                      PRINT stock_num,
                              COLUMN 15, manu_code,
                              COLUMN 30, description,
                              COLUMN 50, unit,
                              COLUMN 60, quantity

               AFTER GROUP OF stock_num
                      SKIP 1 LINE
                      PRINT COLUMN 20, "Total Quantity on Order: ",
                              COLUMN 55, GROUP SUM(quantity) USING "###"
                      SKIP 2 LINES

END REPORT
```

Index

This index covers both the User Guide and the Reference Manual. Page numbers that end in U can be found in the User Guide, while those that end in R are in the Reference Manual.

Access privileges
    database privileges 7-115R
    for a synonym 7-47R
    for database administrator 14-7U
    on views 2-68R
    removing 7-195R
    summary of 2-49R
    table privileges 14-8U, 7-114R
    user access to the database 14-6U
    using UNIX permissions for a database 7-42R
ACE reports with INFORMIX-4GL 9-41U
ADD keyword, ALTER TABLE statement 7-13R
AFTER DELETE clause 11-46U
AFTER FIELD clause
    guidelines for using 11-29U
    with INPUT ARRAY 13-45U
AFTER GROUP OF control block 4-29R
AFTER INSERT clause 11-38U
Aggregate function
    AVG 4-28U, 7-265R
    COUNT 4-28U, 7-265R
    in a report 9-32U, 4-30R, 4-56R
    in a SELECT statement 4-29U, 7-265R
    MAX 4-28U, 7-265R
    MIN 4-29U, 7-266R
    PERCENT 9-32U
    SUM 4-28U, 7-265R
    with DISTINCT 7-266R
    with NULL values 2-59R
ALL keyword, GRANT statement 7-115R
ALTER INDEX statement
    definition of 2-12R
    syntax 7-11R
Alternation operator 12-11U
ALTER TABLE statement
    definition of 14-12U, 2-11R, 7-13R
    guidelines for using 7-14R
    MODIFY 14-14U
    with ADD 14-13U
    with BEFORE 14-13U
    with DROP 14-14U

AND keyword 4-31U
arg_val function 5-6R
Arithmetic operators
in a report 9-27U
in a SELECT clause 4-28U
in expressions 3-14U
Array
definition of 1-12R
displaying a program array 11-26U, 11-51U
function that renumbers array items 11-41U
of records 11-13U
program array 11-12U
screen array 11-8U
screen record, corresponding to a screen array 11-10U
arr_count function
definition of 11-23U
examples 13-49U, 13-54U
guidelines for using 5-8R
with INPUT ARRAY 7-137R
arr_curr function
definition of 11-23U
examples 13-50U, 13-54U
guidelines for using 5-10R
with INPUT ARRAY 7-137R
Assignment statements
definition of 1-22R
INITIALIZE 7-120R
LET 3-12U, 7-146R
Asterisk (*) notation
guidelines for using 1-19R
in a SELECT clause 7-230R
pattern matching with 4-33U
querying the database 12-10U
representing all columns 3-29U
AT clause
with OPEN WINDOW 13-13U
ATTRIBUTE clause
BORDER keyword 13-21U
changing default values 13-20U
FIRST keyword 13-20U
LAST keyword 13-20U
local to current window 13-24U
setting the form line 13-25U
with OPEN WINDOW 13-12U, 13-19U
ATTRIBUTES section of form specification
assigning attributes 6-20U

ATTRIBUTES section of form specification (Continued)
definition of 6-18U, 3-14R
form-only field 3-14R
layout 6-19U
linking field names and field tags 6-19U, 3-14R
syntax recognized by FORM4GL 3-20R
table of attributes 6-21U
using fields linked to columns 3-15R
using form-only fields 3-17R
using multiple-column fields 3-19R
Attribute types
AUTONEXT 6-28U, 3-21R
COMMENTS 6-28U, 3-23R
DEFAULT 6-26U, 3-25R
default screen 3-53R
DISPLAY LIKE 6-21U, 3-27R
DOWNSHIFT 6-27U, 3-29R
FORMAT 6-21U, 3-30R
INCLUDE 6-25U, 3-33R
NOENTRY 6-23U, 3-35R
PICTURE 6-29U, 3-37R
REQUIRED 6-23U, 3-39R
REVERSE 6-24U, 3-41R
UPSHIFT 6-27U, 3-42R
VALIDATE LIKE 6-22U, 3-44R
VERIFY 6-22U, 3-46R
Audit trails
RDSQL 7-39R, 7-85R, 7-185R
AUTONEXT attribute 6-28U, 3-21R
AVG aggregate function 4-28U, 9-32U, 4-57R, 7-265R
BEFORE DELETE clause 11-25U
BEFORE FIELD clause
guidelines for using 11-27U
with INPUT ARRAY 13-44U
BEFORE GROUP OF control block
definition of 4-32R
guidelines for using 4-33R
BEFORE INSERT clause 11-36U
BEFORE keyword, ALTER TABLE statement 7-13R
BEFORE ROW clause 11-25U
BEGIN WORK statement
definition of 2-43R
executing transactions 14-16U
syntax and notes 7-16R
Binding to database and forms 1-18R, 3-15R

Boolean expression
definition of 1-16R
examples 3-18U
with IF 4-14U
with NULL values 3-18U, 2-60R
with WHERE 4-31U
Bordered window
graphics characters used 13-21U
how displayed on screen 13-21U
position on screen 13-22U
BORDER keyword
ATTRIBUTE clause 13-21U
BOTTOM MARGIN statement 4-18R, 4-44R
Built-in functions 1-32R
BY NAME keywords
with DISPLAY 7-19U, 7-74R
with INPUT 7-17U, 7-124R
CALL statement
definition of 5-7U, 1-24R
passing parameters with 5-17U
RETURNING clause 5-21U
syntax and notes 7-17R
CASE statement
definition of 1-24R
examples 13-78U
EXIT CASE 7-96R
guidelines for using 7-20R
syntax 7-19R
C functions
calling 1-57R, 7-17R
use with INFORMIX-4GL functions 1-58R
CHAR data type
acceptable values 7-50R
subscripting a CHAR column 3-16R
subscripting columns 7-227R
with database columns 2-22U, 2-7R
with display fields 6-16U
with variables 3-9U, 1-11R
CLEAR statement
definition of 1-25R
forms of 7-20U
syntax and notes 7-22R
WINDOW keyword 13-33U, 13-63U, 13-64U, 7-22R
CLIPPED keyword
definition of 3-24U
in a string expression 3-16U

CLIPPED keyword (Continued)
with DISPLAY 3-24U
CLOSE DATABASE statement
definition of 14-11U, 2-10R
syntax and notes 7-26R
CLOSE FORM statement
closing an opened form 7-162R
definition of 1-26R
syntax and notes 7-27R
CLOSE statement
and the SQLCA record 2-29R
and the status variable 2-29R
syntax and notes 7-24R
with an INSERT cursor 2-26R, 7-24R
with a SELECT cursor 4-21U, 2-19R, 7-24R
CLOSE WINDOW statement
definition of 1-26R
guidelines for using 7-29R
syntax and notes 13-26U, 13-65U
Clustering
ALTER INDEX statement 7-11R
CREATE INDEX statement 7-44R
definition of 2-53R
Column
adding 14-12U, 7-13R
changing column values 7-212R
changing data type 14-14U, 7-13R
CHAR 2-22U
DATE 2-26U
DECIMAL 2-23U
definition of 2-8U
designated as NOT NULL 7-51R
determining length 2-29U
FLOAT 2-24U
indexed 2-17U
INTEGER 2-24U
joining 2-10U
lengths of DECIMAL and MONEY columns 2-29U
modifying 7-13R
MONEY 2-27U
naming conventions 2-21U, 2-6R, 7-51R
NULL value in 2-58R
removing 14-14U, 7-13R
renaming 14-12U, 7-187R
SERIAL 2-25U
SMALLFLOAT 2-24U

Column (Continued)
SMALLINT 2-24U
virtual 2-66R, 7-56R
COMMAND clause
CONTINUE MENU 8-15U
designating a help message 8-27U
EXIT MENU 8-16U
NEXT OPTION 8-15U
syntax 8-12U
using statements within 8-14U
COMMENTS attribute 6-28U, 3-23R
COMMIT WORK statement
definition of 2-43R
ending transactions 14-16U
syntax and notes 7-31R
Compile-time errors 10-6U
Compiling, at operating system level 1-63R
Conditional statements
CASE 1-24R, 7-19R
IF 1-24R, 7-118R
CONNECT access privilege 2-49R, 7-116R
Constant
date 1-8R
floating numeric 1-8R
integer 1-7R
string 1-7R
CONSTRUCT statement
definition of 12-14U, 1-26R
guidelines for using 7-35R
symbols recognized 7-33R
syntax 7-32R
wildcard characters 7-34R
CONTINUE statement
definition of 1-24R
FOR 11-16U, 7-105R
FOREACH 4-15U, 7-107R
forms of 7-38R
MENU 8-15U, 7-155R
syntax 7-38R
COUNT aggregate function 4-28U, 9-32U, 9-34U, 4-57R, 7-265R
CREATE AUDIT statement 7-39R
CREATE DATABASE statement
current database 7-41R
database naming conventions 2-20U
definition of 2-10R, 7-41R
syntax 2-19U

CREATE DATABASE statement (Continued)
system catalogs 7-41R
WITH LOG IN 14-15U, 2-44R, 7-42R
CREATE INDEX statement
definition of 2-12R
index naming conventions 2-34U
syntax 2-33U, 7-44R
with ASC 7-45R
with DESC 2-35U, 7-45R
with UNIQUE and DISTINCT 2-35U, 7-44R
CREATE SYNONYM statement
definition of 2-12R, 7-47R
guidelines for using 7-47R
CREATE TABLE statement
CHAR data type 2-22U
conventions for naming columns 2-21U
DATE data type 2-26U
DECIMAL data type 2-24U
definition of 2-11R
FLOAT data type 2-24U
guidelines for using 7-51R
INTEGER data type 2-24U
MONEY data type 2-27U
NOT NULL clause 2-58R, 7-51R
SERIAL data type 2-25U
SMALLFLOAT data type 2-24U
SMALLINT data type 2-24U
syntax 2-20U, 7-49R
TEMP 7-51R
CREATE VIEW statement
definition of 14-19U, 2-11R, 2-65R
guidelines for using 7-56R
syntax 7-55R
WITH CHECK OPTION 2-67R, 7-57R
Current database
CLOSE DATABASE statement 7-26R
closing 7-26R
creating 7-41R
DATABASE statement 7-60R
definition of 2-10R
selecting 7-60R
Current window
definition of 13-60U
CURRENT WINDOW statement
definition of 1-26R
guidelines for using 7-58R

CURRENT WINDOW statement (Continued)
syntax and notes 13-60U
Cursor
advancing 4-18U, 4-24U, 7-98R
closing 4-21U, 7-24R
declaring 7-62R
definition of 4-10U
insert 7-62R
naming, with DECLARE 4-10U
non-scrolling 4-10U, 2-15R
opening 4-17U, 7-159R
position after DELETE 2-23R
position after UPDATE 2-25R
scrolling 4-10U, 2-15R
select 7-62R
UPDATE statement 7-213R
with FOREACH 4-11U
with INSERT 2-26R
with SELECT 2-14R, 7-228R
Cursor movement
as determined by a SCREEN RECORD 6-34U
as determined by INPUT 7-125R
during data entry 7-17U
in a menu 8-6U
in a screen array 11-21U, 7-78R
in a screen form 6-7U
customer form 6-6U
customer table
stores database Introduction-10R
Database
access privileges to 2-49R
binding, to forms 1-18R
closing 14-11U, 7-26R
CREATE DATABASE statement 2-19U
creating 7-49R
creating views 14-19U
database subdirectory 2-6R, 7-41R
data manipulation statements 2-13R
definition of 2-6U
granting user access 14-6U
indexes in 2-17U
naming conventions 2-20U, 2-6R, 7-42R
recovering 2-46R
relational 2-6R
removing 14-11U, 7-86R
removing indexes 14-10U

Database (Continued)
removing tables 14-10U
renaming tables 14-11U
revoking user access 14-6U
sample program that creates a database 2-39U
stores demonstration Preface-17U
system catalogs 2-6R
tables in 2-7U, 2-6R
transactions 2-43R
with a transaction log 14-15U
Database administrator (DBA) access privileges 14-7U, 2-49R, 7-116R
DATABASE section of form specification
definition of 3-7R
format 6-13U
WITHOUT NULL INPUT 3-8R, 3-25R
DATABASE statement
definition of 2-10R
EXCLUSIVE 7-61R
guidelines for using 3-6U
selecting the current database 7-60R
syntax and notes 7-60R
Data conversion
in an INSERT statement 7-142R
in an UPDATE statement 7-215R
in expressions 1-12R
Data manipulation statements
DELETE 2-13R, 7-70R
INSERT 2-13R, 7-140R
SELECT 2-13R, 7-204R, 7-224R
UPDATE 2-13R, 7-212R
Data type
ARRAY 1-12R
changing 14-14U, 7-13R
CHAR 2-22U, 1-11R, 2-7R
conversion 3-19U, 1-12R
DATE 2-26U, 1-11R, 2-9R
DECIMAL 2-23U, 1-10R, 1-13R, 2-7R
FLOAT 2-24U, 1-10R, 2-8R
floating decimal point in 2-8R
INTEGER 2-24U, 1-9R, 2-7R
MONEY 2-27U, 1-10R, 2-8R
numeric data type table 2-23U
of columns 2-7R
of parameters 5-17U
of variables 3-8U
of view columns 7-56R

Data type (Continued)
RECORD 1-11R
SERIAL 2-25U, 2-9R
SMALLFLOAT 2-24U, 1-10R
SMALLINT 2-24U, 1-9R, 2-7R
space requirements 7-52R
Data validation
how to set up 1-31R
using views 2-68R
VALIDATE statement 1-31R
Date constant 1-8R
DATE data type
acceptable values 7-51R
format for display fields 3-31R, 7-52R
with database columns 2-26U, 2-9R
with display fields 6-16U
with variables 3-9U, 1-11R
DATE expression, formatting 1-48R
DATE function of a SELECT statement 7-267R
DAY function of a SELECT statement 7-268R
DBPRINT variable 4-11R
DBUPDATE utility 2-59R
DECIMAL data type
acceptable values 7-50R
floating decimal point in 2-24U, 1-10R, 2-8R
scale and precision 1-10R, 1-13R, 1-14R
with database columns 2-23U, 2-7R
with display fields 6-16U
with FORMAT attribute 3-30R
with variables 3-9U, 1-10R
DECLARE statement
assigning a cursor 4-10U
FOR UPDATE 7-63R, 7-71R
guidelines for using 2-30R, 7-63R
syntax 7-62R
the SCROLL option 7-63R
with an INSERT cursor 2-26R, 7-63R
with a prepared INSERT statement 2-40R
with a prepared SELECT statement 2-38R
with a SELECT cursor 4-10U, 2-15R, 7-63R
DEFAULT attribute
definition of 6-26U, 3-25R
guidelines for using 6-26U
TODAY 6-26U, 3-26R
with WITHOUT NULL INPUT database 3-25R

Default form specification file
creating at operating system level 3-58R
definition of 6-36U
Default screen record 6-32U
DEFER statement
definition of 1-28R
forms of 10-16U
global flags set 1-31R, 7-66R
INTERRUPT 10-17U, 7-67R
QUIT 7-67R
syntax and notes 7-66R
DEFINE section of REPORT statement
defining variables in a report 9-9U
using an argument list 4-8R
DEFINE statement
DEFINE RECORD 7-69R
defining a program array 11-12U
defining a record 4-6U
defining global variables 7-111R
defining variables 3-7U
definition of 1-21R
examples 3-10U, 4-7U
in a function 5-16U
LIKE keyword 3-10U, 7-68R
syntax 3-7U, 7-68R
DELETE keyword
GRANT statement 7-114R
DELETE statement
examples 3-38U
syntax 3-38U, 7-70R
WHERE CURRENT OF 2-23R, 7-71R
Delimiter
changing 6-34U
use of 6-15U
DELIMITERS statement 6-34U, 3-47R
Demonstration database
copying Preface-18U
description of Introduction-10R
introduction to Preface-17U
restoring the original database Preface-19U
DISPLAY ARRAY statement
arr_curr function 5-10R
ATTRIBUTE clause 7-79R
definition of 11-51U, 1-27R, 7-77R
displaying rows in a program array 11-51U
EXIT DISPLAY 7-96R

DISPLAY ARRAY statement (Continued)
guidelines for using 7-78R
set_count function 5-25R
with ON KEY 7-80R
Display field
as created by FORM4GL 3-8R
default field widths 6-16U
definition of 6-6U, 6-15U
determining field widths 3-11R
dividing long CHAR columns 3-16R
field tag 6-15U, 3-10R
form-only field 3-14R, 3-16R
format of 6-15U, 3-10R
guidelines for using 3-10R
labels for 6-15U
multiple column 3-19R
querying with relational operators 12-9U
querying with wildcard characters 12-10U
specifying search criteria 12-7U
verifying field widths 3-11R
DISPLAY FORM statement
definition of 7-7U, 1-26R
syntax and guidelines 7-83R
DISPLAY LIKE attribute 6-21U, 3-27R
DISPLAY statement
AT 7-10U
ATTRIBUTE 7-75R
BY NAME 7-19U, 7-74R
CLIPPED 3-24U
definition of 1-26R
displaying values on the screen form 7-18U
formatting 3-23U
guidelines for using 7-74R
syntax 3-23U, 7-73R
TO 7-18U
with SELECT 3-32U
DOWNSHIFT attribute 6-27U, 3-29R
downshift function 5-12R
DROP AUDIT statement 7-85R
DROP DATABASE statement
definition of 14-11U, 2-11R
syntax and notes 7-86R
DROP INDEX statement
definition of 14-10U, 2-12R
syntax and notes 7-88R

DROP keyword, ALTER TABLE statement 7-14R
DROP SYNONYM statement
    definition of 2-12R
    syntax and notes 7-89R
DROP TABLE statement
    definition of 14-10U, 2-11R
    syntax and notes 7-90R
DROP VIEW statement
    definition of 14-20U, 2-11R, 2-65R
    syntax and notes 7-91R
Environment variable
    DBEDIT 3-58R
    DBPATH 7-60R
    DBPRINT 4-11R
Error handling
    built-in functions 1-30R
    creating an error log 10-10U, 5-28R
    defining an alternate interrupt key 10-23U
    displaying error messages 10-7U, 7-92R
    exception handling 1-31R
    INTERRUPT key 10-16U
    logging programmer-defined error messages 10-11U
    non-fatal errors 10-14U, 10-15U
    overview of 1-28R
    runtime errors 10-6U
    SQLCA global record 2-72R
    trapping errors 10-7U, 1-29R, 7-220R
    trapping interrupts 10-17U, 7-66R
    trapping warnings 7-219R
    with status variable 1-29R
errorlog function 10-11U, 1-30R, 5-16R
ERROR statement
    definition of 1-25R
    displaying the error line 7-92R
    syntax and notes 7-92R
err_get function
    definition of 1-30R
    syntax and description 5-13R
err_print function
    definition of 1-30R
    syntax and description 5-14R
err_quit function
    definition of 1-30R
    syntax and description 5-15R
EVERY ROW statement 4-24R

EXCLUSIVE keyword, LOCK TABLE statement 7-148R
EXECUTE statement
guidelines for using 2-30R, 2-35R
syntax and notes 7-94R
USING 2-36R, 7-94R
EXIT statement
definition of 1-24R
FOR 11-16U, 7-105R
FOREACH 4-16U, 7-107R
MENU 8-16U, 7-152R
PROGRAM 7-97R
syntax and notes 7-96R
Expression
Boolean 3-18U
data conversion 1-12R
definition of 1-14R
functions in 5-25U
in a report 4-56R
in a SELECT statement 7-226R
LINE NO 4-60R
NULL values 2-59R
numeric 3-12U, 1-15R
PAGENO 4-61R
string 3-15U, 1-15R
TIME 4-62R
using in statements 1-17R
FETCH statement
default condition 7-99R
examples 8-46U
guidelines for using 4-24U, 2-18R
INTO clause 7-99R
NOTFOUND code 4-20U, 4-24U, 8-45U, 7-100R
syntax and notes 4-18U, 7-98R
with SELECT 7-232R
Field tag
definition of 6-17U
generated by FORM4GL 6-17U, 3-11R
in the ATTRIBUTES section 6-19U
in the SCREEN section 6-14U, 3-10R
naming conventions 6-17U
File extensions
.dat 2-6R
.dbs 2-6R
.idx 2-6R
Fill character
asterisks 9-30U

Fill character (Continued)
definition of 9-29U
dollar sign 9-31U
pound sign 9-29U
table of 9-30U
FINISH REPORT statement 9-12U, 1-28R
FIRST keyword
ATTRIBUTE clause 13-20U
FIRST PAGE HEADER control block 4-35R, 4-41R
FLOAT data type
acceptable values 7-50R
with database columns 2-24U, 2-8R
with display fields 6-16U
with FORMAT attribute 3-30R
with variables 3-9U, 1-10R
Floating numeric constant 1-8R
FLUSH statement
and the SQLCA record 2-29R
and the status variable 2-29R
guidelines for using 2-26R, 7-102R
FOREACH statement
CONTINUE FOREACH 4-15U, 7-38R, 7-107R
definition of 1-24R
examples 4-13U, 2-17R
EXIT FOREACH 4-16U, 7-96R, 7-107R
guidelines for using 4-12U, 7-107R
INTO clause 7-107R
syntax 4-11U, 7-106R
Form-only field
form-only table 3-18R
guidelines for using 3-17R
syntax 3-16R
FORM4GL
attribute syntax 3-20R
command line syntax 3-58R, 3-59R
creating a default form specification file 6-36U, 3-57R
default field tags generated 6-17U, 3-11R
default screen records generated 6-32U
definition of 3-5R
display field generated 3-8R
file extensions created by 6-11U
screen form created 3-57R
subscripting a CHAR column 3-11R
using defaults in syscolval and syscolatt 14-20U, 3-15R
verifying field widths 3-11R

FORMAT attribute
definition of 3-30R
guidelines for using 3-30R
FORMAT section of REPORT statement
AFTER GROUP OF 9-22U, 4-29R
aggregate functions 4-57R
arithmetic operators 9-27U
BEFORE GROUP OF 9-22U, 4-32R
control blocks 4-28R
definition of 9-11U, 4-22R
EVERY ROW 9-14U, 4-14R, 4-24R
FIRST PAGE HEADER control block 9-16U, 4-35R
format keywords 9-18U
LINENO 4-60R
NEED 4-47R
ON EVERY ROW control block 9-20U, 4-37R
ON LAST ROW control block 9-33U, 4-39R
PAGE HEADER control block 9-17U, 9-19U, 4-41R
PAGENO 4-61R
PAGE TRAILER control block 9-20U, 4-44R
PAUSE 4-48R
PRINT 9-17U, 4-50R
PRINT FILE 4-53R
SKIP 9-18U, 4-54R
TIME 4-62R
Formatting a report
automatic page numbering 9-19U
default report format 9-14U, 4-24R
formatting numbers 9-29U
grouping data 9-22U
page headers and trailers 9-16U, 4-35R, 4-41R, 4-44R
printing column headings 9-19U
printing rows of data 9-20U
setting margins 4-12R, 4-14R, 4-16R, 4-18R
setting page length 4-20R
skipping to top of page 4-54R
starting on a new page 4-47R
Formatting DATE expressions 1-48R
Formatting numeric expressions 1-46R
Form specification file, sections of
ATTRIBUTES 6-18U, 3-6R, 3-14R
DATABASE 6-13U, 3-6R, 3-7R
INSTRUCTIONS 6-31U, 3-6R, 3-47R
SCREEN 6-14U, 3-6R, 3-8R
TABLES 6-17U, 3-6R, 3-12R

Form specification file, using
correcting errors 6-41U, 6-44U
creating at operating system level 3-58R
creating with a text editor 6-43U
default form specification file 6-36U
definition of 6-9U, 3-5R
multiple-tables in 11-7U
overview 6-12U
structure 3-6R
FOR statement
CONTINUE FOR 7-38R, 7-105R
definition of 11-15U, 1-24R
EXIT FOR 7-96R, 7-105R
guidelines for using 7-105R
STEP 7-105R
syntax 7-104R
FOR UPDATE clause
DECLARE statement 7-63R
Fourth-generation languages 1-6U
FROM clause
OUTER keyword 2-70R, 7-234R
with SELECT 7-234R
Function
aggregate 4-28U, 9-32U
ASCII 1-33R
built-in, error handling 1-30R
built-in, for working with arrays 11-23U
calling a function 5-7U, 7-17R, 7-110R
CLIPPED 1-35R
COLUMN 1-37R
DATE 1-39R
DATE() 1-40R
DAY() 1-41R
definition of 5-6U
for working with C functions 1-58R
function library 5-5R
GROUP 9-36U
in an expression 5-25U
MDY() 1-42R
MONTH() 1-43R
SPACES 1-44R
that return values 5-21U
TODAY 1-45R
USING 1-46R
FUNCTION routine
RETURN 7-194R

FUNCTION statement
defining a record in 7-110R
definition of 1-23R
guidelines for using 7-110R
parameters in 5-16U
syntax 5-6U, 7-109R
Global flags
int_flag 1-31R
quit_flag 1-31R
GLOBALS statement
definition of 5-9U, 1-23R
syntax and notes 7-111R
with DEFINE LIKE 7-112R
GOTO statement
definition of 1-24R
LABEL 7-113R, 7-144R
syntax and notes 7-113R
GRANT statement
CONNECT 14-6U, 7-116R
DBA 14-8U, 7-116R
definition of 2-42R
guidelines for using 7-116R
RESOURCE 14-7U, 7-116R
syntax 7-114R
table privileges 14-8U
GROUP aggregate functions 9-36U
GROUP BY clause
syntax and notes 7-253R
with NULL values 2-62R
GROUP keyword 4-57R
HAVING clause
of a SELECT statement 7-255R
Help file
calling help messages 8-27U, 7-154R, 7-178R
compiling with mkmessage 8-25U
definition of 8-23U
designating a help key 8-26U
format of 8-23U
guidelines for creating 8-24U
showhelp function 5-26R
specifying the help file pathname 8-26U
Identifier
comparison of INFORMIX-4GL and RDSQL 2-7R
defining 7-68R
naming conventions 1-6R
RDSQL 2-6R

Identifier (Continued)
scope of 1-6R
IF statement
Boolean expressions in 4-14U
definition of 4-14U, 1-24R
syntax and notes 7-118R
INCLUDE attribute
definition of 6-25U, 3-33R
guidelines for using 3-33R
specifying values in 6-25U
Index
ascending and descending 2-35U
creating the index 2-33U
definition of 2-17U
guidelines for using 2-32U, 2-51R
multiple-column 2-31U
naming conventions 2-34U
NULL value 2-59R
removing from a database 14-10U, 7-88R
single-column 2-31U
INDEX keyword
GRANT statement 7-114R
infield function
ON KEY 13-45U
syntax and notes 5-18R, 7-127R, 7-137R
INFORMIX-4GL
as a development tool 1-9U
as a menu-building utility 1-15U
as a programming language 1-10U
as a report writer 1-16U
as a screen-building utility 1-11U
definition of 1-6U
general features of 1-8U
language overview 1-5R
INITIALIZE statement
definition of 1-22R
syntax and notes 7-120R
TO NULL 7-121R
IN keyword 4-32U
INPUT ARRAY statement
AFTER DELETE 11-46U
AFTER FIELD 11-29U
AFTER INSERT 11-38U
arr_curr function 5-10R
BEFORE FIELD 11-27U
BEFORE INSERT 11-36U

INPUT ARRAY statement (Continued)
definition of 11-19U, 1-27R
guidelines for using 7-131R
infield function 5-18R, 7-137R
optional clauses, format with 11-25U
scr_line function 5-23R
set_count function 5-25R
syntax 11-19U, 7-129R
table of functions in 11-23U, 7-137R
terminating 7-133R
WITHOUT DEFAULTS 11-26U, 7-131R
INPUT statement
AFTER FIELD 7-125R
arr_count function 5-8R
BY NAME 7-17U, 7-124R
definition of 7-12U, 1-27R
EXIT INPUT 7-96R
FROM 7-12U
guidelines for using 7-13U, 7-124R
infield function 5-18R, 7-127R
NEXT FIELD 7-125R
ON KEY 10-23U
syntax 7-122R
terminating 7-126R
transferring data into variables 7-14U, 7-21U
WITHOUT DEFAULTS 7-21U, 7-124R
INSERT cursor
closing 7-24R
declaring 7-62R
flushing the insert buffer 7-102R
guidelines for using 2-26R
opening 7-159R
putting a row in the insert buffer 7-182R
INSERT keyword
GRANT statement 7-114R
INSERT statement
guidelines for using 3-25U, 7-141R
inserting through a view 2-67R
SERIAL columns in 7-142R
syntax 3-25U, 7-140R
with NULL values 2-63R
INSTRUCTIONS section of form specification
definition of 6-31U, 3-47R
DELIMITERS statement 6-34U, 3-47R
of multiple-table form 11-10U
screen arrays 3-51R

INSTRUCTIONS section of form specification (Continued)
screen records 3-49R
SCREEN RECORD statement 6-32U, 11-10U, 3-49R
Integer constant 1-7R
INTEGER data type
acceptable values 7-50R
with database columns 2-24U, 2-7R
with display fields 6-16U
with variables 3-9U, 1-9R
INTERRUPT key
DOS Preface-17U
UNIX Preface-17U
INTO clause
of a SELECT statement 7-232R
INTO TEMP clause
INSERT statement 7-141R
SELECT statement 7-259R
int_flag variable 10-17U, 1-31R, 7-66R
IS keyword 2-61R
items table
stores database Introduction-10R
Join
definition of 2-10U
in a SELECT statement 4-37U, 7-246R
in the stores database 2-11U, A-9R, A-9U
joined columns with the same name 7-247R
multiple joins 7-247R
outer join 2-70R, 7-247R
self-join 7-247R
where a column is NULL 2-62R
Key
basic keys for SQL Preface-16U
help keys 8-26U
interrupt 10-16U
restricted 8-26U, 10-24U
scrolling and editing 11-21U
used in screen arrays 11-21U, 7-132R
LABEL statement
definition of 1-24R
GOTO 7-113R, 7-144R
syntax and notes 7-144R
LAST keyword
ATTRIBUTE clause 13-20U
LEFT MARGIN statement 4-12R
length function 5-20R

LET statement
definition of 3-12U, 1-22R
examples 3-20U
numeric expressions in 3-13U
syntax and notes 7-146R
with NULL values 3-17U
with string expressions 3-15U
Library functions (INFORMIX-4GL)
arg_count 5-8R
arg_val 5-6R
arr-curr 5-10R
downshift 5-12R
errorlog 5-16R
err_get 5-13R
err_print 5-14R
err_quit 5-15R
infield 5-18R
length 5-20R
num_args 5-21R
scr_line 5-23R
set_count 5-25R
showhelp 5-26R
startlog 5-28R
upshift 5-30R
LIKE keyword 1-11R
LINE NO expression 4-60R
LOCK TABLE statement
definition of 2-42R, 2-54R
EXCLUSIVE 7-148R
MODE 7-148R
SHARE 7-148R
syntax and notes 7-148R
Looping statements
FOR 11-15U, 1-24R, 7-104R
FOREACH 4-11U, 1-24R, 7-106R
WHILE 1-24R, 7-222R
MAIN statement 2-18U, 1-22R, 7-150R
manufact table
stores database Introduction-10R
MATCHES keyword 4-33U
MAX aggregate function 4-28U, 9-32U, 4-58R, 7-265R
MDY function in a SELECT statement 7-269R
Menu, programmer-defined
changing the message line 8-18U
changing the prompt line 8-20U
displaying in a window 13-31U

Menu, programmer-defined (Continued)
displaying options 8-7U
layout 8-6U
naming menu options 8-12U
requesting help from 8-21U
selecting options 8-8U, 7-154R
MENU statement
COMMAND clause 8-12U
CONTINUE MENU 8-15U, 7-38R, 7-155R
definition of 8-10U, 1-25R
designating a help message 8-27U
EXIT MENU 8-16U, 7-96R
general syntax 8-10U
guidelines for using 7-153R
MENU and END MENU 8-11U
nesting menus 8-42U
NEXT OPTION 8-15U
summary of operations 8-10U
syntax 7-151R
MESSAGE statement
definition of 1-25R
erasing a message 7-9U
guidelines for using 7-157R
syntax 7-8U, 7-157R
MIN aggregate function 4-29U, 9-32U, 4-58R, 7-266R
mkmessage utility 8-25U
MODE keyword, LOCK TABLE statement 7-148R
MODIFY keyword, ALTER TABLE statement 7-14R
Module
definition of 1-23R
MONEY data type
acceptable values 7-51R
with database columns 2-27U, 2-8R
with display fields 6-16U
with variables 3-9U, 1-10R
MONTH function in a SELECT statement 7-270R
Naming conventions
column 2-21U, 2-6R, 7-51R
database 2-20U, 2-6R, 7-42R
field tags 6-17U
identifiers 1-6R
index 2-34U
table 2-6R, 7-51R
variables 3-7U
NEED FORMAT-only statement 4-47R

NEXT FIELD statement
in an ON KEY clause 13-45U
NEXT OPTION statement 8-15U
NOENTRY attribute 6-23U, 3-35R
Non-procedural language 1-7U
NULL values
assigning 3-16U
definition of 2-58R
in a column 2-58R
in a GROUP BY clause 2-62R
in an ORDER BY clause 2-62R
in Boolean expressions 3-18U, 2-60R
in expressions 2-59R
in joined columns 2-62R
in numeric expressions 3-17U
in string expressions 3-17U
NOT NULL clause 2-58R, 7-51R
truth tables 2-60R
with INSERT 2-63R
WITHOUT NULL INPUT 3-8R
with UPDATE 2-63R
Numeric expression
arithmetic operators in 3-14U
columns in 3-13U
constants in 3-13U
definition of 3-12U, 1-15R
formatting 1-46R
functions in 3-13U
NULL values in 3-17U
variables in 3-13U
num_args function 5-21R
ON EVERY ROW control block 4-37R
ON KEY clause
in an INPUT ARRAY statement 13-44U
in an INPUT statement 10-23U
in a PROMPT statement 10-24U
ON LAST ROW control block 4-39R
OPEN FORM statement
definition of 7-6U, 1-26R
opening a closed form 7-27R
syntax and notes 7-162R
OPEN statement
guidelines for using 7-159R
syntax 4-17U
USING clause 2-39R, 7-160R
with an INSERT cursor 2-26R, 7-160R

OPEN statement (Continued)
with a SELECT cursor 4-17U, 2-18R, 7-159R
OPEN WINDOW statement
AT clause 13-13U
ATTRIBUTE clause 13-12U, 13-19U, 7-166R
definition of 1-26R
guidelines for using 13-12U, 7-164R
syntax 13-12U
WITH clause 13-14U
WITH FORM clause 13-15U
Operator
alternation 12-11U
arithmetic 3-14U
range 12-11U, 7-34R
relational 4-30U, 1-16R
string 1-15R
UNION 7-261R
wildcard 12-10U
OPTIONS statement
ACCEPT KEY 7-172R
COMMENT LINE 7-171R
definition of 8-18U, 1-25R
DELETE KEY 7-172R
effect on window attributes 13-20U, 13-23U
ERROR LINE 7-171R
FORM LINE 7-171R
guidelines for using 7-173R
HELP FILE 8-26U, 7-172R
HELP KEY 8-26U, 7-173R
INPUT NO WRAP 7-171R
INPUT WRAP 7-171R
INSERT KEY 7-172R
MESSAGE LINE 8-18U, 7-170R
NEXT KEY 7-172R
PREVIOUS KEY 7-172R
PROMPT LINE 8-20U, 7-170R
syntax 7-170R
ORDER BY clause
ASC 7-257R
DESC 7-257R
in an INSERT statement 7-141R
in a REPORT statement 9-10U
in a SELECT statement 4-34U, 9-23U, 7-257R
multiple-column sorting 4-35U
syntax 4-34U
with NULL values 2-62R

ORDER BY section of REPORT statement
definition of 4-21R
grouping data 4-29R
order form 11-6U
orders table
stores database Introduction-10R
OR keyword 4-31U
OUTPUT section of REPORT statement
BOTTOM MARGIN 4-18R
guidelines for using 9-9U
LEFT MARGIN 4-12R
PAGE LENGTH 4-20R
REPORT TO 4-10R
REPORT TO PRINTER 9-39U
RIGHT MARGIN 4-14R
syntax 4-9R
TOP MARGIN 4-16R
OUTPUT TO REPORT statement 9-12U, 1-28R, 7-175R
PAGE HEADER control block 4-35R, 4-41R
PAGE LENGTH statement 4-20R
PAGENO expression 4-61R
PAGE TRAILER control block 4-44R
Parameter 5-16U
Pattern matching
special characters for 4-33U
with MATCHES 4-33U
PAUSE FORMAT-only statement 4-48R
PERCENT aggregate function 9-32U, 4-57R
PERFORM (INFORMIX-SQL) screens
with INFORMIX-4GL 3-5R, 3-60R
PICTURE attribute 6-30U
PICTURE attribute
definition of 6-29U, 3-37R
guidelines for using 3-37R
PREPARE statement
executing 12-19U, 7-94R
guidelines for using 2-30R, 7-176R
syntax 12-18U, 7-176R
using placeholders for values 2-32R
with a character string 2-32R
with a character variable 2-33R
PRINT FILE FORMAT-only statement 4-53R
PRINT FORMAT-only statement 4-50R
PRINT statement
COLUMN 9-18U
definition of 9-17U

PRINT statement (Continued)
fill characters in 9-29U
floating dollar sign in 9-31U
PAGENO 9-19U
SPACE or SPACES 9-18U
USING 9-29U
Procedural language 1-7U
Program array
arg_count function 5-8R
array of records 11-13U
arr_curr function 5-10R
definition of 11-12U
displaying rows in 11-51U
inserting rows 7-132R
table of functions for 11-23U, 7-137R
Program features
arrays 11-12U
assignment statements 3-12U, 1-22R
calling C functions 7-17R
case sensitivity 2-18U
commenting 2-40U, 1-5R
compiling, at operating system level 1-63R
compiling multi-module programs 6-11R
conditional statements 4-14U, 7-19R
error messages 10-7U, 1-28R
executing a non-4GL process 7-200R
expressions 1-14R
formatting programs 2-40U
functions 5-6U
identifiers 1-6R
INFORMIX-4GL language overview 1-5R
looping statements 4-11U, 11-15U
main program 2-18U
program flow statements 1-24R
records 4-6U
reports 9-6U, 1-27R
screen interaction statements 1-25R
statements for organizing a program 1-22R
statement types 1-21R
suspending program operation 7-207R
types of program modules 6-7R
Programmer's Environment
accessing 2-37U
compiling a program 2-43U
creating a database 2-41U
definition of 1-22U, 6-5R

Programmer's Environment (Continued)
Form option, steps for using 6-37U, 3-57R, 6-12R
modifying a form specification file 6-39U
Module option, steps for using 2-41U, 6-7R
options 2-38U, 6-6R
Program design option, steps for using 6-16R
RDSQL option 6-22R
Program samples
in C, called by INFORMIX-4GL 1-60R
that create a database 2-28U, 2-39U
that create a menu 8-30U
that delete rows from a table 7-35U
that enter a row into a table 7-24U
that enter several rows into a table 7-26U
that handle errors 10-12U
that perform queries by example 12-21U, 12-24U
that retrieve data from a database 3-6U, 3-34U
that select and display rows 7-28U, 7-30U
that test for operator interrupts 10-18U, 10-21U
that update rows in a table 7-32U
that use a FOR loop 11-16U
that use an alternate interrupt key 10-25U, 10-28U
that use arrays 11-52U
that use global variables 5-15U
that use parameters 5-20U
PROMPT statement
CHAR 7-179R
definition of 1-25R
guidelines for using 3-21U, 7-179R
HELP 7-178R
ON KEY 10-24U, 7-180R
syntax 7-178R
PUBLIC keyword, of a GRANT statement 7-115R
PUT statement
and the SQLCA record 2-29R
and the status variable 2-29R
FROM clause 2-41R, 7-184R
guidelines for using 2-26R, 7-182R
Query by example
CONSTRUCT 12-14U, 7-33R
constructing a SELECT statement 12-16U
definition of 12-6U
EXECUTE 12-19U
FOREACH 12-19U
PREPARE 12-18U
table of operators 12-8U, 7-33R

Query by example (Continued)
using the alternation operator 12-11U
using the range operator 12-11U
using wildcard characters 12-10U, 7-34R
Querying the database
compound queries 7-261R
query by example 12-6U
querying short fields 12-11U
searching FLOAT and SMALLFLOAT fields 12-12U
searching for rows with NULL values 2-61R
subqueries 7-213R, 7-249R
through views 2-66R
using relational operators 4-30U, 12-8U
with SELECT 3-28U, 4-27U
quit_flag variable 1-31R, 7-66R
RDSQL
ALTER INDEX statement 7-11R
ALTER TABLE statement 7-13R
audit trails 7-39R, 7-85R, 7-185R
BEGIN WORK statement 7-16R
CLOSE DATABASE statement 7-26R
CLOSE statement 4-21U, 2-19R, 2-26R
COMMIT WORK statement 7-31R
comparison with INFORMIX-4GL identifiers 2-7R
CREATE AUDIT statement 7-39R, 7-185R
CREATE DATABASE statement 7-41R
CREATE INDEX statement 7-44R
CREATE SYNONYM statement 7-47R
CREATE TABLE statement 7-49R
CREATE VIEW statement 7-55R
data access statements 2-42R
DATABASE statement 7-60R
data conversion 7-142R
data definition statements 2-10R
data integrity statements 2-43R
data manipulation statements 2-13R
DECLARE statement 4-10U, 2-15R, 2-26R, 2-30R, 7-62R
definition of 2-5R
DELETE statement 7-70R
DROP AUDIT statement 7-85R, 7-185R
DROP DATABASE statement 7-86R
DROP INDEX statement 7-88R
DROP SYNONYM statement 7-89R
DROP TABLE statement 7-90R
DROP VIEW statement 7-91R
EXECUTE statement 2-30R, 7-94R

RDSQL (Continued)
FETCH statement 4-18U, 4-24U, 2-18R, 7-98R
FLUSH statement 2-26R, 7-102R
FOREACH statement 4-11U, 2-17R
GRANT statement 7-114R
identifiers 2-6R
INSERT statement 7-140R
interactive query language 2-5R
LOCK TABLE statement 7-148R
OPEN statement 4-17U, 2-18R, 2-26R
PREPARE statement 2-30R, 7-176R
PUT statement 2-26R, 7-182R
RECOVER TABLE statement 7-185R
RENAME COLUMN statement 7-187R
RENAME TABLE statement 7-189R
REVOKE statement 7-195R
ROLLBACK WORK statement 7-198R
ROLLFORWARD DATABASE statement 7-199R
SELECT statement 7-204R, 7-224R
SET LOCK MODE statement 7-205R
START DATABASE statement 7-208R
subscripting CHAR columns 7-227R
syntax conventions Introduction-8R
testing statement execution with sqlca 2-72R
UNLOCK TABLE statement 7-211R
update notes for Version 1.10 users 2-59R
UPDATE statement 7-212R
UPDATE STATISTICS statement 7-217R
WEEKDAY() function 1-55R
WHENEVER statement 7-219R
YEAR() function 1-56R
Record
definition of 4-6U, 1-11R
in an array 11-13U
passing to a function 7-110R
SQLCA global record 2-72R
with SELECT 4-7U
RECORD keyword
defining variables 4-7U
LIKE keyword 4-9U
syntax 4-6U
RECOVER TABLE statement 7-185R
Relational operator
in a query by example 12-8U, 12-9U
table of 4-30U

RENAME COLUMN statement
definition of 2-12R
guidelines for using 14-12U, 7-187R
syntax 7-187R
RENAME TABLE statement
definition of 14-11U, 2-11R
syntax and notes 7-189R
REPORT routine
syntax 7-191R
Reports, programmer-defined
aggregate functions 9-32U
calculations in 9-27U
calculations on groups 9-36U
counting rows 9-34U
creating custom formats 9-15U, 4-22R, 4-28R
default layout 9-10U, 4-24R
definition of 9-6U
features 4-5R
minimal report 4-7R
output of a report 9-39U
piping a report to a program 9-40U
printing data 4-50R
running a report 9-12U
sending output to a file 9-40U, 4-10R
sorting data 4-21R
starting report processing 7-209R
steps for creating 9-7U
structure 4-7R
REPORT statement
aggregate functions 4-56R
control blocks 4-28R
DEFINE section 9-9U, 4-7R, 4-8R
definition of 9-7U, 1-23R
FORMAT-only statements 4-46R
FORMAT section 9-11U, 4-7R, 4-22R
general syntax 9-8U
grouping data 4-28R
guidelines for using 7-192R
ORDER BY section 9-10U, 4-7R, 4-21R
ORDER EXTERNAL BY 9-25U
OUTPUT section 9-9U, 9-39U, 4-7R, 4-9R
table of GROUP functions 9-36U
REPORT TO statement 4-10R
REQUIRED attribute 6-23U, 3-39R
RESOURCE access privilege 2-49R, 7-116R

RETURN statement 7-194R
REVERSE attribute 6-24U, 3-41R
REVOKE statement
ALL 7-195R
ALTER 7-195R
CONNECT 14-6U, 7-196R
DBA 14-8U, 7-196R
definition of 2-42R
DELETE 7-195R
guidelines for using 7-196R
INDEX 7-195R
INSERT 7-195R
RESOURCE 14-7U, 7-196R
SELECT 7-195R
syntax 7-195R
table privileges 14-8U
UPDATE 7-195R
RIGHT MARGIN statement 4-14R
ROLLBACK WORK statement
definition of 2-43R
restoring previous database 14-16U
syntax and notes 7-198R
ROLLFORWARD DATABASE statement
definition of 14-17U, 2-44R
recovering a database 2-46R
syntax 7-199R
Row
definition of 2-9U, 2-6R
deleting from a table 7-70R
determining row length 2-30U
duplicates in a view 2-67R
ROWID 2-71R
RUN statement 1-25R
RUN statement
syntax and guidelines 7-200R
Runtime errors 10-6U
Screen array
definition of 3-51R
format of 11-8U
keys for scrolling and editing 11-21U, 7-132R
screen record corresponding to 11-10U
table of functions for 11-23U, 7-137R
Screen display characteristics
changing attributes 3-56R, 7-76R, 7-79R, 7-84R, 7-93R, 7-158R
clearing the screen 7-22R
default screen attributes 3-53R

Screen display characteristics (Continued)
reverse video 3-41R
screen coordinates 7-75R
setting up screen attributes 14-21U
table of color/intensity values 3-54R
Screen form
clearing the form 7-22R
clearing values in the form 7-20U, 7-22R
customer form 6-6U
definition of 6-6U
displaying 7-6U, 7-7U, 7-83R
displaying values in fields 7-18U, 7-73R
opening a 7-6U, 7-162R
order form 11-6U
Screen record
default screen record 6-32U
definition of 6-32U, 3-49R
moving rows through a screen array 7-202R
that specifies a screen array 11-10U
SCREEN RECORD statement
definition of 6-32U
syntax 6-32U, 6-33U, 3-49R
SCREEN section of form specification
definition of 6-14U, 3-8R
display field 6-15U, 3-10R
for a multiple-table form 11-8U
page layout 6-14U, 3-8R
using delimiters in 6-15U, 3-10R, 3-48R
SCROLL statement
definition of 1-27R
guidelines for using 7-202R
syntax 7-202R
scr_line function 11-23U, 5-23R, 7-138R
SELECT clause
ALL 7-229R
display label 7-230R
DISTINCT and UNIQUE keywords 7-229R
guidelines for using 7-229R
INTO 7-232R
syntax 7-229R
SELECT cursor
advancing 4-18U, 4-24U
closing 4-21U, 7-24R
declaring 7-62R
opening 4-17U, 7-159R
syntax and notes 2-14R

SELECT keyword
GRANT statement 7-114R
SELECT statement
aggregate functions in 4-29U, 7-265R
arithmetic operators in 4-28U
constructing, for query by example 12-13U
constructing, with LET 12-16U
creating temporary tables 7-259R
defining a view 2-65R
definition of 3-28U, 7-204R
displaying results 3-32U
expression in 7-226R
FROM clause 3-30U, 7-234R
GROUP BY clause 7-253R
HAVING clause 7-255R
INSERT statement 7-140R
INTO clause 3-30U, 7-99R, 7-107R
INTO TEMP clause 7-259R
joining columns with 4-37U, 7-246R
ORDER BY clause 4-34U, 7-257R
overview of clauses 7-227R
preparing, for query by example 12-18U
relational operators 7-227R
SELECT clause 3-29U, 7-229R
singleton SELECT statements 3-28U, 7-232R
syntax 7-224R
UNION operator 7-261R
using multiple tables 4-36U
WHERE clause 3-31U, 7-236R
with an outer join 2-70R, 7-247R
with a record 4-7U
with a self-join 7-247R
with multiple joins 7-247R
with subqueries 7-249R
SERIAL data type
acceptable values 7-51R
INSERT statement 7-142R
retrieved by SQLCA.SQLERRD[2] 7-22U, 13-45U
with database columns 2-25U, 2-9R
with display fields 6-16U
SET keyword, UPDATE statement 7-212R
SET LOCK MODE statement
syntax 7-205R
set_count function
definition of 11-23U
examples 13-49U, 13-54U

set_count function (Continued)
guidelines for using 5-25R, 7-78R
with INPUT ARRAY 7-138R
SHARE keyword, LOCK TABLE statement 7-148R
showhelp function 5-26R
SKIP FORMAT-only statement 4-54R
SLEEP statement
definition of 1-25R
syntax and notes 7-207R
SMALLFLOAT data type
acceptable values 7-50R
with database columns 2-24U, 2-8R
with display fields 6-16U
with FORMAT attribute 3-30R
with variables 3-9U, 1-10R
SMALLINT data type
acceptable values 7-50R
with database columns 2-24U, 2-7R
with display fields 6-16U
with variables 3-9U, 1-9R
Sorting data
in an index 2-35U
in a report 9-10U, 9-23U, 4-21R
multiple-column sorting 4-35U
with NULL values 2-62R
with ORDER BY 4-34U, 9-10U, 4-30R, 7-257R
SQLCA record
definition of 2-72R
set by CLOSE 2-29R
set by FLUSH 2-29R
set by PUT 2-29R
SQLERRD[2] used 13-45U
START DATABASE statement
definition of 2-43R
syntax and notes 7-208R
startlog function
creating an error log 10-10U
guidelines for using 5-28R
syntax 1-30R
START REPORT statement
definition of 9-12U, 1-27R
directing output 4-10R
syntax and notes 7-209R
Statements
syntax conventions, Introduction-8R

Statement type
assignment 1-22R
data access 2-42R
data definition 2-10R
data integrity 2-43R
data manipulation 2-13R
error and exception handling 1-28R
program flow 1-24R
program organization 1-22R
report generation 1-27R
screen interaction 1-25R
variable definition 1-21R
state table
stores database Introduction-10R
status variable
definition of 10-6U, 1-29R
NOTFOUND code 4-20U, 4-24U, 8-45U, 7-100R
stores database
copy protection Preface-18U
creating Preface-19U
customer form 6-6U
customer table Introduction-10R
items table Introduction-10R
join columns in 2-11U
manufact table Introduction-10R
order form 11-6U
orders table Introduction-10R
overview Preface-17U
restoring the original Preface-19U
sample reports 4-8R
state table Introduction-10R
stock table Introduction-10R
tables in 2-7U, Introduction-10R
String constant 1-7R
String expression
CLIPPED keyword 3-16U
columns in 3-15U
concatenation operator 3-16U
constants 3-15U
definition of 3-15U, 1-15R
functions in 3-15U
NULL values in 3-17U
substring 1-15R
table of operators 1-15R
USING keyword 3-16U

Substring 1-15R, 7-147R
SUM aggregate function 4-28U, 9-32U, 4-57R, 7-265R
Synonym
creating 7-47R
for a table 7-235R
removing a synonym 7-89R
Syntax conventions
RDSQL statements Introduction-8R
syscolatt table
changing color names 3-54R
color/intensity values 3-54R
creating with upscol 14-21U
default structure 3-53R
definition of 14-20U
use by FORM4GL 3-15R
with DISPLAY LIKE 3-27R
syscolval table
as used by INITIALIZE 7-121R
comparing values of variables 7-218R
creating with upscol 14-21U
data validation 1-31R
default structure 3-53R
definition of 14-20U
use by FORM4GL 3-15R
with VALIDATE LIKE attribute 3-44R
System catalogs
creating 7-41R
definition of 2-6R
syscolumns 2-65R
systables 7-217R
sysviews 2-65R
Table
access privileges 2-50R
adding a column 14-12U, 7-13R
alias for table name 7-235R
creating 2-20U
creating an alternate name (synonym) 7-47R
creating a temporary table 7-259R
definition of 2-6U, 2-6R
granting user access 14-8U
guidelines for indexing 2-32U, 2-51R
joining tables 2-10U, 7-246R
locking 7-148R
modifying 7-13R
modifying a row 7-212R
naming conventions 2-6R, 7-51R

Table (Continued)
outer join 2-70R
removing a column 14-14U, 7-13R
removing a synonym 7-89R
removing a table 14-10U, 7-90R
renaming 14-11U, 7-189R
revoking user access 14-8U, 7-196R
rows and columns in 2-8U
syscolatt 14-20U
syscolval 14-20U
unlocking 7-211R
TABLES section of form specification
definition of 3-12R
guidelines for using 6-17U
Third generation languages 1-6U
THRU keyword 1-19R
TIME expression 4-62R
TODAY function 2-75R
TODAY keyword 3-26R
TOP MARGIN statement 4-16R, 4-35R, 4-41R
Transaction
committing modifications 7-31R
cycle 2-44R
definition of 14-15U, 2-43R
log 14-15U, 14-17U, 2-44R, 7-42R, 7-208R
log file maintenance 2-46R
recovering transactions 7-199R
starting 7-16R
statements for specifying 14-16U
undoing modifications 7-198R
UNLOCK TABLE statement
definition of 2-42R, 2-55R
syntax and notes 7-211R
UPDATE keyword
GRANT statement 7-115R
UPDATE statement
data conversion in 7-215R
examples 3-36U
guidelines for using 3-35U, 7-213R
subqueries 7-213R
syntax 3-35U, 7-212R
updating through a view 2-66R
WHERE CURRENT OF 2-24R
with NULL values 2-63R
UPDATE STATISTICS statement
definition of 2-12R

UPDATE STATISTICS statement (Continued)
FOR TABLE 7-217R
syntax and notes 7-217R
upscol utility 14-21U
UPSHIFT attribute 6-27U, 3-42R
upshift function 5-30R
USER function 2-75R
USING keyword 3-16U
VALIDATE LIKE attribute 6-22U, 3-44R
VALIDATE statement
definition of 1-31R
syntax and notes 7-218R
VALUES keyword
INSERT statement 7-140R
Variable
as a parameter 5-16U
data types 3-8U, 1-9R
declaring a 3-7U
definition of 3-7U, 1-9R
global 5-9U, 5-13U, 7-111R
in a DISPLAY statement 7-18U
in an INPUT statement 7-13U, 7-17U
in a REPORT statement 9-9U
in numeric expressions 3-13U
int_flag 10-17U, 1-31R
local 5-9U, 5-13U
naming conventions 3-7U, 5-13U
quit_flag 1-31R
scope of 5-11U, 5-13U, 1-6R
statements for assigning values 1-22R
statements for defining 1-21R
status variable 10-6U, 1-29R
VERIFY attribute 3-46R
View
access privileges 2-68R
alternate name (synonym) 7-47R
characteristics 7-56R
creating 2-65R
definition and uses 14-19U, 2-64R
deleting 2-65R, 7-91R
limitations 2-64R
modifying the database through 2-66R
querying the database through 2-66R
virtual column 7-56R
with duplicate rows 2-67R

Wait for lock
definition of 2-57R
SET LOCK MODE statement 7-205R
WEEKDAY function
in a SELECT statement 1-55R, 7-271R
WHENEVER statement
definition of 1-28R
ERROR 7-219R
forms of WHENEVER ERROR 10-8U, 10-9U
forms of WHENEVER WARNING 7-219R
guidelines for using 7-220R
scope of WHENEVER ERROR 1-29R, 7-220R
syntax 7-219R
trapping errors 10-7U, 1-29R
WHERE clause
ALL 7-250R
ANY 7-250R
comparison condition 7-236R
DATE function 7-267R
DAY function 7-268R
EXISTS 7-250R
IN 7-240R, 7-250R
joining columns in 4-36U, 7-246R
LIKE 7-241R
MATCHES 7-243R
MDY function 7-269R
MONTH function 7-270R
NULL values 7-245R
OR 4-31U
pattern matching in 4-33U, 7-243R
ranges in 4-31U
relational operators in 4-29U
search conditions 7-225R, 7-236R
sets in 4-32U
SOME 7-250R
syntax 7-236R
WEEKDAY function 7-271R
with AND 4-31U
with a subquery 7-249R
with BETWEEN 7-239R
with DELETE 3-38U
with NULL values 2-61R
with relational operators 7-237R
with SELECT 3-31U
with UPDATE 3-36U, 7-213R
YEAR function 7-272R

WHERE CURRENT OF
with UPDATE statement 7-213R
WHILE statement
CONTINUE WHILE 4-22U, 7-38R, 7-223R
definition of 1-24R
EXIT WHILE 4-22U, 7-96R, 7-223R
guidelines for using 7-223R
syntax 4-21U, 7-222R
Wildcard character 12-10U, 1-19R, 7-34R
Window
attributes 7-166R
automatic sizing to accommodate a form 13-15U, 7-165R
border 7-167R
BORDER attribute 13-21U
changing the current window 7-58R
clearing of text 13-33U, 13-63U, 7-22R
closing 13-26U, 7-29R
color 7-169R
context of 13-63U
current 13-58U
default attribute settings 13-20U
default comment line 13-19U
default form line 13-19U
default message line 13-19U
default prompt line 13-19U
definition of 13-6U
determining the size of 13-16U
displaying a form in 13-30U
displaying a menu in 13-31U
displaying a message in 13-27U
displaying a prompt in 13-29U
effect of OPTIONS statement 13-20U, 13-23U
including in a program 13-5U, 13-12U
in demonstration application 13-6U, 13-9U
indicating attributes 13-19U
indicating the current 13-60U
locating on screen 13-13U
multiple window program 13-9U
naming conventions 13-12U
opening 7-164R
opening a form in 13-15U, 13-18U, 13-25U
providing explicit dimensions 13-14U
removing from screen 13-26U, 13-65U
reserved lines in 13-19U
runtime errors with 13-19U, 13-23U, 13-67U
screen array displayed in 13-7U

Window (Continued)
setting prompt and message lines 13-23U
specifying attributes 13-19U
specifying the size of 13-14U
stack 13-58U, 13-60U, 7-29R, 7-59R, 7-169R
stack, example 13-59U
stack, how managed 13-62U
statements running in the current 13-63U
user access of 13-10U
uses for 13-6U
working with multiple 13-60U
Window management statements 13-5U
WITH clause
OPEN WINDOW statement 13-14U
WITH FORM clause
OPEN WINDOW statement 13-15U
WITHOUT DEFAULTS clause
in an INPUT ARRAY statement 11-26U, 7-131R
in an INPUT statement 7-124R
YEAR function
SELECT statement 1-56R, 7-272R