Programmer's Guide to the Oracle® Pro*C/C++™ Precompiler

Release 2.2

February 1996

Part No. A32548–1

ORACLE®

Programmer's Guide to the Oracle® Pro*C/C++™ Precompiler, Release 2.2

Part No. A32548–1

Contributing Authors: Jack Godwin, Tim Smith, Tom Portfolio
Contributors: Stephen Arnold, Julie Basu, Michael Chiocca, Sanford Dreskin, Pierre Dufour, Phil Locke, Valarie Moore, Jacqui Pons, Gael Turk, Scott Urman, Peter Vasterd, John Weisz

This software was not developed for use in any nuclear, aviation, mass transit, medical, or other inherently dangerous applications. It is the customer's responsibility to take all appropriate measures to ensure the safe use of such applications if the programs are used for such purposes.

The information in this document is subject to change without notice. If you find any problems in the documentation, please report them to us in writing. Oracle Corporation does not warrant that this document is error free.

Preface

This manual is a comprehensive user's guide and on–the–job reference to the Oracle Pro*C/C++ Precompiler release 2.2. It shows you how to use the database language SQL and Oracle's procedural extension—PL/SQL—in conjunction with Pro*C/C++ to manipulate data in an Oracle7 database. It explores a full range of topics—from underlying concepts to advanced programming techniques—and uses hands–on examples to teach you all you need to know.

What This Guide Has to Offer

This Guide shows you how the Oracle Pro*C/C++ Precompiler and embedded SQL can benefit your entire applications development process. It gives you the know–how to design and develop applications that harness the power of Oracle. And, as quickly as possible, it helps you become proficient in writing embedded SQL programs.

An important feature of this Guide is its emphasis on getting the most out of Pro*C/C++ and embedded SQL. To help you master these tools, this Guide shows you all the "tricks of the trade" including ways to improve program performance. It also includes many program examples to better your understanding and demonstrate the usefulness of embedded SQL and embedded PL/SQL.

> **Note:** You will not find installation instructions or other system–specific information in this Guide; refer to your system–specific Oracle documentation.

Who Should Read This Guide?

Anyone developing new applications or converting existing applications to run in the Oracle environment will benefit from reading this Guide. Written especially for programmers, this comprehensive treatment of the Oracle Pro*C/C++ Precompiler, Release 2.2, will also be of value to systems analysts, project managers, and others interested in embedded SQL applications. To use this Guide effectively, you need a working knowledge of applications programming in C or C++. It would be helpful to have some familiarity with the SQL database language, although this book will Guide you through most of the complexities of embedded SQL programming.

How the Pro*C/C++ Guide Is Organized

This Guide contains thirteen chapters and five appendices. The chapters are divided into two parts.

- Chapters 1 through 6 present the basics of Pro*C/C++ programming and tell you how to run the precompiler. For many Pro*C/C++ developers, these chapters are sufficient to enable them to write useful and powerful Pro*C/C++ applications.
- Chapters 7 through 12 offer more detailed descriptions of Pro*C/C++. They describe some of the more complicated things that you can do with the product, such as writing user exits for SQL Forms and SQL*Report, and writing Pro*C/C++programs that use dynamic SQL.

Chapter 1: Introduction
This chapter introduces you to the Oracle Precompilers. You look at their role in developing application programs that manipulate Oracle data and find out what they allow your applications to do.

Chapter 2: Learning the Basics
This chapter explains how embedded SQL programs do their work. You examine the special environment in which they operate, the impact of this environment on the design of your applications, the key concepts of embedded SQL programming, and the steps you take in developing an application.

Chapter 3: Developing a Pro*C/C++ Application
This chapter gives you the basic information you need to develop a Pro*C/C++ application. You learn the embedded SQL commands that declare variables and connect to an Oracle database. You also learn about the Oracle datatypes, National Language Support (NLS), data conversion, and how to take advantage of datatype equivalencing. In addition, this chapter shows you how to embed Oracle Call Interface (OCI) calls in your program and how to develop X/Open applications.

Chapter 4: Using Embedded SQL
This chapter teaches you the essentials of embedded SQL programming. You learn how to use host variables, indicator variables, cursors, cursor variables, and the fundamental SQL commands that insert, update, select, and delete Oracle data.

Chapter 5: Using Embedded PL/SQL
This chapter shows you how to improve performance by embedding PL/SQL transaction processing blocks in your program. You learn how to use PL/SQL with host variables, indicator variables, cursors, stored procedures, host arrays, and dynamic SQL.

Chapter 6: Using C++
This chapter describes how to precompile your C++ application, and lists three sample Pro*C/C++ programs written using C++.

Chapter 7: Running the Pro*C/C++ Precompiler
This chapter details the requirements for running the Oracle Pro*C/C++ Precompiler. You learn what happens during precompilation, how to issue the precompiler command, and how to specify the many useful precompiler options.

Chapter 8: Defining and Controlling Transactions
This chapter describes transaction processing. You learn the basic techniques that safeguard the consistency of your database.

Chapter 9: Handling Runtime Errors
This chapter discusses error reporting and recovery. It shows you how to use the SQLSTATE and SQLCODE status variables with the WHENEVER statement to detect errors and status changes. It also shows you how to use the SQLCA and ORACA to detect error conditions and diagnose problems.

Chapter 10: Using Host Arrays
This chapter looks at using arrays to improve program performance. You learn how to manipulate Oracle data using arrays, how to operate on all the elements of an array with a single SQL statement, and how to limit the number of array elements processed.

Chapter 11: Using Dynamic SQL
This chapter shows you how to take advantage of dynamic SQL. You are taught three methods—from simple to complex—for writing flexible programs that, among other things, let users build SQL statements interactively at run time.

Chapter 12: Implementing Dynamic SQL Method 4
This chapter gives you an in–depth explanation of Dynamic SQL Method 4—dynamic SQL using descriptors. With this technique, your application can process and execute *any* SQL statement at runtime.

Chapter 13: Writing User Exits
This chapter focuses on writing user exits for Oracle Tools applications. You learn about the commands that are used to interface between a forms application and a Pro*C/C++ user exit, and how to write and link a forms user exit.

Appendix A: New Features
This appendix highlights the improvements and new features introduced with release 2.2 of the Pro*C/C++ Precompiler.

Appendix B: Oracle Reserved Words, Keywords, and Namespaces
This appendix lists words that have a special meaning to Oracle and namespaces that are reserved for Oracle libraries.

Appendix C: Performance Tuning
This appendix shows you some simple, easy–to–apply methods for improving the performance of your applications.

Appendix D: Syntactic and Semantic Checking
This appendix shows you how to use the SQLCHECK option to control the type and extent of syntactic and semantic checking done on embedded SQL statements and PL/SQL blocks.

Appendix E: System–Specific References
This appendix documents the aspects of Pro*C/C++ that can be system–specific.

Appendix F: Embedded SQL Commands and Directives
This appendix contains descriptions of precompiler directives, embedded SQL commands, and Oracle embedded SQL extensions.

Conventions Used in This Guide

Important terms being defined for the first time are *italicized*. In discussions, UPPERCASE is used for database objects and SQL keywords, and *italicized lowercase* is used for the names of C variables, constants, and parameters. C keywords are in **bold face** type

Notation

The following notation is used in this Guide:

[ ] Square brackets enclose optional items in syntax descriptions.

< > Angle brackets enclose the name of a syntactic element in syntax descriptions.

{ } Braces enclose items, only one of which is required.

| A vertical bar separates options within brackets or braces.

. . Two dots separate the lowest and highest values in a range.

. . . An ellipsis shows that the preceding parameter can be repeated or that statements or clauses irrelevant to the discussion were left out.

ANSI/ISO Compliance

Release 2.2 of the Pro*C/C++ Precompiler complies *fully* with the ANSI/ISO SQL standards. Compliance with these standards was certified by the National Institute of Standards and Technology (NIST). To flag extensions to ANSI/ISO SQL, a FIPS Flagger is provided.

Requirements

ANSI standard X3.135–1992 (known informally as SQL92) provides three levels of compliance:

- Full SQL
- Intermediate SQL (a subset of Full SQL)
- Entry SQL (a subset of Intermediate SQL)

ANSI standard X3.168–1992 specifies the syntax and semantics for embedding SQL statements in application programs written in a standard programming language such as C.

A conforming SQL implementation must support at least Entry SQL. The Oracle Pro*C/C++ Precompiler does conform to Entry SQL92.

NIST standard FIPS PUB 127–1, which applies to RDBMS software acquired for federal use, also adopts the ANSI standards. In addition, it specifies minimum sizing parameters for database constructs and requires a “FIPS Flagger” to identify ANSI extensions.

For copies of the ANSI standards, write to

American National Standards Institute
1430 Broadway
New York, NY 10018
USA

Migrating an Application

To migrate applications from earlier releases of Pro*C and Pro*C / C++, see "Migrating an Application" on page 3 – 8.

Your Comments Are Welcome

The Oracle Corporation technical staff values your comments. As we write and revise, your opinions are the most important input we receive. Please use the Reader's Comment Form at the back of this manual to tell us what you like and dislike about this Oracle publication. If the form has been used or you want to contact us, please use the following address or FAX number:

Oracle Languages Documentation Manager
Oracle Corporation
500 Oracle Parkway
Redwood City, CA 94065
U.S.A.
FAX: 415–506–7200

Contents

Chapter 4

Chapter 5

CHAPTER

1

Introduction

This chapter introduces you to the Pro*C/C++ Precompiler. You look at its role in developing application programs that manipulate Oracle data and find out what it allows your applications to do. This chapter answers the following questions:

- What is an Oracle Precompiler?
- Why use the Oracle Pro*C/C++ Precompiler?
- Why use SQL?
- Why use PL/SQL?
- Does the Oracle Pro*C/C++ Precompiler meet industry standards?
- What are the answers to some frequently–asked questions about Pro*C/C++?

What Is an Oracle Precompiler?

An Oracle Precompiler is a programming tool that allows you to embed SQL statements in a high–level source program. As Figure 1 – 1 shows, the precompiler accepts the source program as input, translates the embedded SQL statements into standard Oracle runtime library calls, and generates a modified source program that you can compile, link, and execute in the usual way.

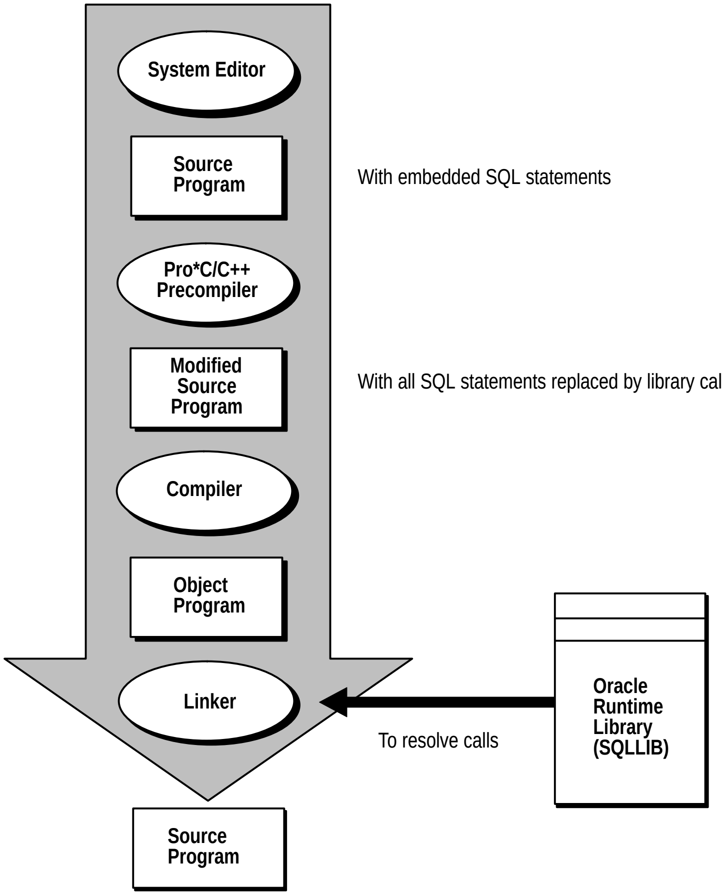


Figure 1 – 1 Embedded SQL Program Development

Why Use the Oracle Pro*C/C++ Precompiler?

The Oracle Pro*C/C++ Precompiler lets you use the power and flexibility of SQL in your application programs. A convenient, easy to use interface lets your application access Oracle directly.

Unlike many application development tools, the Pro*C/C++ Precompiler lets you create highly customized applications. For example, you can create user interfaces that incorporate the latest windowing and mouse technology. You can also create applications that run in the background without the need for user interaction.

Furthermore, Pro*C/C++ helps you fine–tune your applications. It allows close monitoring of resource use, SQL statement execution, and various runtime indicators. With this information, you can tweak program parameters for maximum performance.

Although precompiling adds a step to the application development process, it saves time because the precompiler, not you, translates each embedded SQL statement into several calls to the Oracle runtime library (SQLLIB).

Why Use SQL?

If you want to access and manipulate Oracle data, you need SQL. Whether you use SQL interactively through SQL*Plus or embedded in an application program depends on the job at hand. If the job requires the procedural processing power of C or C++, or must be done on a regular basis, use embedded SQL.

SQL has become the database language of choice because it is flexible, powerful, and easy to learn. Being non–procedural, it lets you specify what you want done without specifying how to do it. A few English–like statements make it easy to manipulate Oracle data one row or many rows at a time.

You can execute any SQL (not SQL*Plus) statement from an application program. For example, you can

- CREATE, ALTER, and DROP database tables dynamically
- SELECT, INSERT, UPDATE, and DELETE rows of data
- COMMIT or ROLLBACK transactions

Before embedding SQL statements in an application program, you can test them interactively using SQL*Plus. Usually, only minor changes are required to switch from interactive to embedded SQL.

Why Use PL/SQL?

An extension to SQL, PL/SQL is a transaction processing language that supports procedural constructs, variable declarations, and robust error handling. Within the same PL/SQL block, you can use SQL and all the PL/SQL extensions.

The main advantage of embedded PL/SQL is better performance. Unlike SQL, PL/SQL allows you to group SQL statements logically and send them to Oracle in a block rather than one by one. This reduces network traffic and processing overhead.

For more information about PL/SQL, including how to embed it in an application program, see Chapter 5, "Using Embedded PL/SQL".

What Does the Pro*C/C++ Precompiler Offer?

As Figure 1 – 2 shows, Pro*C/C++ offers many features and benefits, which help you to develop effective, reliable applications.

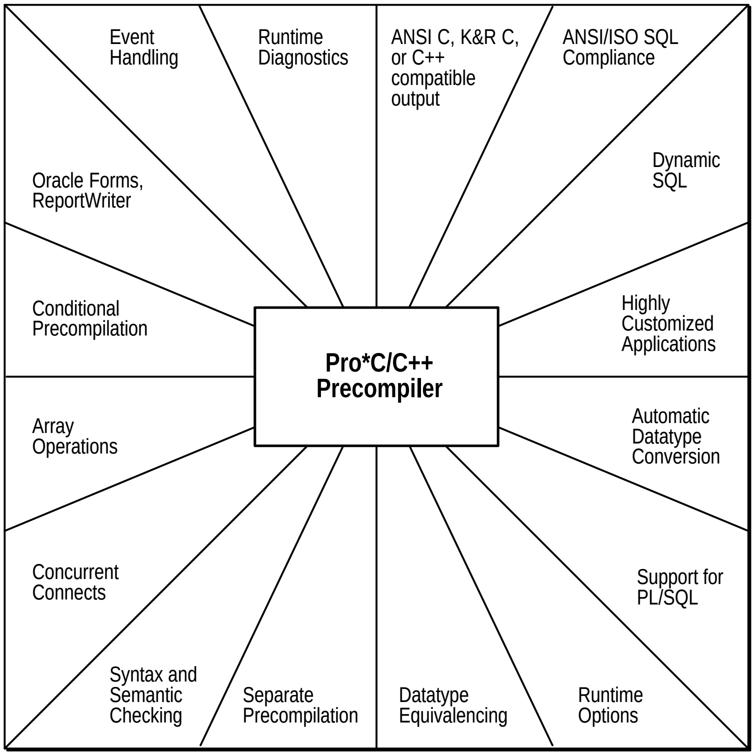


Figure 1 – 2 Features and Benefits

For example, Pro*C/C++ allows you to

- write your application in C or C++
- follow the ANSI/ISO standards for embedding SQL statements in a high–level language
- take advantage of dynamic SQL, an advanced programming technique that lets your program accept or build any valid SQL statement at runtime
- design and develop highly customized applications
- write multi–threaded applications
- automatically convert between Oracle internal datatypes and high–level language datatypes
- improve performance by embedding PL/SQL transaction processing blocks in your application program
- specify useful precompiler options inline and on the command line and change their values during precompilation

- use datatype equivalencing to control the way Oracle interprets input data and formats output data
- separately precompile several program modules, then link them into one executable program
- completely check the syntax and semantics of embedded SQL data manipulation statements and PL/SQL blocks
- concurrently access Oracle databases on multiple nodes using SQL*Net
- use arrays as input and output program variables
- conditionally precompile sections of code in your host program so that it can run in different environments
- directly interface with SQL*Forms via user exits written in a high–level language
- handle errors and warnings with the SQL Communications Area (SQLCA) and the WHENEVER or DO statement
- use an enhanced set of diagnostics provided by the Oracle Communications Area (ORACA)

To sum it up, the Pro*C/C++ Precompiler is a full–featured tool that supports a professional approach to embedded SQL programming.

Does the Oracle Pro*C/C++ Precompiler Meet Industry Standards?

SQL has become the standard language for relational database management systems. This section describes how the Pro*C/C++ Precompiler conforms to SQL standards established by the following organizations:

- American National Standards Institute (ANSI)
- International Standards Organization (ISO)
- U.S. National Institute of Standards and Technology (NIST)

These organizations have adopted SQL as defined in the following publications:

- ANSI standard X3.135–1992, *Database Language SQL*
- ISO/IEC standard 9075:1992, *Database Language SQL*
- ANSI standard X3.135–1989, *Database Language SQL with Integrity Enhancement*
- ANSI standard X3.168–1989, *Database Language Embedded SQL*
- ISO standard 9075–1989, *Database Language SQL with Integrity Enhancement*
- NIST standard FIPS PUB 127–1, *Database Language SQL* (FIPS is an acronym for Federal Information Processing Standards)

Requirements

ANSI standard X3.135–1992 (known informally as SQL92) provides three levels of compliance:

- Full SQL
- Intermediate SQL (a subset of Full SQL)
- Entry SQL (a subset of Intermediate SQL)

ANSI standard X3.168–1992 specifies the syntax and semantics for embedding SQL statements in application programs written in a standard programming language such as Ada, C, COBOL, FORTRAN, Pascal, or PL/I.

A conforming SQL implementation must support at least Entry SQL. The Oracle Pro*C/C++ Precompiler does conform to Entry SQL92.

NIST standard FIPS PUB 127–1, which applies to RDBMS software acquired for federal use, also adopts the ANSI standards. In addition, it specifies minimum sizing parameters for database constructs and requires a “FIPS Flagger” to identify ANSI extensions.

For copies of the ANSI standards, write to

American National Standards Institute
1430 Broadway
New York, NY 10018
USA

For a copy of the ISO standard, write to the national standards office of any ISO participant. For a copy of the NIST standard, write to

National Technical Information Service
U.S. Department of Commerce
Springfield, VA 22161
USA

Compliance

Under Oracle7, the Pro*C/C++ Precompiler complies 100% with current ANSI/ISO standards.

The Pro*C Precompiler also complies 100% with the NIST standard. It provides a FIPS Flagger and an option named FIPS, which enables the FIPS Flagger. For more information, see the section "FIPS Flagger" below.

Certification

NIST tested the Pro*C/C++ Precompiler for ANSI SQL92 compliance using the *SQL Test Suite*, which consists of nearly 300 test programs. Specifically, the programs tested for conformance to the C embedded SQL standard. The result: the Oracle Pro*C/C++ Precompiler was certified 100% ANSI–compliant for Entry SQL92.

For more information about the tests, write to

National Computer Systems Laboratory
Attn: Software Standards Testing Program
National Institute of Standards and Technology
Gaithersburg, MD 20899
USA

FIPS Flagger

According to FIPS PUB 127–1, "an implementation that provides additional facilities not specified by this standard shall also provide an option to flag nonconforming SQL language or conforming SQL language that may be processed in a nonconforming manner." To meet this requirement, the Pro*C/C++ Precompiler provides the *FIPS Flagger*, which flags ANSI extensions. An *extension* is any SQL element that violates ANSI format or syntax rules, except privilege enforcement rules. For a list of Oracle extensions to standard SQL, see the *Oracle7 Server SQL Reference.*

You can use the FIPS Flagger to identify

- nonconforming SQL elements that might have to be modified if you move the application to a conforming environment
- conforming SQL elements that might behave differently in another processing environment

Thus, the FIPS Flagger helps you develop portable applications.

FIPS Option

The FIPS precompiler option governs the FIPS Flagger. To enable the FIPS Flagger, specify FIPS=YES inline or on the command line. For more information about the FIPS option, see the section "Using the Precompiler Options" on page 7 – 9.

Migrating an Application from Earlier Releases

There are several semantic changes in database operations between Oracle V6 and Oracle7. For information on how this affects Pro*C/C++ applications, see the section "Migrating an Application" on page 3 – 8, and the discussion of the DBMS precompiler option on page 7 – 13.

Frequently Asked Questions

This section presents some questions that are frequently asked about Pro*C/C++, and about Oracle7 in relation to Pro*C/C++. The answers are more informal than the documentation in the rest of this Guide, but do provide references to places where you can find the reference material.

Question: I'm confused by VARCHAR. What's a VARCHAR?

Answer: Here's a short description of VARCHARs:

VARCHAR2 — A kind of column in the database that contains variable–length character data, up to 2000 bytes. This is what Oracle calls an "internal datatype", because it's a possible column type. See page 3 – 50.

VARCHAR — An Oracle "external datatype" (datatype code 9). You use this only if you're doing dynamic SQL Method 4, or datatype equivalencing. See page 3 – 52 for datatype equivalencing, and Chapter 12 for dynamic SQL Method 4.

VARCHAR[n]
varchar[n] — This is a Pro*C/C++ "pseudotype" that you can declare as a host variable in your Pro*C/C++ program. It's actually generated by Pro*C as a **struct**, with a 2–byte length element, and a [n]–byte character array. See page 3 – 35.

Question: Does Pro*C/C++ generate calls to the Oracle Call Interface (OCI)?

Answer: No. Pro*C/C++ generates data structures, and calls to its runtime library: SQLLIB (*libsql.a* in UNIX). SQLLIB, in turn, calls the UPI to communicate with the database.

Question: Then why not just code using SQLLIB calls, and not use Pro*C/C++?

Answer: SQLLIB is not externally documented, is unsupported, and might change from release to release. Also, Pro*C/C++ is an ANSI/ISO compliant product, that follows the standard requirements for embedded SQL.

If you need to do low–level coding, use the OCI. It is supported, and is guaranteed to stay compatible from release to release. Also, Oracle is committed to supporting the OCI.

You can also mix OCI and Pro*C. See page 3 – 97.

Question: Can I call a PL/SQL stored procedure from a Pro*C/C++ program?

Answer: Certainly. See Chapter 5. There's a demo program starting on page 5 – 21.

Question: Can I write C++ code, and precompile it using Pro*C/C++?

Answer: Yes. Starting with Pro*C/C++ release 2.1, you can precompile C++ applications. See Chapter 6.

Question: Can I use bind variables anyplace in a SQL statement? For example, I'd like to be able to input the name of a table in my SQL statements at runtime. But when I use host variables, I get precompiler errors.

Answer: In general, you can use host variables at any place in a SQL, or PL/SQL, statement where expressions are allowed. See page 3 – 18. The following SQL statement, where *table_name* is a host variable, is *illegal*:

```
EXEC SQL SELECT ename,sal INTO :name, :salary FROM :table_name;
```

To solve your problem, you need to use dynamic SQL. See Chapter 11. There is a demo program that you can adapt to do this starting on page 11 – 9.

Question: I am confused by character handling in Pro*C/C++. It seems that there are many options. Can you help?

Answer: There are many options, but we can simplify. First of all, if you need compatibility with previous V1.x precompilers, and with both Oracle V6 and Oracle7, the safest thing to do is use VARCHAR[n] host variables. See page 3 – 35.

The default datatype for all other character variables in Pro*C/C++ is CHARZ; see page 3 – 55. Briefly, this means that you must null–terminate the string on input, and it is both blank–padded and null–terminated on output.

If neither VARCHAR nor CHARZ works for your application, and you need total C–like behavior (null termination, absolutely no blank–padding), use the TYPE command and the C **typedef** statement, and use datatype equivalencing to convert your character host variables to STRING. See page 3 – 59. There is a sample program that shows how to use the TYPE command starting on page 3 – 25.

Question: What about character pointers? Is there anything special about them?

Answer: Yes. When Pro*C/C++ binds an input or output host variable, it must know the length. When you use VARCHAR[n], or declare a host variable of type **char[n]**, Pro*C/C++ knows the length from your declaration. But when you use a character pointer as a host variable, and *malloc()* the buffer in your program, Pro*C/C++ has no way of knowing the length.

So, what you must do on output is not only allocate the buffer, but pad it out with some non–null characters, then null–terminate it. On input or output, Pro*C/C++ calls *strlen()* for the buffer to get the length. See page 3 – 33.

Question: Why doesn't SPOOL work in Pro*C/C++?

Answer: SPOOL is a special command used in SQL*Plus. It is not an embedded SQL command. See page 2 – 3.

Question: Where can I find the on-line versions of the sample programs?

Answer: Each Oracle installation should have a *demo* directory. On UNIX systems, for example, it is located in *$ORACLE_HOME/proc/demo*. If the directory is not there, or it does not contain the sample programs, see your system or database administrator.

Question: How can I compile and link my application?

Answer: Compiling and linking are very platform specific. Your system–specific Oracle documentation has instructions on how to link a Pro*C/C++ application. On UNIX systems, there is a makefile called *proc.mk* in the *demo* directory. To link, say, the demo program *sample1.pc*, you would enter the command line

```
make –f proc.mk sample1
```

If you need to use special precompiler options, you can run Pro*C/C++ separately, then do the make. Or, you can create your own custom makefile. For example, if your program contains embedded PL/SQL code, you can do

```
proc cv_demo userid=scott/tiger sqlcheck=semantics
make –f proc.mk cv_demo
```

On VMS systems, there is a script called LNPROC that you use to link your Pro*C/C++ applications.

Question: I have been told that Pro*C/C++ now supports using structures as host variables. How does this work with the array interface?

Answer: You can use arrays inside a single structure. See page 3 – 21. However, you cannot use an array of structures with the array interface. See page 3 – 22.

Question: Is it possible to have recursive functions in Pro*C/C++, if I use embedded SQL in the function?

Answer: Yes. With release 2.1 of Pro*C/C++, you can also use cursor variables in recursive functions.

Question: Can I use any release of the Oracle Pro*C or Pro*C/C++ Precompiler with any version of the Oracle Server?

Answer: No. You can use an older version of Pro*C or Pro*C/C++ with a newer version of the server, but you cannot use a newer version of Pro*C or Pro*C/C++ with an older version of the server.

For example, you can use release 1.4 of Pro*C with Oracle7, but you cannot use Pro*C/C++ release 2.1 with the Oracle version 6 server.

Question: When my application runs under Oracle7, I keep getting an ORA–1405 error (fetched column value is null). It worked fine under Oracle V6. What is happening?

Answer: You are selecting a null into a host variable that does not have an associated indicator variable. This is not in compliance with the ANSI/ISO standards, which is why Oracle7 changed.

If possible, rewrite your program using indicator variables, and use indicators in future development. Indicator variables are described on page 3 – 19.

Alternatively, if precompiling with MODE=ORACLE and DBMS=V7 or V6_CHAR, specify UNSAFE_NULL=YES the command line (see "UNSAFE_NULL" on page 7 – 36 for more information) to disable the ORA–01405 message, or precompile with DBMS=V6.

Question: Are all SQLLIB functions private?

Answer: No. There are some SQLLIB functions that you can call to get information about your program, or its data. The SQLLIB public functions are shown here:

sqlald()
sqlaldt()
	Used to allocate a SQL descriptor array (SQLDA) for dynamic SQL Method 4. See page 12 – 4.

sqlcda()
sqlcdat()
	Used to convert a Pro*C cursor variable to an OCI cursor data area. See page 3 – 81.

sqlclu()
sqlclut()
	Used to free a SQLDA allocated using *sqlald()*. See page 12 – 32.

sqlcur()
sqlcurt()
	Used to convert an OCI cursor data area to a Pro*C cursor variable. See page 3 – 81.

sqlglm()
sqlglmt()
	Returns a long error message. See page 9 – 22.

sqlgls()
sqlglst()
	Used to return the text of the most recently executed SQL statement. See page 9 – 29.

sqlld2()
sqlld2t()
	Used to obtain a valid Logon Data Area for a named connection, when OCI calls are being used in a Pro*C program. See page 3 – 115.

sqllda()
sqlldat()
	Used to obtain a valid Logon Data Area for the most recent connection, when OCI calls are being used in a Pro*C program. See page 3 – 97.

sqlnul()
sqlnult()
	Returns an indication of null status, for dynamic SQL Method 4. See page 12 – 15.

sqlprc()
sqlprct()
Returns precision and scale of NUMBERS. See page 12 – 13.

sqlpr2()
sqlpr2t()
A variant of *sqlprc()*. See page 12 – 15.

sqlvcp()
sqlvcpt()
Used for getting the padded size of a VARCHAR[n]. See page 3 – 37.

In the preceding list, the function names ending in *t* are the thread–safe SQLLIB public functions. Use these functions in multi–threaded applications. For more information about thread–safe public functions, see "Thread–safe SQLLIB Public Functions" on page 3 – 107.

CHAPTER

2

Learning the Basics

This chapter explains how embedded SQL programs do their work. You examine the special environment in which they operate and the impact of this environment on the design of your applications.

After covering the key concepts of embedded SQL programming and the steps you take in developing an application, this chapter uses a simple program to illustrate the main points.

Key Concepts of Embedded SQL Programming

This section lays the conceptual foundation on which later chapters build. It discusses the following subjects:

- embedded SQL statements
- executable versus declarative SQL statements
- static versus dynamic SQL statements
- embedded PL/SQL blocks
- host and indicator variables
- Oracle datatypes
- arrays
- datatype equivalencing
- private SQL areas, cursors, and active sets
- transactions
- errors and warnings

Embedded SQL Statements

The term *embedded SQL* refers to SQL statements placed within an application program. Because it houses the SQL statements, the application program is called a *host program*, and the language in which it is written is called the *host language*. For example, the Pro*C/C++ Precompiler allows you to embed SQL statements in a C host program.

Figure 2 – 1 shows all the interactive SQL statements your application program can execute.

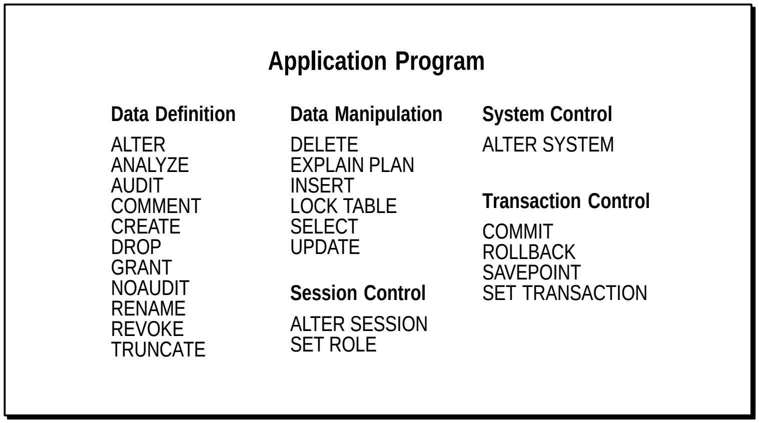


Figure 2 – 1 SQL Allowed in a Program

For example, to manipulate and query Oracle data, you use the INSERT, UPDATE, DELETE, and SELECT statements. INSERT adds rows of data to database tables, UPDATE modifies rows, DELETE removes unwanted rows, and SELECT retrieves rows that meet your search condition.

The Pro*C/C++ Precompiler supports all the Oracle7 SQL statements. For example, the powerful SET ROLE statement lets you dynamically manage database privileges. A *role* is a named group of related system and/or object privileges granted to users or other roles. Role definitions are stored in the Oracle data dictionary. Your applications can use the SET ROLE statement to enable and disable roles as needed.

Only SQL statements—not SQL*Plus statements—are valid in an application program. (SQL*Plus has additional statements for setting environment parameters, editing, and report formatting.)

Executable versus Declarative Statements

Embedded SQL includes all the interactive SQL statements plus others that allow you to transfer data between Oracle and a host program. There are two types of embedded SQL statements: *executable* and *declarative*. Executable statements result in calls to the runtime library SQLLIB. You use them to connect to Oracle, to define, query, and manipulate Oracle data, to control access to Oracle data, and to process transactions. They can be placed wherever C language executable statements can be placed.

Declarative statements, on the other hand, do not result in calls to SQLLIB and do not operate on Oracle data. You use them to declare Oracle objects, communications areas, and SQL variables. They can be placed wherever C variable declarations can be placed. You treat the ALLOCATE statement, however, as an executable, not a declarative, statement.

Table 2 – 1 groups the various embedded SQL statements.

Declarative SQL	
STATEMENT	**PURPOSE**
ARRAYLEN*	To use host arrays with PL/SQL

BEGIN DECLARE SECTION*
END DECLARE SECTION*
	To declare host variables

DECLARE*	To name Oracle objects
INCLUDE*	To copy in files
TYPE*	To equivalence datatypes
VAR*	To equivalence variables
WHENEVER*	To handle runtime errors
Executable SQL	
STATEMENT	**PURPOSE**
ALLOCATE* ALTER ANALYZE AUDIT COMMENT CONNECT* CREATE DROP GRANT NOAUDIT RENAME REVOKE TRUNCATE CLOSE*	To define and control Oracle data
DELETE EXPLAIN PLAN FETCH* INSERT LOCK TABLE OPEN* SELECT UPDATE	To query and manipulate Oracle data
COMMIT ROLLBACK SAVEPOINT SET TRANSACTION	To process transactions
DESCRIBE* EXECUTE* PREPARE*	To use dynamic SQL
ALTER SESSION SET ROLE	To control sessions
*Has no interactive counterpart	

Table 2 – 1 Embedded SQL Statements

Embedded SQL Syntax

In your application program, you can freely intermix complete SQL statements with complete C statements and use C variables or structures in SQL statements. The only special requirement for building SQL statements into your host program is that you begin them with the keywords EXEC SQL and end them with a semicolon. Pro*C/C++ translates all EXEC SQL statements into calls to the runtime library SQLLIB.

Many embedded SQL statements differ from their interactive counterparts only through the addition of a new clause or the use of program variables. The following example compares interactive and embedded ROLLBACK statements:

```
ROLLBACK WORK;           -- interactive
EXEC SQL ROLLBACK WORK;  -- embedded
```

These statements have the same effect, but you would use the first in an interactive SQL environment (such as when running SQL*Plus), and the second in a Pro*C/C++ program.

Static versus Dynamic SQL Statements

Most application programs are designed to process static SQL statements and fixed transactions. In this case, you know the makeup of each SQL statement and transaction before runtime; that is, you know which SQL commands will be issued, which database tables might be changed, which columns will be updated, and so on.

However, some applications might be required to accept and process any valid SQL statement at runtime. So, you might not know until runtime all the SQL commands, database tables, and columns involved.

Dynamic SQL is an advanced programming technique that lets your program accept or build SQL statements at run time and take explicit control over datatype conversion.

Embedded PL/SQL Blocks

The Pro*C/C++ Precompiler treats a PL/SQL block like a single embedded SQL statement. So, you can place a PL/SQL block anywhere in an application program that you can place a SQL statement. To embed PL/SQL in your host program, you simply declare the variables to be shared with PL/SQL and bracket the PL/SQL block with the keywords EXEC SQL EXECUTE and END–EXEC.

From embedded PL/SQL blocks, you can manipulate Oracle data flexibly and safely because PL/SQL supports all SQL data manipulation and transaction processing commands. For more information about PL/SQL, see Chapter 5, “Using Embedded PL/SQL”.

Host and Indicator Variables

Host variables are the key to communication between Oracle and your program. A *host variable* is a scalar or aggregate variable declared in C and shared with Oracle, meaning that both your program and Oracle can reference its value.

Your program uses *input* host variables to pass data to Oracle. Oracle uses *output* host variables to pass data and status information to your program. The program assigns values to input host variables; Oracle assigns values to output host variables.

Host variables can be used anywhere a SQL expression can be used. In SQL statements, host variables must be prefixed with a colon (:) to set them apart from Oracle objects.

You can also use a C **struct** to contain a number of host variables. When you name the structure in an embedded SQL statement, prefixed with a colon, Oracle uses each of the components of the **struct** as a host variable.

You can associate any host variable with an optional indicator variable. An *indicator variable* is a short integer variable that "indicates" the value or condition of its host variable. You use indicator variables to assign nulls to input host variables and to detect nulls or truncated values in output host variables. A *null* is a missing, unknown, or inapplicable value.

In SQL statements, an indicator variable must be prefixed with a colon and immediately follow its associated host variable. The keyword INDICATOR can be placed between the host variable and its indicator for additional clarity.

If the host variables are packaged in a **struct**, and you want to use indicator variables, you simply create a **struct** that has an indicator variable for each host variable in the host structure, and name the indicator **struct** in the SQL statement, immediately following the host variable **struct**, and prefixed with a colon. You can also use the INDICATOR keyword to separate a host structure and its associated indicator structure.

Oracle Datatypes

Typically, a host program inputs data to Oracle, and Oracle outputs data to the program. Oracle stores input data in database tables and stores output data in program host variables. To store a data item, Oracle must know its *datatype,* which specifies a storage format and valid range of values.

Oracle recognizes two kinds of datatypes: *internal* and *external*. Internal datatypes specify how Oracle stores data in database columns. Oracle also uses internal datatypes to represent database pseudocolumns, which return specific data items but are not actual columns in a table.

External datatypes specify how data is stored in host variables. When your host program inputs data to Oracle, if necessary, Oracle converts between the external datatype of the input host variable and the internal datatype of the target database column. When Oracle outputs data to your host program, if necessary, Oracle converts between the internal datatype of the source database column and the external datatype of the output host variable.

Arrays

Pro*C/C++ lets you define array host variables called *host arrays* and operate on them with a single SQL statement. Using the array SELECT, FETCH, DELETE, INSERT, and UPDATE statements, you can query and manipulate large volumes of data with ease. You can also use host arrays inside a host variable **struct**.

Datatype Equivalencing

Pro*C/C++ adds flexibility to your applications by letting you *equivalence* datatypes. That means you can customize the way Oracle interprets input data and formats output data.

On a variable–by–variable basis, you can equivalence supported C datatypes to the Oracle external datatypes. You can also equivalence user–defined datatypes to Oracle external datatypes.

Private SQL Areas, Cursors, and Active Sets

To process a SQL statement, Oracle opens a work area called a *private SQL area*. The private SQL area stores information needed to execute the SQL statement. An identifier called a *cursor* lets you name a SQL statement, access the information in its private SQL area, and, to some extent, control its processing.

For static SQL statements, there are two types of cursors: *implicit* and *explicit*. Oracle implicitly declares a cursor for all data definition and data manipulation statements, including SELECT statements (queries) that return only one row. However, for queries that return more than one row, to process beyond the first row, you must explicitly declare a cursor (or use host arrays).

The set of rows returned is called the *active set*; its size depends on how many rows meet the query search condition. You use an explicit cursor to identify the row currently being processed, called the *current row*.

Imagine the set of rows being returned to a terminal screen. A screen cursor can point to the first row to be processed, then the next row, and so on. In the same way, an explicit cursor “points” to the current row in the active set. This allows your program to process the rows one at a time.

Transactions

A *transaction* is a series of logically related SQL statements (two UPDATEs that credit one bank account and debit another, for example) that Oracle treats as a unit, so that all changes brought about by the statements are made permanent or undone at the same time.

All the data manipulation statements executed since the last data definition, COMMIT, or ROLLBACK statement was executed make up the current transaction.

To help ensure the consistency of your database, Pro*C/C++ lets you define transactions using the COMMIT, ROLLBACK, and SAVEPOINT statements.

COMMIT makes permanent any changes made during the current transaction. ROLLBACK ends the current transaction and undoes any changes made since the transaction began. SAVEPOINT marks the current point in the processing of a transaction; used with ROLLBACK, it undoes part of a transaction.

Errors and Warnings

When you execute an embedded SQL statement, it either succeeds or fails, and might result in an error or warning. You need a way to handle these results. Pro*C/C++ provides two error handling mechanisms: the SQL Communications Area (SQLCA) and the WHENEVER statement.

The SQLCA is a data structure that you include (or hardcode) in your host program. It defines program variables used by Oracle to pass runtime status information to the program. With the SQLCA, you can take different actions based on feedback from Oracle about work just attempted. For example, you can check to see if a DELETE statement succeeded and, if so, how many rows were deleted.

With the WHENEVER statement, you can specify actions to be taken automatically when Oracle detects an error or warning condition. These actions include continuing with the next statement, calling a function, branching to a labeled statement, or even stopping.

Steps in Developing an Embedded SQL Application

Figure 2 – 2 walks you through the embedded SQL application development process.

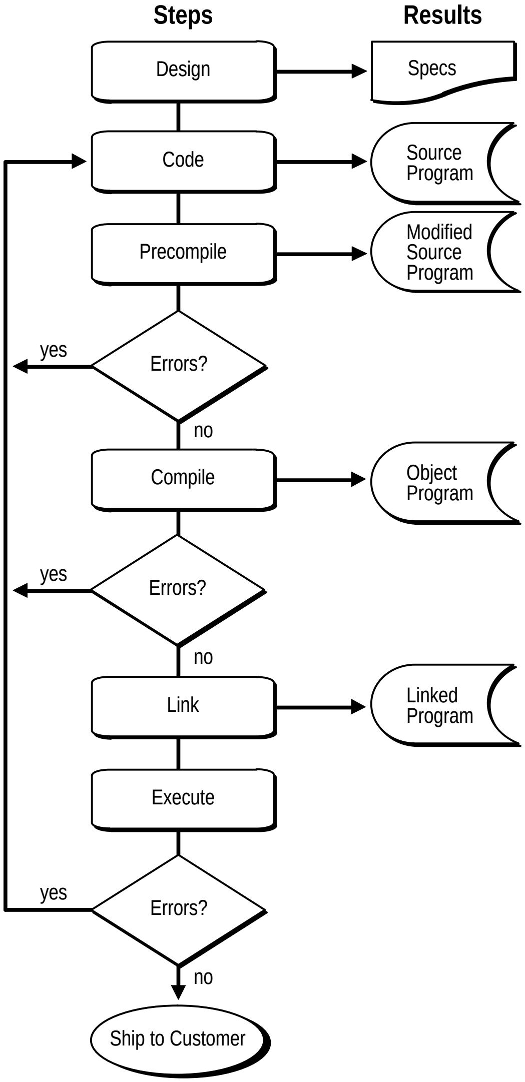


Figure 2 – 2 Embedded SQL Application Development Process

As you can see, precompiling results in a modified source file that can be compiled normally. Though precompiling adds a step to the traditional development process, that step is well worth taking because it lets you write very flexible applications.

Sample Tables

Most programming examples in this guide use two sample database tables: DEPT and EMP. Their definitions follow:

```
CREATE TABLE DEPT
    (DEPTNO    NUMBER(2) NOT NULL,
     DNAME     VARCHAR2(14),
     LOC       VARCHAR2(13))

CREATE TABLE EMP
    (EMPNO     NUMBER(4) NOT NULL,
     ENAME     VARCHAR2(10),
     JOB       VARCHAR2(9),
     MGR       NUMBER(4),
     HIREDATE  DATE,
     SAL       NUMBER(7,2),
     COMM      NUMBER(7,2),
     DEPTNO    NUMBER(2))
```

Sample Data

Respectively, the DEPT and EMP tables contain the following rows of data:

```
DEPTNO  DNAME      LOC
------- ---------- ---------
10      ACCOUNTING NEW YORK
20      RESEARCH   DALLAS
30      SALES      CHICAGO
40      OPERATIONS BOSTON

EMPNO ENAME   JOB          MGR  HIREDATE    SAL   COMM  DEPTNO
----- ------- --------- ------ --------- ------ ------ -------
 7369 SMITH   CLERK       7902 17–DEC–80    800             20
 7499 ALLEN   SALESMAN    7698 20–FEB–81   1600    300      30
 7521 WARD    SALESMAN    7698 22–FEB–81   1250    500      30
 7566 JONES   MANAGER     7839 02–APR–81   2975             20
 7654 MARTIN  SALESMAN    7698 28–SEP–81   1250   1400      30
 7698 BLAKE   MANAGER     7839 01–MAY–81   2850             30
 7782 CLARK   MANAGER     7839 09–JUN–81   2450             10
 7788 SCOTT   ANALYST     7566 19–APR–87   3000             20
 7839 KING    PRESIDENT        17–NOV–81   5000             10
 7844 TURNER  SALESMAN    7698 08–SEP–81   1500             30
 7876 ADAMS   CLERK       7788 23–MAY–87   1100             20
 7900 JAMES   CLERK       7698 03–DEC–81    950             30
 7902 FORD    ANALYST     7566 03–DEC–81   3000             20
 7934 MILLER  CLERK       7782 23–JAN–82   1300             10
```

Sample Program: A Simple Query

One way to get acquainted with Pro*C/C++ and embedded SQL is to study a program example. The program listed below is also available on–line in the file *sample1.pc* in your Pro*C/C++ *demo* directory.

The program connects to Oracle, then loops, prompting the user for an employee number. It queries the database for the employee's name, salary, and commission, displays the information, and then continues the loop. The information is returned to a host structure. There is also a parallel indicator structure to signal whether any of the output values SELECTed might be null.

You should precompile sample programs using the precompiler option MODE=ORACLE.

```
/*
 *  sample1.pc
 *
 *  Prompts the user for an employee number,
 *  then queries the emp table for the employee's
 *  name, salary and commission.  Uses indicator
 *  variables (in an indicator struct) to determine
 *  if the commission is NULL.
 *
 */

#include <stdio.h>
#include <string.h>


/* Define constants for VARCHAR lengths. */
#define     UNAME_LEN      20
#define     PWD_LEN        40

/* Declare variables.  No declare section is
   needed if MODE=ORACLE. */
VARCHAR     username[UNAME_LEN];  /* VARCHAR is an Oracle–supplied
struct */
varchar     password[PWD_LEN];    /* varchar can be in lower case
also. */
```

```
/* Define a host structure for the output values of
   a SELECT statement.  */
struct
{
    VARCHAR   emp_name[UNAME_LEN];
    float     salary;
    float     commission;
} emprec;

/* Define an indicator struct to correspond
   to the host output struct. */
struct
{
    short     emp_name_ind;
    short     sal_ind;
    short     comm_ind;
} emprec_ind;

/*  Input host variable. */
int         emp_number;

int         total_queried;

/* Include the SQL Communications Area.
   You can use #include or EXEC SQL INCLUDE. */
#include <sqlca.h>


/* Declare error handling function. */
void sql_error();


main()
{
    char temp_char[32];

/* Connect to ORACLE––
 * Copy the username into the VARCHAR.
 */
    strncpy((char *) username.arr, ”SCOTT”, UNAME_LEN);

/* Set the length component of the VARCHAR. */
    username.len = strlen((char *) username.arr);

/* Copy the password. */
    strncpy((char *) password.arr, ”TIGER”, PWD_LEN);
    password.len = strlen((char *) password.arr);
```

```
/* Register sql_error() as the error handler. */
    EXEC SQL WHENEVER SQLERROR DO sql_error(”ORACLE error––\n”);

/* Connect to ORACLE.  Program will call sql_error()
 * if an error occurs when connecting to the default database.
 */
    EXEC SQL CONNECT :username IDENTIFIED BY :password;

    printf(”\nConnected to ORACLE as user: %s\n”, username.arr);

/* Loop, selecting individual employee’s results */

    total_queried = 0;

    for (;;)
    {
/* Break out of the inner loop when a
 * 1403 (”No data found”) condition occurs.
 */
        EXEC SQL WHENEVER NOT FOUND DO break;
        for (;;)
        {
            emp_number = 0;
            printf(”\nEnter employee number (0 to quit): ”);
            gets(temp_char);
            emp_number = atoi(temp_char);
            if (emp_number == 0)
                break;

            EXEC SQL SELECT ename, sal, comm
                INTO :emprec INDICATOR :emprec_ind
                FROM EMP
                WHERE EMPNO = :emp_number;

/* Print data. */
            printf(”\n\nEmployee\tSalary\t\tCommission\n”);
            printf(”–––––––––\t––––––\t\t––––––––––\n”);
/* Null–terminate the output string data. */
            emprec.emp_name.arr[emprec.emp_name.len] = ’\0’;
            printf(”%–8s\t%6.2f\t\t”,
                emprec.emp_name.arr, emprec.salary);

            if (emprec_ind.comm_ind == –1)
                printf(”NULL\n”);
            else
                printf(”%6.2f\n”, emprec.commission);

            total_queried++;
        }  /* end inner for (;;) */
        if (emp_number == 0) break;
        printf(”\nNot a valid employee number – try again.\n”);
    } /* end outer for(;;) */

    printf(”\n\nTotal rows returned was %d.\n”, total_queried);
    printf(”\nG’day.\n\n\n”);

/* Disconnect from ORACLE. */
    EXEC SQL COMMIT WORK RELEASE;
    exit(0);
}


void
sql_error(msg)
char *msg;
{
    char err_msg[128];
    int buf_len, msg_len;

    EXEC SQL WHENEVER SQLERROR CONTINUE;

    printf(”\n%s\n”, msg);
    buf_len = sizeof (err_msg);
    sqlglm(err_msg, &buf_len, &msg_len);
    printf(”%.*s\n”, msg_len, err_msg);

    EXEC SQL ROLLBACK RELEASE;
    exit(1);
}
```

CHAPTER

3

Developing a Pro*C/C++ Application

This chapter provides the basic information you need to write a Pro*C/C++ program. This chapter covers the following topics:

- how to use the Pro*C/C++ preprocessor
- how to migrate a Pro*C/C++ application from earlier releases
- how to declare and use host variables
- how to declare and use indicator variables
- how to use Oracle datatypes
- how to equivalence datatypes
- how to connect to Oracle

This chapter also includes several complete demonstration programs that you can study. These programs illustrate the techniques described. They are available on–line in your *demo* directory, so you can compile and run them, and modify them for your own uses.

The Pro*C/C++ Preprocessor

Release 2.2 of the Pro*C/C++ Precompiler incorporates a C parser and preprocessor. Pro*C/C++ supports most C preprocessor directives. Some of the things that you can do using the Pro*C/C++ preprocessor are

- define constants and macros using the **#define** directive, and use the defined entities to parameterize Pro*C/C++ datatype declarations, such as VARCHAR
- read files required by the precompiler, such as *sqlca.h*, using the **#include** directive
- define constants and macros in a separate file, and have the precompiler read this file using the **#include** directive

How the Pro*C/C++ Preprocessor Works

The Pro*C/C++ preprocessor recognizes most C preprocessor commands, and effectively performs the required macro substitutions, file inclusions, and conditional source text inclusions or exclusions. The Pro*C/C++ preprocessor uses the values obtained from preprocessing, but it only alters the source output text (the generated *.c* output file) in cases where the precompiler uses the values.

An example should clarify this point. Consider the following program fragment:

```
#include ”my_header.h”
...
VARCHAR name[VC_LEN];              /* a Pro*C–supplied datatype */

char    another_name[VC_LEN];              /* a pure C datatype */
...
```

Suppose the file *my_header.h* in the current directory contains, among other things, the line

```
#define VC_LEN   20
```

The precompiler reads the file *my_header.h*, and uses the defined value of VC_LEN (20) when it declares the structure for the VARCHAR *name*, replacing the defined constant VC_LEN with its value, 20.

However, the file *my_header.h* is not physically included in the precompiler source file by the Pro*C/C++ preprocessor. For this reason, VC_LEN in the declaration of the C **char** variable *another_name* is not replaced by the constant 20.

This does not matter, since the precompiler does not need to process declarations of C datatypes, even when they are used as host variables. It is left up to the C compiler's preprocessor to physically include the file *my_header.h*, and perform the substitution of 20 for VC_LEN in the declaration of *another_name*.

Preprocessor Directives

The preprocessor directives that Pro*C/C++ supports are:

- **#define**, to write macros for use by the precompiler and the C/C++ compiler
- **#include**, to read other source files for use by the precompiler
- **#ifdef**, to precompile and compile source text conditionally, depending on the existence of a defined constant
- **#ifndef**, to exclude source text conditionally
- **#endif**, to end an **#ifdef** or **#ifndef** command
- **#else**, to select an alternative body of source text to be precompiled and compiled, in case an **#ifdef** or **#ifndef** condition is not satisfied
- **#elif**, to select an alternative body of source text to be precompiled and compiled, depending on the value of a constant or a macro argument

Directives Ignored

Some C preprocessor directives are not used by the Pro*C/C++ preprocessor. Most of these directives are not relevant for the precompiler. For example, **#pragma** is a directive for the C compiler—the precompiler does not process C pragmas. The C preprocessor directives not processed by the precompiler are

- **#**, to convert a preprocessor macro parameter to a string constant
- **##**, to merge two preprocessor tokens in a macro definition
- **#error**, to produce a compile–time error message
- **#pragma**, to pass implementation–dependent information to the C compiler
- **#line**, to supply a line number for C compiler messages

While your C compiler preprocessor may support these directives, Pro*C/C++ does not use them. Most of these directives are not used by the precompiler. You can use these directives in your Pro*C/C++ program if your compiler supports them, but only in C/C++ code, not in embedded SQL statements or declarations of variables using datatypes supplied by the precompiler, such as VARCHAR.

ORA_PROC Macro

Pro*C/C++ predefines a C preprocessor macro called ORA_PROC that you can use to avoid having the precompiler process unnecessary or irrelevent sections of code. Some applications include large header files, which provide information that is unnecessary when precompiling. By conditionally including such header files based on the ORA_PROC macro, the precompiler never reads the file.

The following example uses the ORA_PROC macro to *ex*clude the *irrelevant.h* file:

```
#ifndef  ORA_PROC
#include <irrelevant.h>
#endif
```

Because ORA_PROC is defined during precompilation, the *irrelevant.h* file is never included.

The ORA_PROC macro is available only for C preprocessor directives, such as **#ifdef** or **#ifndef**. The EXEC ORACLE conditional statements do *not* share the same the namespaces as the C preprocessor macros. Therefore, the condition in the following example does *not* use the predefined ORA_PROC macro:

```
EXEC ORACLE IFNDEF ORA_PROC;
   <section of code to be ignored>
EXEC ORACLE ENDIF;
```

ORA_PROC, in this case, must be set using either the DEFINE option or an EXEC ORACLE DEFINE statement for this conditional code fragment to work properly.

Specifying the Location of Header Files

The Pro*C/C++ Precompiler for each port assumes a standard location for header files to be read by the preprocessor, such as *sqlca.h*, *oraca.h*, and *sqlda.h*. For example, on most UNIX ports, the standard location is *$ORACLE_HOME/proc/lib*. For the default location on your system, see your system–specific Oracle documentation. If header files that you need to include are not in the default location, you must use the INCLUDE= option, on the command line or as an EXEC ORACLE option. See Chapter 7 for more information about the precompiler options, and about the EXEC ORACLE options.

To specify the location of system header files, such as *stdio.h* or *iostream.h*, where the location might be different from that hard–coded into Pro*C/C++, use the SYS_INCLUDE precompiler option. See Chapter 7 for more information.

Some Preprocessor Examples

You can use the **#define** command to create named constants, and use them in place of "magic numbers" in your source code. You can use **#define** d constants for declarations that the precompiler requires, such as VARCHAR[]. For example, instead of potentially buggy code like

```
...
VARCHAR  emp_name[10];
VARCHAR  dept_loc[14];
...
...
/* much later in the code ... */
f42()
{
    /* did you remember the correct size? */
    VARCHAR new_dept_loc[10];
   ...
}
```

you can code

```
#define ENAME_LEN     10
#define LOCATION_LEN  14
VARCHAR  new_emp_name[ENAME_LEN];
   ...
/* much later in the code ... */
f42()
{
    VARCHAR new_dept_loc[LOCATION_LEN];
   ...
}
```

You can use preprocessor macros with arguments for objects that the precompiler must process, just as you can for C objects. For example:

```
#define ENAME_LEN    10
#define LOCATION_LEN 14
#define MAX(A,B)  ((A) > (B) ? (A) : (B))

   ...
f43()
{
    /* need to declare a temporary variable to hold either an
       employee name or a department location */
    VARCHAR  name_loc_temp[MAX(ENAME_LEN, LOCATION_LEN)];
   ...
}
```

You can use the **#include**, **#ifdef** and **#endif** preprocessor directives to conditionally include a file that the precompiler requires. For example:

```
#ifdef ORACLE_MODE
#   include <sqlca.h>
#else
    long SQLCODE;
#endif
```

Using #define

There are restrictions on the use of the **#define** preprocessor directive in Pro*C/C++. You cannot use the **#define** directive to create symbolic constants for use in *executable* SQL statements. The following *invalid* example demonstrates this:

```
#define RESEARCH_DEPT   40
...
EXEC SQL SELECT empno, sal
    INTO :emp_number, :salary /* host arrays */
    FROM emp
    WHERE deptno = RESEARCH_DEPT;  /* INVALID! */
```

The only declarative SQL statements where you can legally use a **#define**d macro are TYPE and VAR statements. So, for example, the following uses of a macro are legal in Pro*C/C++:

```
#define STR_LEN      40
...
typedef char asciiz[STR_LEN];
...
EXEC SQL TYPE asciiz IS STRING(STR_LEN) REFERENCE;
...
EXEC SQL VAR password IS STRING(STR_LEN) REFERENCE;
```

Other Preprocessor Restrictions

You cannot use the preprocessor commands **#** and **##** to create tokens that the precompiler must recognize. You can, of course, use these commands (if your C compiler's preprocessor supports them) in pure C code, that the precompiler does not have to process. Using the preprocessor command **##** is *not* valid in this example:

```
#define MAKE_COL_NAME(A)    col ## A
...
EXEC SQL SELECT MAKE_COL_NAME(1), MAKE_COL_NAME(2)
    INTO :x, :y
    FROM table1;
```

The example is invalid because the precompiler must be able to process the embedded SQL statement, and **##** is not supported by the Pro*C/C++ preprocessor.

An Important Restriction

Because of the way the Pro*C/C++ preprocessor handles the **#include** directive, as described in the previous section, you cannot use the **#include** directive to include files that contain embedded SQL statements. You use **#include** to include files that contain purely declarative statements and directives; for example, **#define**s, and declarations of variables and structures required by the precompiler, such as in *sqlca.h*.

Including the SQLCA, ORACA, and SQLDA

You can include the *sqlca.h*, *oraca.h*, and *sqlda.h* declaration header files in your Pro*C/C++ program using either the Pro*C/C++ preprocessor **#include** command, or the EXEC SQL INCLUDE command. For example, you use the following statement to include the SQL Communications Area structure (SQLCA) in your program with the EXEC SQL option:

```
EXEC SQL INCLUDE sqlca;
```

To include the SQLCA using the Pro*C/C++ preprocessor directive, add the following code:

```
#include <sqlca.h>
```

When you use the preprocessor **#include** directive, you must specify the file extension (such as *.h*).

> **Note:** If you need to include the SQLCA in multiple places, using the **#include** directive, you should precede the **#include** with the directive **#undef** SQLCA. This is because *sqlca.h* starts with the lines
>
> ```
> #ifndef SQLCA
> #define SQLCA 1
> ```
>
> and then declares the SQLCA struct only in the case that SQLCA is not defined.

When you precompile a file that contains a **#include** directive or an EXEC SQL INCLUDE statement, you have to tell the precompiler the location of all files to be included. You can use the INCLUDE= option, either on the command line or in a system or user configuration file. See Chapter 7 for more information about the INCLUDE precompiler option and configuration files.

The default location for standard preprocessor include files, such as *sqlca.h*, *oraca.h*, and *sqlda.h*, is built into the precompiler. The location varies from system to system. See your system–specific Oracle documentation for the default location on your system.

When you compile the *.c* output file that Pro*C/C++ generates, you must use the option provided by your compiler and operating system to identify the location of **#include**'d files.

For example, on most UNIX systems, you can compile the generated C/C++ source file using the command

```
cc –o progname –I$ORACLE_HOME/sqllib/public ... filename.c ...
```

On VAX/OPENVMS systems, you would prepend the include directory path to the value in the logical VAXC$INCLUDE.

EXEC SQL INCLUDE and #include Summary

When you use an EXEC SQL INCLUDE statement in your program, the precompiler *textually* includes the source in the output (*.c*) file. So, you can have declarative and executable embedded SQL statements in a file that is included using EXEC SQL INCLUDE.

When you include a file using **#include**, the precompiler merely reads the file, and keeps track of **#define** d macros.

⚠ **Warning:** VARCHAR declarations and SQL statements are NOT allowed in #**included** files. For this reason, you cannot have SQL statements in files that are included using the Pro*C/C++ preprocessor **#include** directive.

Migrating an Application

Release 2.2 of the Pro*C/C++ Precompiler is compatible with earlier Pro*C and Pro*C/C++ releases. However, there are several things that you should consider when you migrate your application from Pro*C release 1.6, or an earlier Pro*C/C++ release, to Pro*C/C++ release 2.2. This section discusses some of these issues.

Character Strings

Many applications have been written under the assumption that character strings are of varying length (such as VARCHAR2). By default, Oracle7 uses fixed–length character strings (CHAR), to conform to the current SQL standards.

If your application assumes that character strings are varying in length (and this is especially important in the string comparison semantics), then you should precompile your application using the DBMS=V6 or DBMS=V6_CHAR options. DBMS=V6 provides Oracle V6 semantics in several areas, not just character string semantics. V6_CHAR gives you complete Oracle7 semantics *except* for character strings, which retain Oracle V6 semantics.

Note: The DBMS option partially replaces the MODE option of the release 1.5 and 1.6 Pro*C Precompilers.

See the description of the DBMS option on page 7 – 13 for a complete list of the effects of the DBMS=V6 and DBMS=V6_CHAR options.

Precompiler Options

The following precompiler options, which are simply parsed and ignored, are not supported in Pro*C / C++ release 2.2:

- ASACC
- IRECLEN
- LRECLEN
- ORECLEN
- PAGELEN
- TEST
- XREF

The AREASIZE, REBIND, and REENTRANT options have not been supported since Pro*C release 1.4. If you use them on the command line, the fatal error PCC–F–02041 is returned by Pro*C / C++. The other options (those listed above) are quietly accepted by the precompiler option processor but have no effect. For more information, see "Obsolete Options" on page 7 – 38.

Defined Macros

If you define macros on the C compiler's command line, you might also have to define these macros on the precompiler command line, depending on the requirements of your application. For example, if you C compile with a UNIX command line such as

```
cc –DDEBUG ...
```

you should precompile using the DEFINE= option, namely

```
proc DEFINE=DEBUG ...
```

Error Message Codes

Error and warning codes (PCC errors) are different between Pro*C release 1.6 and Pro*C / C++ release 2.2. See the *Oracle7 Server Messages* manual for a complete list of PCC codes and messages.

The runtime messages issued by SQLLIB now have the prefix SQL–, rather than the RTL– prefix used in earlier Pro*C / C++ and Pro*C releases. The message codes remain the same as those of earlier releases.

Include Files

Because Pro*C / C++ does a complete C parse of the input file, which involves a preprocessing step, the location of all included files that need to be precompiled must be specified on the command line, or in a configuration file. (See Chapter 7 for complete information about precompiler options and configuration files.)

For example, if you are developing under UNIX, and your application includes files in the directory */home/project42/include*, you must specify this directory both on the Pro*C / C++ command line and on the *cc* command line. You use commands like these:

```
proc iname=my_app.pc include=/home/project42/include . . .
cc –I/home/project42/include . . . my_app.c
```

or you would include the appropriate macros in a *makefile*. For complete information about compiling and linking your Pro*C / C++ release 2.2 application, see your system–specific Oracle documentation.

Output Files

With Pro*C releases 1.6 and earlier, the output file (.c) is generated in the working directory, i.e., the directory from which the precompiler command line is issued. Beginning with Pro*C release 2.0 and continuing with Pro*C / C++ release 2.2, the output file is generated in the same directory as the input file.

To generate an output file in a different directory from the input file, use the ONAME option to explicitly specify the desired location of your output file. For more information, see "ONAME" on page 7 – 29.

Indicator Variables

If you are migrating an application from Pro*C release 1.3 or release 1.4, used with Oracle V6, to Oracle7, there is a major change in behavior if you do not use indicator variables. Oracle V6 does not return an error if you SELECT or FETCH a null into a host variable that has no associated indicator variable. With Oracle7, the normal behavior is that SELECTing or FETCHing a null into a host variable that has no associated indicator variable does cause an error.

The error code is –01405 in SQLCODE and ''22002'' in SQLSTATE.

To avoid this error without recoding your application, you can specify DBMS=V6, or you can specify UNSAFE_NULL=YES (as described on page 7 – 36) with DBMS=V7 or V6_CHAR and MODE=ORACLE. See the description of the DBMS option on page 7 – 13 for complete information. However, Oracle recommends that you always use indicator variables in new Pro*C / C++ applications.

Programming Guidelines

This section deals with embedded SQL syntax, coding conventions, and C–specific features and restrictions. Topics are arranged alphabetically for quick reference.

C++ Support

Release 2.2 of the Pro*C / C++ Precompiler can optionally generate code that can be compiled using supported C++ compilers. See Chapter 6 for a complete explanation of this capability.

Comments

You can place C–style comments (/* ... */) in a SQL statement wherever blanks can be placed (except between the keywords EXEC SQL). Also, you can place ANSI–style comments (– – ...) *within* SQL statements at the end of a line, as the following example shows:

```
EXEC SQL SELECT ENAME, SAL
    INTO :emp_name, :salary  -- output host variables
    FROM EMP
    WHERE DEPTNO = :dept_number;
```

You cannot nest C–style comments.

You can use C++ style comments (/ /) in your Pro*C / C++ source if you precompile using the CODE=CPP precompiler option.

Constants

An *L* or *l* suffix specifies a **long** integer constant, a *U* or *u* suffix specifies an **unsigned** integer constant, a *0X* or *0x* prefix specifies a hexadecimal integer constant, and an *F* or *f* suffix specifies a **float** floating–point constant. These forms are *not* allowed in SQL statements.

Delimiters

While C uses apostrophes to delimit single characters, as in

```
ch = getchar();
switch (ch)
{
case ’U’: update();  break;
case ’I’: insert();  break;
...
```

SQL uses apostrophes to delimit character strings, as in

```
EXEC SQL SELECT ENAME, SAL FROM EMP WHERE JOB = ’MANAGER’;
```

While C uses double quotes to delimit character strings, as in

```
printf(”\nG’Day, mate!”);
```

SQL uses quotation marks to delimit identifiers containing special or lowercase characters, as in

```
EXEC SQL CREATE TABLE ”Emp2” (empno  number(4), ...);
```

Embedded SQL Syntax

The maximum number of open cursors per user process is set by the Oracle initialization parameter OPEN_CURSORS. You can override this parameter by using MAXOPENCURSORS to specify a lower (but not higher) value. The only special requirement for building SQL commands into your Pro*C/C++ program is that you preface them with the EXEC SQL clause. The Pro*C/C++ Precompiler translates all EXEC SQL statements into Oracle calls recognizable by your C compiler.

File Length

The Pro*C/C++ Precompiler cannot process arbitrarily long source files. Some of the variables used internally limit the size of the generated file. There is a limit to the number of lines allowed; the following aspects of the source file are contributing factors to the file–size constraint:

- complexity of the embedded SQL statements (for example, the number of bind and define variables)
- whether a database name is used (for example, connecting to a database with an AT clause)
- number of embedded SQL statements

To prevent problems related to this limitation, use multiple program units to sufficiently reduce the size of the source files.

Function Prototyping

The ANSI C standard (X3.159–1989) provides for function prototyping. A *function prototype* declares a function and the datatypes of its arguments, so that the C compiler can detect missing or mismatched arguments.

The CODE option, which you can enter on the command line or in a configuration file, determines the way that the Pro*C precompiler generates C or C++ code.

When you precompile your program with CODE=ANSI_C, the precompiler generates fully prototyped function declarations; for example:

```
extern void sqlora(long *, void *);
```

When you precompile with the option CODE=KR_C (KR for “Kernighan and Ritchie”—the default), the precompiler generates function prototypes in the same way that it does for ANSI_C, except that function parameter lists are commented out. For example:

```
extern void sqlora(/*_ long *, void *  _*/);
```

So, make sure to set the precompiler option CODE to KR_C if you use a C compiler that does not support ANSI C. When the CODE option is set to ANSI_C, the precompiler can also generate other ANSI–specific constructs; for example, the **const** type qualifier.

Host Variable Names

Host variable names can consist of upper or lowercase letters, digits, and underscores, but must begin with a letter. They can be any length, but only the first 31 characters are significant to the Pro*C/C++ Precompiler. Your C compiler or linker might require a shorter maximum length, so check your C compiler user's guide.

For SQL89 standards conformance, restrict the length of host variable names to 18 or fewer characters.

Line Continuation

You can continue SQL statements from one line to the next. You must use a backslash (\) to continue a string literal from one line to the next, as the following example shows:

```
EXEC SQL INSERT INTO dept (deptno, dname) VALUES (50, 'PURCHAS\
ING');
```

In this context, the precompiler treats the backslash as a continuation character.

MAXLITERAL Default Value

The precompiler option MAXLITERAL lets you specify the maximum length of string literals generated by the precompiler, so that compiler limits are not exceeded. The MAXLITERAL default value is 1024. But, you might have to specify a lower value. For example, if your C compiler cannot handle string literals longer than 512 characters, you would specify MAXLITERAL=512. Check your C compiler user's guide.

Nulls

In C, a null statement is written as a single semicolon, strings are terminated by the null character ('\0'), and the pointer to null points to a special address (0).

In SQL, a null column value is simply one that is missing, unknown, or inapplicable; it equates neither to zero nor to a blank. So, use the NVL function to convert nulls to non–nulls, use the IS [NOT] NULL comparison operator to search for nulls, and use indicator variables to insert and test for nulls.

Operators

The logical operators and the “equal to” relational operator are different in C and SQL, as the list below shows. These C operators are *not* allowed in SQL statements.

SQL Operators	***C Operator***
NOT	!
AND	&&
OR	||
=	==

Nor are the following C operators:

Type	***C Operator***
address	&
bitwise	&, |, ^, ~
compound assignment	+=, –=, *=, etc.
conditional	?:
decrement	––
increment	++
indirection	*
modulus	%
shift	>>,<<

Statement Labels

You can associate standard C statement labels (*label_name*:) with SQL statements, as this example shows:

```
EXEC SQL WHENEVER SQLERROR GOTO connect_error;
...
connect_error:
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK RELEASE;
    printf(”\nInvalid username/password\n”);
    exit(1);
```

Label names can be any length, but only the first 31 characters are significant. Your C compiler might require a different maximum length. Check your C compiler user’s guide.

Statement Terminator

Embedded SQL statements are always terminated by a semicolon, as the following example shows:

```
EXEC SQL DELETE FROM emp WHERE deptno = :dept_number;
```

Host Variables

Host variables are the key to communication between your host program and Oracle. Typically, a precompiler program inputs data to Oracle, and Oracle outputs data to the program. Oracle stores input data in database columns, and stores output data in program host variables.

A host variable can be any arbitrary C expression that resolves to a scalar type. But, a host variable must be an *lvalue*. Host arrays of most host variables are also supported. See the section “Host Arrays” on page 3 – 28 for more information.

Declaring Host Variables

You declare a host variable according to the rules of C, specifying a C datatype supported by the Oracle program interface. You do not have to declare host variables in a special Declare Section. However, if you do not use a Declare Section, the FIPS flagger warns you about this, as the Declare Section is part of the SQL Standard.

The C datatype must be compatible with that of the source or target database column. Table 3 – 1 shows the C datatypes and the pseudotypes that you can use when declaring host variables. Only these datatypes can be used for host variables. Table 3 – 2 shows the compatible Oracle internal datatypes.

C Datatype or Pseudotype	Description
char	single character
char[n]	n–character array (string)
int	integer
short	small integer
long	large integer
float	floating–point number (usually single precision)
double	floating–point number (always double precision)
VARCHAR[n]	variable–length string

Table 3 – 1 C Datatypes for Host Variables

Internal Type	C Type	Description
VARCHAR2(Y) (Note 1)	char	single character
CHAR(X) (Note 1)	char[n] VARCHAR[n] int short long float double	n–byte character array n–byte variable–length character array integer small integer large integer floating–point number double–precision floating–point number
NUMBER NUMBER(P,S) (Note 2)	int short long float double char char[n] VARCHAR[n]	integer small integer large integer floating–point number double–precision floating–point number single character n–byte character array n–byte variable–length character array
DATE	char[n] VARCHAR[n]	n–byte character array n–byte variable–length character array
LONG	char[n] VARCHAR[n]	n–byte character array n–byte variable–length character array
RAW(X) (Note 1)	unsigned char[n] VARCHAR[n]	n–byte character array n–byte variable–length character array
LONG RAW	unsigned char[n] VARCHAR[n]	n–byte character array n–byte variable–length character array
ROWID	unsigned char[n] VARCHAR[n]	n–byte character array n–byte variable–length character array
MLSLABEL	unsigned char[n] VARCHAR[n]	n–byte character array n–byte variable–length character array

Notes:
1. X ranges from 1 to 255. 1 is the default value. Y ranges from 1 to 2000.
2. P ranges from 2 to 38. S ranges from –84 to 127.

Table 3 – 2 C–Oracle Datatype Compatibility

One–dimensional arrays of simple C types can also serve as host variables. For **char[n]** and VARCHAR[n], *n* specifies the maximum string length, *not* the number of strings in the array. Two–dimensional arrays are allowed only for **char[m][n]** and VARCHAR[m][n], where *m* specifies the number of strings in the array and *n* specifies the maximum string length.

Pointers to simple C types are supported. Pointers to **char[n]** and VARCHAR[n] variables should be declared as pointer to **char** or VARCHAR (with no length specification). Arrays of pointers, however, are not supported.

Storage–Class Specifiers

Pro*C lets you use the **auto**, **extern**, and **static** storage–class specifiers when you declare host variables. However, you cannot use the **register** storage–class specifier to store host variables, since the precompiler takes the address of host variables by placing an ampersand (&) before them. Following the rules of C, you can use the **auto** storage class specifier only within a block.

To comply with the ANSI C standard, the Pro*C Precompiler allows you to declare an **extern char[n]** host variable with or without a maximum length, as the following example shows:

```
extern char  protocol[15];
extern char  msg[];
```

However, you should always specify the maximum length. In the last example, if *msg* is an output host variable declared in one precompilation unit but defined in another, the precompiler has no way of knowing its maximum length. If you have not allocated enough storage for *msg* in the second precompilation unit, you might corrupt memory. (Usually, ‘‘enough” is the number of bytes in the longest column value that might be SELECTed or FETCHed into the host variable, plus one byte for a possible null terminator.)

If you neglect to specify the maximum length for an **extern char[ ]** host variable, the precompiler issues a warning message. Also, it assumes that the host variable will store a CHARACTER column value, which cannot exceed 255 characters in length. So, if you want to SELECT or FETCH a VARCHAR2 or a LONG column value of length greater than 255 characters into the host variable, you *must* specify a maximum length.

Type Qualifiers

You can also use the **const** and **volatile** type qualifiers when you declare host variables.

A **const** host variable must have a constant value, that is, your program cannot change its initial value. A **volatile** host variable can have its value changed in ways unknown to your program (by a device attached to the system, for instance).

Referencing Host Variables

You use host variables in SQL data manipulation statements. A host variable must be prefixed with a colon (:) in SQL statements but must not be prefixed with a colon in C statements, as the following example shows:

```
char    buf[15];
int     emp_number;
float   salary;
...
gets(buf);
emp_number = atoi(buf);

EXEC SQL SELECT sal INTO :salary FROM emp
     WHERE empno = :emp_number;
```

Though it might be confusing, you can give a host variable the same name as an Oracle table or column, as this example shows:

```
int     empno;
char    ename[10];
float   sal;
...
EXEC SQL SELECT ename, sal INTO :ename, :sal FROM emp
     WHERE empno = :empno;
```

Restrictions

A host variable name is a C identifier, hence it must be declared and referenced in the same upper/lower case format. It cannot substitute for a column, table, or other Oracle object in a SQL statement, and must not be an Oracle reserved word. See Appendix B for a list of Oracle reserved words and keywords.

A host variable must resolve to an address in the program. For this reason, function calls and numeric expressions cannot serve as host variables. The following code is *invalid*:

```
#define MAX_EMP_NUM    9000
...
int get_dept();
...
EXEC SQL INSERT INTO emp (empno, ename, deptno) VALUES
    (:MAX_EMP_NUM + 10, ’CHEN’, :get_dept());
```

Indicator Variables

You can associate every host variable with an optional indicator variable. An indicator variable must be defined as a 2–byte integer and, in SQL statements, must be prefixed with a colon and immediately follow its host variable (unless you use the keyword INDICATOR). If you are using Declare Sections, you must also declare indicator variables inside the Declare Sections.

Using the Keyword INDICATOR

To improve readability, you can precede any indicator variable with the optional keyword INDICATOR. You must still prefix the indicator variable with a colon. The correct syntax is

```
:host_variable INDICATOR :indicator_variable
```

which is equivalent to

```
:host_variable:indicator_variable
```

You can use both forms of expression in your host program.

An Example

Typically, you use indicator variables to assign nulls to input host variables and detect nulls or truncated values in output host variables. In the example below, you declare three host variables and one indicator variable, then use a SELECT statement to search the database for an employee number matching the value of host variable *emp_number*. When a matching row is found, Oracle sets output host variables *salary* and *commission* to the values of columns SAL and COMM in that row and stores a return code in indicator variable *ind_comm*. The next statements use *ind_comm* to select a course of action.

```
EXEC SQL BEGIN DECLARE SECTION;
    int    emp_number;
    float  salary, commission;
    short comm_ind;  /* indicator variable  */
EXEC SQL END DECLARE SECTION;
    char temp[16];
    float  pay;      /* not used in a SQL statement */
...
printf(”Employee number? ”);
gets(temp);
emp_number = atof(temp);
EXEC SQL SELECT SAL, COMM
    INTO :salary, :commission:ind_comm
    FROM EMP
    WHERE EMPNO = :emp_number;
if(ind_comm == –1)    /* commission is null */
    pay = salary;
else
    pay = salary + commission;
```

For more information about using indicator variables, see “Using Indicator Variables” on page 4 – 3.

Guidelines

The following guidelines apply to declaring and referencing indicator variables. An indicator variable must

- be declared explicitly (in the Declare Section if present) as a 2–byte integer
- be prefixed with a colon (:) in SQL statements
- immediately follow its host variable in SQL statements and PL / SQL blocks (unless preceded by the keyword INDICATOR)

An indicator variable must *not*

- be prefixed with a colon in host language statements
- follow its host variable in host language statements
- be an Oracle reserved word

Oracle Restrictions

When DBMS=V6, Oracle does not issue an error if you SELECT or FETCH a null into a host variable that is not associated with an indicator variable. However, when DBMS=V7, if you SELECT or FETCH a null into a host variable that has no indicator, Oracle issues the following error message:

```
ORA–01405: fetched column value is NULL
```

When precompiling with MODE=ORACLE and DBMS=V7 or V6_CHAR specified, you can specify UNSAFE_NULL=YES to disable the ORA–01405 message. For more information, see “UNSAFE_NULL” on page 7 – 36.

Host Structures

You can use a C structure to contain host variables. You reference a structure containing host variables in the INTO clause of a SELECT or a FETCH statement, and in the VALUES list of an INSERT statement. Every component of the host structure must be a legal Pro*C/C++ host variable, as defined in Table 3 – 1.

When a structure is used as a host variable, only the name of the structure is used in the SQL statement. However, each of the members of the structure sends data to Oracle, or receives data from Oracle on a query. The following example shows a host structure that is used to add an employee to the EMP table:

```
typedef struct
{
    char  emp_name[11]; /* one greater than column length */
    int   emp_number;
    int   dept_number;
    float salary;
} emp_record;
...
/* define a new structure of type ”emp_record” */
emp_record new_employee;

strcpy(new_employee.emp_name, ”CHEN”);
new_employee.emp_number = 9876;
new_employee.dept_number = 20;
new_employee.salary = 4250.00;

EXEC SQL INSERT INTO emp (ename, empno, deptno, sal)
    VALUES (:new_employee);
```

The order that the members are declared in the structure must match the order that the associated columns occur in the SQL statement, or in the database table if the column list in the INSERT statement is omitted.

For example, the following use of a host structure is *invalid*, and causes a runtime error:

```
struct
{
    int empno;
    float salary;          /* struct components in wrong order */
    char emp_name[10];
} emp_record;

...
SELECT empno, ename, sal
   INTO :emp_record FROM emp;
```

The example is wrong because the components of the structure are not declared in the same order as the associated columns in the select list. The correct form of the SELECT statement is

```
SELECT empno, sal, ename   /* reverse order of sal and ename */
    INTO :emp_record FROM emp;
```

Host Structures and Arrays

You can use host arrays as components of host structures. See the section ''Host Arrays'' on page 3 – 28 for more information about host arrays. In the following example, a structure containing arrays is used to INSERT three new entries into the EMP table:

```
struct
{
    char emp_name[3][10];
    int emp_number[3];
    int dept_number[3];
} emp_rec;
...
strcpy(emp_rec.emp_name[0], “ANQUETIL”);
strcpy(emp_rec.emp_name[1], “MERCKX”);
strcpy(emp_rec.emp_name[2], “HINAULT”);
emp_rec.emp_number[0] = 1964; emp_rec.dept_number[0] = 5;
emp_rec.emp_number[1] = 1974; emp_rec.dept_number[1] = 5;
emp_rec.emp_number[2] = 1985; emp_rec.dept_number[2] = 5;

EXEC SQL INSERT INTO emp (ename, empno, deptno)
    VALUES (:emp_rec);
```

Arrays of Structures

You cannot, however, use an array of structures as a host array. The following declaration of an array of structs is *illegal* as a host array:

```
struct
{
    char emp_name[11];
    int emp_number;
    int dept_number;
} emp_rec[3];
```

It *is* possible to use a single **struct** within an array of **struct**s as a host variable. The following code fragment shows an example of this:

```
struct employee
{
    char emp_name[11];
    int emp_number;
    int dept_number;
} emp_rec[3];
...
EXEC SQL SELECT ename, empno, deptno
    INTO :emp_rec[1]
    FROM emp;
...
```

PL/SQL Records

You cannot use a C **struct** as a host variable for a PL/SQL RECORD variable.

Nested Structures and Unions

You cannot nest host structures. The following example is *invalid*:

```
struct
{
    int emp_number;
    struct
    {
        float salary;
        float commission;
    } sal_info;            /* INVALID */
    int dept_number;
} emp_record;
...
EXEC SQL SELECT empno, sal, comm, deptno
    INTO :emp_record
    FROM emp;
```

Also, you cannot use a C **union** as a host structure, nor can you nest a **union** in a structure that is to be used as a host structure.

Host Indicator Structures

When you need to use indicator variables, but your host variables are contained in a host structure, you set up a second structure that contains an indicator variable for each host variable in the host structure. You must have an indicator for each host variable, even if you do not need it.

For example, suppose you declare a host structure *student_record* as follows:

```
struct
{
    char s_name[32];
    int s_id;
    char grad_date[9];
} student_record;
```

If you want to use the host structure in a query such as

```
EXEC SQL SELECT student_name, student_idno, graduation_date
    INTO :student_record
    FROM college_enrollment
    WHERE student_idno = 7200;
```

and you need to know if the graduation date can be NULL, then you must declare a separate host indicator structure. You declare this as

```
struct
{
    short s_name_ind;  /* indicator variables must be shorts */
    short s_id_ind;
    short grad_date_ind;
} student_record_ind;
```

You must have an indicator variable in the indicator structure for each component of the host structure, even though you might only be interested in the null / not null status of the *grad_date* component.

Reference the indicator structure in the SQL statement in the same way that you would a host indicator variable:

```
EXEC SQL SELECT student_name, student_idno, graduation_date
    INTO :student_record INDICATOR :student_record_ind
    FROM college_enrollment
    WHERE student_idno = 7200;
```

When the query completes, the null / not null status of each selected component is available in the host indicator structure.

> **Note:** This Guide conventionally names indicator variables and indicator structures by appending *_ind* to the host variable or structure name. However, the names of indicator variables are completely arbitrary. You can adopt a different convention, or use no convention at all.

Sample Program: Cursor and a Host Structure

The demonstration program in this section shows a query that uses an explicit cursor, selecting data into a host structure. This program is available on–line in the file *sample2.pc* in your *demo* directory.

```
/*
 *  sample2.pc
 *
 *  This program connects to ORACLE, declares and opens a cursor,
 *  fetches the names, salaries, and commissions of all
 *  salespeople, displays the results, then closes the cursor.
 */

#include <stdio.h>
#include <sqlca.h>

#define UNAME_LEN      20
#define PWD_LEN        40

/*
 * Use the precompiler typedef’ing capability to create
 * null–terminated strings for the authentication host
 * variables. (This isn’t really necessary––plain char *’s
 * would work as well. This is just for illustration.)
 */
typedef char asciiz[PWD_LEN];

EXEC SQL TYPE asciiz IS STRING(PWD_LEN) REFERENCE;
asciiz     username;
asciiz     password;

struct emp_info
{
    asciiz     emp_name;
    float      salary;
    float      commission;
};


/* Declare function to handle unrecoverable errors. */
void sql_error();


main()
{
    struct emp_info *emp_rec_ptr;

/* Allocate memory for emp_info struct. */
    if ((emp_rec_ptr =
        (struct emp_info *) malloc(sizeof(struct emp_info))) == 0)
    {
        fprintf(stderr, ”Memory allocation error.\n”);
        exit(1);
    }

/* Connect to ORACLE. */
    strcpy(username, ”SCOTT”);
    strcpy(password, ”TIGER”);

    EXEC SQL WHENEVER SQLERROR DO sql_error(”ORACLE error––”);

    EXEC SQL CONNECT :username IDENTIFIED BY :password;
    printf(”\nConnected to ORACLE as user: %s\n”, username);

/* Declare the cursor. All static SQL explicit cursors
 * contain SELECT commands. ’salespeople’ is a SQL identifier,
 * not a (C) host variable.
 */
    EXEC SQL DECLARE salespeople CURSOR FOR
        SELECT ENAME, SAL, COMM
            FROM EMP
            WHERE JOB LIKE ’SALES%’;

/* Open the cursor. */
    EXEC SQL OPEN salespeople;

/* Get ready to print results. */
    printf(”\n\nThe company’s salespeople are––\n\n”);
    printf(”Salesperson   Salary   Commission\n”);
    printf(”–––––––––––   ––––––   ––––––––––\n”);

/* Loop, fetching all salesperson’s statistics.
 * Cause the program to break the loop when no more
 * data can be retrieved on the cursor.
 */
    EXEC SQL WHENEVER NOT FOUND DO break;

    for (;;)
    {
        EXEC SQL FETCH salespeople INTO :emp_rec_ptr;
        printf(”%–11s%9.2f%13.2f\n”, emp_rec_ptr–>emp_name,
                emp_rec_ptr–>salary, emp_rec_ptr–>commission);
    }

/* Close the cursor. */
    EXEC SQL CLOSE salespeople;

    printf(”\nArrivederci.\n\n”);
    EXEC SQL COMMIT WORK RELEASE;
    exit(0);
}



void
sql_error(msg)
char *msg;
{
    char err_msg[512];
    int buf_len, msg_len;

    EXEC SQL WHENEVER SQLERROR CONTINUE;

    printf(”\n%s\n”, msg);

/* Call sqlglm() to get the complete text of the
 * error message.
 */
    buf_len = sizeof (err_msg);
    sqlglm(err_msg, &buf_len, &msg_len);
    printf(”%.*s\n”, msg_len, err_msg);

    EXEC SQL ROLLBACK RELEASE;
    exit(1);
}
```

Host Arrays

Host arrays can increase performance by letting you manipulate an entire collection of data items with a single SQL statement. With few exceptions, you can use host arrays wherever scalar host variables are allowed. Also, you can associate an indicator array with any host array.

Declaring Host Arrays

You declare host arrays following the normal rules of C. The following example shows three arrays, dimensioned with 50 elements each:

```
int    emp_number[50];
/* array of 50 char arrays, each 11 bytes long */
char   emp_name[50][11];
float  salary[50];
```

Arrays of VARCHARs are also allowed. The following declaration is a valid host language declaration:

```
VARCHAR v_array[10][30];     /* a valid host array */
```

Restrictions

Host arrays of pointers are *not* allowed. Neither are multi–dimensional host arrays—with one exception. C does not have a string datatype, so you must declare an array of strings as a two–dimensional character array. The Pro*C/C++ Precompiler treats the two–dimensional character array as a one–dimensional array of strings.

The following declaration of a two–dimensional integer host array is *invalid* as a host array.

```
int   hi_lo_scores[25][25];   /* won’t work as a host array */
```

Referencing Host Arrays

If you use multiple host arrays in a single SQL statement, their dimensions should be the same. This is not a requirement, however, because the Pro*C/C++ Precompiler always uses the *smallest* dimension for the SQL operation. In this example, only 25 rows are INSERTed:

```
int    emp_number[50];
char   emp_name[50][10];
int    dept_number[25];
/* Populate host arrays here. */

EXEC SQL INSERT INTO emp (empno, ename, deptno)
    VALUES (:emp_number, :emp_name, :dept_number);
```

It is possible to subscript host arrays in SQL statements, and use them in a loop to INSERT or fetch data. For example, you could INSERT every fifth element in an array using a loop such as:

```
for (i = 0; i < 50; i += 5)
     EXEC SQL INSERT INTO emp (empno, deptno)
         VALUES (:emp_number[i], :dept_number[i]);
```

However, if the array elements that you need to process are contiguous, you should not process host arrays in a loop. Simply use the unsubscripted array names in your SQL statement. Oracle treats a SQL statement containing host arrays of dimension *n* like the same statement executed *n* times with *n* different scalar variables. For more information, see Chapter 10.

Using Indicator Arrays

You can use indicator arrays to assign nulls to input host arrays, and to detect null or truncated values in output host arrays. The following example shows how to INSERT with indicator arrays:

```
int    emp_number[50];
int    dept_number[50];
float  commission[50];
short  comm_ind[50];       /* indicator array */

/* Populate the host and indicator arrays.  To insert a null
   into the comm column, assign –1 to the appropriate
   element in the indicator array. */
    EXEC SQL INSERT INTO emp (empno, deptno, comm)
        VALUES (:emp_number, :dept_number,
        :commission INDICATOR :comm_ind);
```

Oracle Restrictions

Mixing scalar host variables with host arrays in the VALUES, SET, INTO, or WHERE clause is *not* allowed. If any of the host variables is an array, all must be arrays.

You cannot use host arrays with the CURRENT OF clause in an UPDATE or DELETE statement.

ANSI Restriction and Requirements

The array interface is an Oracle extension to the ANSI/ISO embedded SQL standard. However, when you precompile with MODE=ANSI, array SELECTs and FETCHes are still allowed. The use of arrays can be flagged using the FIPS flagger precompiler option, if desired.

When DBMS=V6, no error is generated if you SELECT or FETCH null columns into a host array that is not associated with an indicator array. Still, when doing array SELECTs and FETCHes, always use indicator arrays. That way, you can test for nulls in the associated output host array.

When you precompile with the precompiler options DBMS=V6_CHAR or DBMS=V7, if a null is selected or fetched into a host variable that has no associated indicator variable, Oracle stops processing, sets *sqlca.sqlerrd[2]* to the number of rows processed, and returns the following error:

```
ORA–01405: fetched column value is NULL
```

When DBMS=V6, if you SELECT or FETCH a truncated column value into a host array that is not associated with an indicator array, Oracle stops processing, sets *sqlca.sqlerrd[2]* to the number of rows processed, and issues the following error message:

```
ORA–01406: fetched column value was truncated
```

When DBMS=V6_CHAR or DBMS=V7, Oracle does not consider truncation to be an error.

Initialization

When MODE=ANSI, you should declare host variables inside a Declare Section. When you do use a Declare Section, you can initialize host arrays in the Declare Section. For instance, the following declaration is allowed:

```
EXEC SQL BEGIN DECLARE SECTION;
    int  dept_number[3] = {10, 20, 30};
EXEC SQL END DECLARE SECTION;
```

Sample Program: Host Arrays

The demonstration program in this section shows how you can use host arrays when writing a query in Pro*C/C++. Pay particular attention to the use of the "rows processed count" in the SQLCA (*sqlca.sqlerrd[2]*). See Chapter 9 for more information about the SQLCA. This program is available on–line in the file *sample3.pc* in your *demo* directory.

```
/*
 *  sample3.pc
 *  Host Arrays
 *
 *  This program connects to ORACLE, declares and opens a cursor,
 *  fetches in batches using arrays, and prints the results using
 *  the function print_rows().
 */

#include <stdio.h>
#include <string.h>

#include <sqlca.h>

#define NAME_LENGTH   20
#define ARRAY_LENGTH   5
```

```
/* Another way to connect. */
char *username = ”SCOTT”;
char *password = ”TIGER”;

/* Declare a host structure tag. */
struct
{
    int    emp_number[ARRAY_LENGTH];
    char   emp_name[ARRAY_LENGTH][NAME_LENGTH];
    float  salary[ARRAY_LENGTH];
} emp_rec;

/* Declare this program’s functions. */
void print_rows();              /* produces program output */
void sql_error();          /* handles unrecoverable errors */


main()
{
    int  num_ret;               /* number of rows returned */

/* Connect to ORACLE. */
    EXEC SQL WHENEVER SQLERROR DO sql_error(”Connect error:”);

    EXEC SQL CONNECT :username IDENTIFIED BY :password;
    printf(”\nConnected to ORACLE as user: %s\n”, username);


    EXEC SQL WHENEVER SQLERROR DO sql_error(”Oracle error:”);
/* Declare a cursor for the FETCH. */
    EXEC SQL DECLARE c1 CURSOR FOR
        SELECT empno, ename, sal FROM emp;

    EXEC SQL OPEN c1;

/* Initialize the number of rows. */
    num_ret = 0;

/* Array fetch loop – ends when NOT FOUND becomes true. */
    EXEC SQL WHENEVER NOT FOUND DO break;

    for (;;)
    {
        EXEC SQL FETCH c1 INTO :emp_rec;

/* Print however many rows were returned. */
        print_rows(sqlca.sqlerrd[2] – num_ret);
        num_ret = sqlca.sqlerrd[2];        /* Reset the number. */
    }
/* Print remaining rows from last fetch, if any. */
    if ((sqlca.sqlerrd[2] – num_ret) > 0)
        print_rows(sqlca.sqlerrd[2] – num_ret);

    EXEC SQL CLOSE c1;
    printf(”\nAu revoir.\n\n\n”);

/* Disconnect from the database. */
    EXEC SQL COMMIT WORK RELEASE;
    exit(0);
}


void
print_rows(n)
int n;
{
    int i;

    printf(”\nNumber   Employee         Salary”);
    printf(”\n––––––   ––––––––         ––––––\n”);

    for (i = 0; i < n; i++)
        printf(”%–9d%–15.15s%9.2f\n”, emp_rec.emp_number[i],
               emp_rec.emp_name[i], emp_rec.salary[i]);

}


void
sql_error(msg)
char *msg;
{
    EXEC SQL WHENEVER SQLERROR CONTINUE;

    printf(”\n%s”, msg);
    printf(”\n% .70s \n”, sqlca.sqlerrm.sqlerrmc);

    EXEC SQL ROLLBACK WORK RELEASE;
    exit(1);
}
```

Pointer Variables

C supports *pointers,* which “point” to other variables. A pointer holds the address (storage location) of a variable, not its value.

Declaring Pointer Variables

You define pointers as host variables following the normal C practice, as the next example shows:

```
int   *int_ptr;
char  *char_ptr;
```

Referencing Pointer Variables

In SQL statements, prefix pointers with a colon, as shown in the following example:

```
EXEC SQL SELECT intcol INTO :int_ptr FROM ...
```

Except for pointers to character strings, the size of the referenced value is given by the size of the base type specified in the declaration. For pointers to character strings, the referenced value is assumed to be a null–terminated string. Its size is determined at run time by calling the *strlen()* function. For details, see the section “Handling Character Data” on page 3 – 42.

You can use pointers to reference the members of a **struct**. First, declare a pointer host variable, then set the pointer to the address of the desired member, as shown in the example below. The datatypes of the **struct** member and the pointer variable must be the same. Most compilers will warn you of a mismatch.

```
struct
{
    int  i;
    char c;
} structvar;
int   *i_ptr;
char  *c_ptr;
...
main()
{
    i_ptr = &structvar.i;
    c_ptr = &structvar.c;
/* Use i_ptr and c_ptr in SQL statements. */
...
```

Structure Pointers

You can use a pointer to a structure as a host variable. The following example

- declares a structure
- declares a pointer to the structure
- allocates memory for the structure
- uses the struct pointer as a host variable in a query
- dereferences the struct components to print the results

```
struct EMP_REC
{
    int emp_number;
    float salary;
};
char *name = ”HINAULT”;
...
struct EMP_REC *sal_rec;
sal_rec = (struct EMP_REC *) malloc(sizeof (struct EMP_REC));
...
SELECT empno, sal INTO :sal_rec
    FROM emp
    WHERE ename = :name;

printf(”Employee number and salary for %s: ”, name);
printf(”%d, %g\n”, sal_rec–>emp_number, sal_rec–>salary);
```

In the SQL statement, pointers to host structures are referred to in exactly the same way as a host structure. The “address of” notation (**&**) is not required; in fact, it is an error to use it.

VARCHAR Variables

You can use the VARCHAR pseudotype to declare variable–length character strings. When your program deals with strings that are output from, or input to, VARCHAR2 or LONG columns, you might find it more convenient to use VARCHAR host variables instead of standard C strings. The datatype name VARCHAR can be uppercase or lowercase, but it cannot be mixed case. In this Guide, uppercase is used to emphasize that VARCHAR is not a native C datatype.

Declaring VARCHAR Variables

Think of a VARCHAR as an extended C type or predeclared **struct**. For example, the precompiler expands the VARCHAR declaration

```
VARCHAR   username[20];
```

into the following **struct** with array and length members:

```
struct
{
     unsigned short  len;
     unsigned char   arr[20];
} username;
```

The advantage of using VARCHAR variables is that you can explicitly reference the length member of the VARCHAR structure after a SELECT or FETCH. Oracle puts the length of the selected character string in the length member. You can then use this member to do things such as adding the null terminator

```
username.arr[username.len] = ’\0’;
```

or using the length in a *strncpy* or *printf* statement; for example:

```
printf(”Username is %.*s\n”, username.len, username.arr);
```

You specify the maximum length of a VARCHAR variable in its declaration. The length must lie in the range 1..65,533. For example, the following declaration is *invalid* because no length is specified:

```
VARCHAR   null_string[];    /* invalid */
```

The length member holds the current length of the value stored in the array member.

You can declare multiple VARCHARs on a single line; for example:

```
VARCHAR emp_name[ENAME_LEN], dept_loc[DEPT_NAME_LEN];
```

The length specifier for a VARCHAR can be a **#define**d macro, or any complex expression that can be resolved to an integer at precompile time.

You can also declare pointers to VARCHAR datatypes. See the section “Handling Character Data” on page 3 – 42.

Referencing VARCHAR Variables

In SQL statements, you reference VARCHAR variables using the **struct** name prefixed with a colon, as the following example shows:

```
...
int       part_number;
VARCHAR   part_desc[40];
...
main()
{
    ...
    EXEC SQL SELECT pdesc INTO :part_desc
        FROM parts
        WHERE pnum = :part_number;
    ...
```

After the query is executed, *part_desc.len* holds the actual length of the character string retrieved from the database and stored in *part_desc.arr*.

In C statements, you reference VARCHAR variables using the component names, as the next example shows:

```
printf(”\n\nEnter part description: ”);
gets(part_desc.arr);
/* You must set the length of the string
   before using the VARCHAR in an INSERT or UPDATE */
part_desc.len = strlen(part_desc.arr);
```

Returning Nulls to a VARCHAR Variable

Oracle automatically sets the length component of a VARCHAR output host variable. If you SELECT or FETCH a null into a VARCHAR, the server does not change the length or array members.

> **Note:** If you select a null into a VARCHAR host variable, and there is no associated indicator variable, an ORA–01405 error occurs at run time. Avoid this by coding indicator variables with all host variables. (As a temporary fix, use the DBMS precompiler option. See page 7 – 13).

Inserting Nulls Using VARCHAR Variables

If you set the length of a VARCHAR variable to zero before performing an UPDATE or INSERT statement, the column value is set to null. If the column has a NOT NULL constraint, Oracle returns an error.

Passing VARCHAR Variables to a Function

VARCHARs are structures, and most C compilers permit passing of structures to a function by value, and returning structures by copy out from functions. However, in Pro*C/C++ you must pass VARCHARs to functions by reference. The following example shows the correct way to pass a VARCHAR variable to a function:

```
VARCHAR emp_name[20];
...
emp_name.len = 20;
SELECT ename INTO :emp_name FROM emp
WHERE empno = 7499;
...
print_employee_name(&emp_name); /* pass by pointer */
...

print_employee_name(name)
VARCHAR *name;
{
    ...
    printf(”name is %.*s\n”, name–>len, name–>arr);
    ...
}
```

Finding the Length of the VARCHAR Array Component

When the precompiler processes a VARCHAR declaration, the actual length of the array element in the generated structure can be longer than that declared. For example, on a Sun Solaris 1.0 system, the Pro*C/C++ declaration

```
VARCHAR my_varchar[12];
```

is expanded by the precompiler to

```
struct my_varchar
{
    unsigned short len;
    unsigned char  arr[12];
};
```

However, the precompiler or the C compiler on this system pads the length of the array component to 14 bytes. This alignment requirement pads the total length of the structure to 16 bytes: 14 for the padded array and 2 bytes for the length.

The *sqlvcp()* function—part of the SQLLIB runtime library—returns the actual (possibly padded) length of the array member.

You pass the *sqlvcp()* function the length of the data for a VARCHAR host variable or a VARCHAR pointer host variable, and *sqlvcp()* returns the total length of the array component of the VARCHAR. The total length includes any padding that might be added by your C compiler.

The syntax of *sqlvcp()* is

```
sqlvcp(size_t *datlen, size_t *totlen);
```

Put the length of the VARCHAR in the first parameter before calling *sqlvcp()*. When the function returns, the second parameter contains the total length of the array element. Both parameters are pointers to long integers, so must be passed by reference.

Sample Program: Using *sqlvcp()*

The following sample program shows how you can use the *sqlvcp()* function in a Pro*C/C++ application. (The sample also uses the *sqlgls()* function, which is described in Chapter 9.) The sample declares a VARCHAR pointer, then uses the *sqlvcp()* function to determine the size required for the VARCHAR buffer. The program FETCHes employee names from the EMP table and prints them. Finally, the sample uses the *sqlgls()* function to print out the SQL statement and its function code and length attributes. This program is available on line as *sqlvcp.pc* in your *demo* directory.

```
/*
 *  The sqlvcp.pc program demonstrates how you can use the
 *  sqlvcp() function to determine the actual size of a
 *  VARCHAR struct. The size is then used as an offset to
 *  increment a pointer that steps through an array of
 *  VARCHARs.
 *
 *  This program also demonstrates the use of the sqlgls()
 *  function, to get the text of the last SQL statement executed.
 *  sqlgls() is described in the ”Error Handling” chapter of
 *  _The Programmer’s Guide to the Oracle Pro*C Precompiler_.
 */

#include <stdio.h>
#include <sqlca.h>
#include <sqlcpr.h>

/*  Fake a varchar pointer type. */

struct my_vc_ptr
{
    unsigned short len;
    unsigned char arr[32767];
};

/* Define a type for the varchar pointer */
typedef struct my_vc_ptr my_vc_ptr;
my_vc_ptr *vc_ptr;
```

```
EXEC SQL BEGIN DECLARE SECTION;
VARCHAR *names;
int      limit;    /* for use in FETCH FOR clause  */
char    *username = ”scott/tiger”;
EXEC SQL END DECLARE SECTION;
void sql_error();
extern void sqlvcp(), sqlgls();

main()
{
    unsigned int vcplen, function_code, padlen, buflen;
    int i;
    char stmt_buf[120];

    EXEC SQL WHENEVER SQLERROR DO sql_error();

    EXEC SQL CONNECT :username;
    printf(”\nConnected.\n”);

/*  Find number of rows in table. */
    EXEC SQL SELECT COUNT(*) INTO :limit FROM emp;


/*  Declare a cursor for the FETCH statement. */
    EXEC SQL DECLARE emp_name_cursor CURSOR FOR
    SELECT ename FROM emp;
    EXEC SQL FOR :limit OPEN emp_name_cursor;

/*  Set the desired DATA length for the VARCHAR. */
    vcplen = 10;

/*  Use SQLVCP to help find the length to malloc. */
    sqlvcp(&vcplen, &padlen);
    printf(”Actual array length of varchar is %ld\n”, padlen);

/*  Allocate the names buffer for names.
    Set the limit variable for the FOR clause. */
    names = (VARCHAR *) malloc((sizeof (short) +
    (int) padlen) * limit);
    if (names == 0)
    {
        printf(”Memory allocation error.\n”);
        exit(1);
    }
```

```
/*  Set the maximum lengths before the FETCH.
 *  Note the ”trick” to get an effective VARCHAR *.
 */
    for (vc_ptr = (my_vc_ptr *) names, i = 0; i < limit; i++)
    {
        vc_ptr–>len = (short) padlen;
        vc_ptr = (my_vc_ptr *)((char *) vc_ptr +
        padlen + sizeof (short));
    }
/*  Execute the FETCH. */
    EXEC SQL FOR :limit FETCH emp_name_cursor INTO :names;

/*  Print the results. */
    printf(”Employee names––\n”);

    for (vc_ptr = (my_vc_ptr *) names, i = 0; i < limit; i++)
    {
        printf
         (”%.*s\t(%d)\n”, vc_ptr–>len, vc_ptr–>arr, vc_ptr–>len);
        vc_ptr = (my_vc_ptr *)((char *) vc_ptr +
                  padlen + sizeof (short));
    }

/*  Get statistics about the most recent
 *  SQL statement using SQLGLS. Note that
 *  the most recent statement in this example
 *  is not a FETCH, but rather ”SELECT ENAME FROM EMP”
 *  (the cursor).
 */
    buflen = (long) sizeof (stmt_buf);

/*  The returned value should be 1, indicating no error. */
    sqlgls(stmt_buf, &buflen, &function_code);
    if (buflen != 0)
    {
        /* Print out the SQL statement. */
        printf(”The SQL statement was––\n%.*s\n”, buflen,
                                                  stmt_buf);

        /* Print the returned length. */
        printf(”The statement length is %ld\n”, buflen);

        /* Print the attributes. */
        printf(”The function code is %ld\n”, function_code);

        EXEC SQL COMMIT RELEASE;
        exit(0);
    }
    else
    {
        printf(”The SQLGLS function returned an error.\n”);
        EXEC SQL ROLLBACK RELEASE;
        exit(1);
    }
}

void
sql_error()
{
    char err_msg[512];
    int buf_len, msg_len;


    EXEC SQL WHENEVER SQLERROR CONTINUE;

    buf_len = sizeof (err_msg);
    sqlglm(err_msg, &buf_len, &msg_len);
    printf(”%.*s\n”, msg_len, err_msg);

    EXEC SQL ROLLBACK RELEASE;
    exit(1);
}
```

Handling Character Data

This section explains how the Pro*C/C++ Precompiler handles character host variables. There are four host variable character types:

- character arrays
- pointers to strings
- VARCHAR variables
- pointers to VARCHARs

Do not confuse VARCHAR (a host variable data structure supplied by the precompiler) with VARCHAR2 (an Oracle internal datatype for variable–length character strings).

Effect of the DBMS Option

The DBMS option, which you can specify on the command line, determines how Pro*C/C++ treats data in character arrays and strings. The DBMS option allows your program to observe compatibility with ANSI fixed–length strings, or to maintain compatibility with previous releases of Oracle and Pro*C that use variable–length strings. See Chapter 7 for a complete description of the DBMS option.

For character data, there are two ways that you can use the DBMS option:

- DBMS=V7
- DBMS=V6_CHAR

These choices are referred to in this section as Oracle7 character behavior, and Oracle V6 character behavior, respectively. V7 is the default setting, and is in effect when the DBMS option is not specified on the command line or in a configuration file. The DBMS option value V6 also provides V6 character semantics.

> **Note:** The DBMS option does not affect the way Pro*C handles VARCHAR host variables.

The DBMS option affects character data both on input (from your host variables to the Oracle table) and on output (from an Oracle table to your host variables).

On Input

Character Array
On input, the DBMS option determines the format that a host variable character array must have in your program. When the DBMS is set to V6_CHAR (or V6), host variable character arrays must be blank padded, and should not be null–terminated. When the DBMS=V7, character arrays must be null–terminated.

When the DBMS option is set to V6 or V6_CHAR, trailing blanks are stripped up to the first non–blank character before the value is sent to the database. Be careful! An uninitialized character array can contain null characters. To make sure that the nulls are not inserted into the table, you must blank–pad the character array to its length. For example, if you execute the statements

```
char emp_name[10];
...
strcpy(emp_name, “MILLER”);     /* WRONG! Note no blank-padding */
EXEC SQL INSERT INTO emp (empno, ename, deptno) VALUES
    (1234, :emp_name, 20);
```

you might find that the string “MILLER” was inserted as “MILLER\0\0\0\0” (with four null bytes appended to it). This value would not meet the following search condition:

```
. . . WHERE ename = ’MILLER’;
```

To INSERT the character array when DBMS is set to V6 or V6_CHAR, you should execute the statements

```
strncpy(emp_name, “MILLER    ”, 10); /* 4 trailing blanks */
EXEC SQL INSERT INTO emp (empno, ename, deptno) VALUES
    (1234, :emp_name, 20);
```

When DBMS=V7, input data in a character array must be null–terminated. So, make sure that your data ends with a null.

```
char emp_name[11];  /* Note: one greater than column size of 10 */
...
strcpy(emp_name, “MILLER”);        /* No blank-padding required */
EXEC SQL INSERT INTO emp (empno, ename, deptno) VALUES
    (1234, :emp_name, 20);
```

Character Pointer
The pointer must address a null–terminated buffer that is large enough to hold the input data. Your program must allocate this buffer and place the data in it before performing the input statement.

On Output

Character Array
On output, the DBMS option determines the format that a host variable character array will have in your program. When DBMS=V6_CHAR, host variable character arrays are blank padded up to the length of the array, but never null–terminated. When DBMS=V7, character arrays are blank padded, then null–terminated in the final position in the array.

Consider the following example of character output:

```
CREATE TABLE test_char (C_col CHAR(10), V_col VARCHAR2(10));

INSERT INTO test_char VALUES (’MILLER’, ’KING’);
```

A precompiler program to select from this table contains the following embedded SQL:

```
...
char name1[10];
char name2[10];
...
EXEC SQL SELECT C_col, V_col INTO :name1, :name2
    FROM test_char;
```

If you precompile the program with DBMS=V6_CHAR (or V6), *name1* will contain

```
“MILLER####”
```

that is, the name “MILLER” followed by 4 blanks, with no null–termination. (Note that if *name1* had been declared with a size of 15, there would be 9 blanks following the name.)

name2 will contain

```
“KING######”      /* 6 trailing blanks */
```

If you precompile the program with DBMS=V7, *name1* will contain

```
“MILLER###\0” /* 3 trailing blanks, then a null–terminator */
```

that is, a string containing the name, blank–padded to the length of the column, followed by a null terminator. *name2* will contain

```
“KING#####\0”   /* 5 trailing blanks, then a null terminator */
```

In summary, if DBMS=V6 or DBMS=V6_CHAR, the output from either a CHARACTER column or a VARCHAR2 column is blank–padded to the length of the host variable array. If DBMS=V7, the output string is always null–terminated.

Character Pointer

The DBMS option does not affect the way character data are output to a pointer host variable.

When you output data to a character pointer host variable, the pointer must point to a buffer large enough to hold the output from the table, plus one extra byte to hold a null terminator.

The precompiler runtime environment calls *strlen()* to determine the size of the output buffer, so make sure that the buffer does not contain any embedded nulls ('\0'). Fill allocated buffers with some value other than '\0', then null–terminate the buffer, before fetching the data.

> **Note:** C pointers can be used in a Pro*C program that is precompiled with DBMS=V7 and MODE=ANSI. However, pointers are not legal host variable types in a SQL standard compliant program. The FIPS flagger warns you if you use pointers as host variables.

The following code fragment uses the columns and table defined in the previous section, and shows how to declare and SELECT into character pointer host variables:

```
...
char *p_name1;
char *p_name2;
...
p_name1 = (char *) malloc(11);
p_name2 = (char *) malloc(11);
strcpy(p_name1, “          ”);
strcpy(p_name2, “0123456789”);

EXEC SQL SELECT C_col, V_col INTO :p_name1, :p_name2
    FROM test_char;
```

When the SELECT statement above is executed with DBMS=V7 *or* DBMS={V6 | V6_CHAR}, the value fetched is:

```
“MILLER####\0”     /* 4 trailing blanks and a null terminator */

“KING######\0”    /* 6 blanks and null */
```

VARCHAR Variables and Pointers

The following example shows how VARCHAR host variables are declared:

```
VARCHAR   emp_name1[10];   /* VARCHAR variable   */
VARCHAR  *emp_name2;       /* pointer to VARCHAR */
```

You cannot mix declarations of VARCHAR variables and VARCHAR pointers on the same line.

On Input

VARCHAR Variables

When you use a VARCHAR variable as an input host variable, your program need only place the desired string in the array member of the expanded VARCHAR declaration (*emp_name1.arr* in our example) and set the length member (*emp_name1.len*). There is no need to blank–pad the array. Exactly *emp_name1.len* characters are sent to Oracle, counting any blanks and nulls. In the following example, you set *emp_name1.len* to 8:

```
strcpy(emp_name1.arr, “VAN HORN”);
emp_name1.len = strlen(emp_name1.arr);
```

Pointer to a VARCHAR

When you use a pointer to a VARCHAR as an input host variable, you must allocate enough memory for the expanded VARCHAR declaration. Then, you must place the desired string in the array member and set the length member, as shown in the following example:

```
emp_name2 = malloc(sizeof(short) + 10)   /* len + arr */
strcpy(emp_name2–>arr, “MILLER”);
emp_name2–>len = strlen(emp_name2–>arr);
```

Or, to make *emp_name2* point to an existing VARCHAR (*emp_name1* in this case), you could code the assignment

```
emp_name2 = &emp_name1;
```

then use the VARCHAR pointer in the usual way, as in

```
EXEC SQL INSERT INTO EMP (EMPNO, ENAME, DEPTNO)
     VALUES (:emp_number, :emp_name2, :dept_number);
```

On Output

VARCHAR Variables

When you use a VARCHAR variable as an output host variable, the program interface sets the length member but does *not* null–terminate the array member. As with character arrays, your program can null–terminate the *arr* member of a VARCHAR variable before passing it to a function such as *printf()* or *strlen()*. An example follows:

```
emp_name1.arr[emp_name1.len] = ’\0’;
printf(”%s”, emp_name1.arr);
```

Or, you can use the length member to limit the printing of the string, as in:

```
printf(”%.*s”, emp_name1.len, emp_name1.arr);
```

An advantage of VARCHAR variables over character arrays is that the length of the value returned by Oracle is available right away. With character arrays, you might need to strip the trailing blanks yourself to get the actual length of the character string.

VARCHAR Pointers

When you use a pointer to a VARCHAR as an output host variable, the program interface determines the variable’s maximum length by checking the length member (*emp_name2–>len* in our example). So, your program must set this member before *every* fetch. The fetch then sets the length member to the actual number of characters returned, as the following example shows:

```
emp_name2–>len = 10;  /* Set maximum length of buffer. */
EXEC SQL SELECT ENAME INTO :emp_name2 WHERE EMPNO = 7934;
printf(”%d characters returned to emp_name2”, emp_name2–>len);
```

Oracle Datatypes

Oracle recognizes two kinds of datatypes: *internal* and *external*. Internal datatypes specify how Oracle stores column values in database tables, as well as the formats used to represent pseudocolumn values. External datatypes specify the formats used to store values in input and output host variables. For descriptions of the Oracle datatypes, see the *Oracle7 Server SQL Reference*.

Internal Datatypes

For values stored in database columns, Oracle uses the internal datatypes shown in Table 3 – 3.

Name	Description
VARCHAR2	variable–length character string, < 64Kbytes
NUMBER	numeric value, represented in a binary coded decimal format
LONG	variable–length character string <2**31–1 bytes
ROWID	binary value, internally 6 bytes in size
DATE	fixed–length date + time value, 7 bytes
RAW	variable–length binary data, <255 bytes
LONG RAW	variable–length binary data, <2**31–1 bytes
CHAR	fixed–length character string, < 255 bytes
MLSLABEL	tag for operating system label, 2–5 bytes

Table 3 – 3 Oracle Internal Datatypes

These internal datatypes can be quite different from C datatypes. For example, C has no datatype that is equivalent to the Oracle NUMBER datatype. However, NUMBERs can be converted between C datatypes such as **float** and **double**, with some restrictions. For example, the Oracle NUMBER datatype allows up to 38 decimal digits of precision, while no current C implementations can represent **double**s with that degree of precision. Also, the Oracle NUMBER datatype represents values exactly (within the precision limits), while floating–point formats cannot represent values such as 10.0 exactly.

External Datatypes

As shown in Table 3 – 4, the external datatypes include all the internal datatypes plus several datatypes that closely match C constructs. For example, the STRING external datatype refers to a C null–terminated string. There are also several external datatypes designed mainly for use with other languages, such as COBOL.

Name	Description
VARCHAR2	variable–length character string, ≤ 64Kbytes
NUMBER	decimal number, represented using a binary–coded floating–point format
INTEGER	signed integer
FLOAT	real number
STRING	null–terminated variable length character string
VARNUM	decimal number, like NUMBER, but includes representation length component
DECIMAL	COBOL packed decimal
LONG	fixed–length character string, up to 2**31–1 bytes
VARCHAR	variable–length character string, ≤65533 bytes
ROWID	binary value, external length is system dependent
DATE	fixed–length date/time value, 7 bytes
VARRAW	variable–length binary data, ≤65533 bytes
RAW	fixed–length binary data, ≤65533 bytes
LONG RAW	fixed–length binary data, ≤2**31–1 bytes
UNSIGNED	unsigned integer
DISPLAY	COBOL numeric character data
LONG VARCHAR	variable–length character string, ≤2**31–5 bytes
LONG VARRAW	variable–length binary data, ≤2**31–5 bytes
CHAR	fixed–length character string, ≤255 bytes
CHARZ	fixed–length, null–terminated character string, ≤65534 bytes
CHARF	used in TYPE or VAR statements to force CHAR to default to CHAR, instead of VARCHAR2
MLSLABEL	tag for operating system label, 2–5 bytes (Trusted Oracle only)

Table 3 – 4 Oracle External Datatypes

Brief descriptions of the external datatypes follow.

VARCHAR2

You use the VARCHAR2 datatype to store variable–length character strings. The maximum length of a VARCHAR2 value is 64Kbytes.

You specify the maximum length of a VARCHAR2(*n*) value in bytes, not characters. So, if a VARCHAR2(*n*) variable stores multibyte characters, its maximum length can be less than *n* characters.

When you precompile using the options DBMS=V6 or DBMS=V6_CHAR, Oracle assigns the VARCHAR2 datatype to all host variables that you declare as **char[n]** or **char.**

On Input

Oracle reads the number of bytes specified for the input host variable, strips any trailing blanks, then stores the input value in the target database column. Be careful. An uninitialized host variable can contain nulls. So, always blank–pad a character input host variable to its declared length, and do not null terminate it.

If the input value is longer than the defined width of the database column, Oracle generates an error. If the input value contains nothing but blanks, Oracle treats it like a null.

Oracle can convert a character value to a NUMBER column value if the character value represents a valid number. Otherwise, Oracle generates an error.

On Output

Oracle returns the number of bytes specified for the output host variable, blank–padding if necessary, then assigns the output value to the target host variable. If a null is returned, Oracle fills the host variable with blanks.

If the output value is longer than the declared length of the host variable, Oracle truncates the value before assigning it to the host variable. If there is an indicator variable associated with the host variable, Oracle sets it to the original length of the output value.

Oracle can convert NUMBER column values to character values. The length of the character host variable determines precision. If the host variable is too short for the number, scientific notation is used. For example, if you SELECT the column value 123456789 into a character host variable of length 6, Oracle returns the value '1.2E08'.

NUMBER

You use the NUMBER datatype to store fixed or floating–point Oracle numbers. You can specify precision and scale. The maximum precision of a NUMBER value is 38; the magnitude range is 1.0E–129 to 9.99E125. Scale can range from –84 to 127.

NUMBER values are stored in a variable–length format, starting with an exponent byte and followed by up to 20 mantissa bytes. The high–order bit of the exponent byte is a sign bit, which is set for positive numbers. The low–order 7 bits represent the exponent, which is a base–100 digit with an offset of 65.

Each mantissa byte is a base–100 digit in the range 1 .. 100. For positive numbers, 1 is added to the digit. For negative numbers, the digit is subtracted from 101, and, unless there are 20 mantissa bytes, a byte containing 102 is appended to the data bytes. Each mantissa byte can represent two decimal digits. The mantissa is normalized, and leading zeros are not stored. You can use up to 20 data bytes for the mantissa, but only 19 are guaranteed to be accurate. The 19 bytes, each representing a base–100 digit, allow a maximum precision of 38 digits.

On output, the host variable contains the number as represented internally by Oracle. To accommodate the largest possible number, the output host variable must be 21 bytes long. Only the bytes used to represent the number are returned. Oracle does not blank–pad or null–terminate the output value. If you need to know the length of the returned value, use the VARNUM datatype instead.

There is seldom a need to use this external datatype.

INTEGER

You use the INTEGER datatype to store numbers that have no fractional part. An integer is a signed, 2–byte or 4–byte binary number. The order of the bytes in a word is system dependent. You must specify a length for input and output host variables. On output, if the column value is a real number, Oracle truncates any fractional part.

FLOAT

You use the FLOAT datatype to store numbers that have a fractional part or that exceed the capacity of the INTEGER datatype. The number is represented using the floating–point format of your computer and typically requires 4 or 8 bytes of storage. You must specify a length for input and output host variables.

Oracle can represent numbers with greater precision than most floating–point implementations because the internal format of Oracle numbers is decimal. This can cause a loss of precision when fetching into a FLOAT variable.

STRING

The STRING datatype is like the VARCHAR2 datatype, except that a STRING value is always null–terminated.

On Input
Oracle uses the specified length to limit the scan for the null terminator. If a null terminator is not found, Oracle generates an error. If you do not specify a length, Oracle assumes the maximum length of 2000 bytes. The minimum length of a STRING value is 2 bytes. If the first character is a null terminator and the specified length is 2, Oracle inserts a null unless the column is defined as NOT NULL; if the column is defined as NOT NULL, an error occurs. An all–blank value is stored intact.

On Output
Oracle appends a null byte to the last character returned. If the string length exceeds the specified length, Oracle truncates the output value and appends a null byte. If a null is SELECTed, Oracle returns a null byte in the first character position.

VARNUM

The VARNUM datatype is like the NUMBER datatype, except that the first byte of a VARNUM variable stores the length of the representation.

On input, you must set the first byte of the host variable to the length of the value. On output, the host variable contains the length followed by the number as represented internally by Oracle. To accommodate the largest possible number, the host variable must be 22 bytes long. After SELECTing a column value into a VARNUM host variable, you can check the first byte to get the length of the value.

Normally, there is little reason to use this datatype.

DECIMAL

This type is normally used only with COBOL and PL/I programs. See the *Programmer's Guide to the Oracle Precompilers* for more information.

LONG

You use the LONG datatype to store fixed–length character strings. The LONG datatype is like the VARCHAR2 datatype, except that the maximum length of a LONG value is 2147483647 bytes or two gigabytes.

VARCHAR

You use the VARCHAR datatype to store variable–length character strings. VARCHAR variables have a 2–byte length field followed by a <65533–byte string field. However, for VARCHAR array elements, the maximum length of the string field is 65530 bytes. When you specify the length of a VARCHAR variable, be sure to include 2 bytes for the length field. For longer strings, use the LONG VARCHAR datatype.

ROWID

You can use the ROWID datatype to store binary rowids in (typically 13–byte) fixed–length fields. The field size is port specific. So, check your system–specific Oracle documentation.

You can use character host variables to store rowids in a readable format. When you SELECT or FETCH a rowid into a character host variable, Oracle converts the binary value to an 18–byte character string and returns it in the format

```
BBBBBBBB.RRRR.FFFF
```

where BBBBBBBB is the block in the database file, RRRR is the row in the block (the first row is 0), and FFFF is the database file. These numbers are hexadecimal. For example, the rowid

```
0000000E.000A.0007
```

points to the 11th row in the 15th block in the 7th database file.

Typically, you FETCH a rowid into a character host variable, then compare the host variable to the ROWID pseudocolumn in the WHERE clause of an UPDATE or DELETE statement. That way, you can identify the latest row fetched by a cursor. For an example, see the section "Mimicking CURRENT OF" on page 10 – 13.

> **Note:** If you need full portability or your application communicates with a non–Oracle database using Oracle Open Gateway technology, specify a maximum length of 256 (not 18) bytes when declaring the host variable. Though you can assume nothing about its contents, the host variable will behave normally in SQL statements.

DATE

You use the DATE datatype to store dates and times in 7–byte, fixed–length fields. As Table 3 – 5 shows, the century, year, month, day, hour (in 24–hour format), minute, and second are stored in that order from left to right.

Byte	1	2	3	4	5	6	7
Meaning	Century	Year	Month	Day	Hour	Minute	Second
Example 17–OCT–1994 at 1:23:12 PM	119	194	10	17	14	24	13

Table 3 – 5 DATE Format

The century and year bytes are in excess–100 notation. The hour, minute, and second are in excess–1 notation. Dates before the Common Era (B.C.E.) are less than 100. The epoch is January 1, 4712 B.C.E. For this date, the century byte is 53 and the year byte is 88. The hour byte ranges from 1 to 24. The minute and second bytes range from 1 to 60. The time defaults to midnight (1, 1, 1).

Normally, there is little reason to use this datatype.

VARRAW

You use the VARRAW datatype to store variable–length binary data or byte strings. The VARRAW datatype is like the RAW datatype, except that VARRAW variables have a 2–byte length field followed by a <65533–byte data field. For longer strings, use the LONG VARRAW datatype.

When you specify the length of a VARRAW variable, be sure to include 2 bytes for the length field. The first two bytes of the variable must be interpretable as an integer.

To get the length of a VARRAW variable, simply refer to its length field.

RAW

You use the RAW datatype to store binary data or byte strings. The maximum length of a RAW value is 255 bytes.

RAW data is like CHARACTER data, except that Oracle assumes nothing about the meaning of RAW data and does no character set conversions when you transmit RAW data from one system to another.

LONG RAW

You use the LONG RAW datatype to store binary data or byte strings. The maximum length of a LONG RAW value is 2147483647 bytes or two gigabytes.

LONG RAW data is like LONG data, except that Oracle assumes nothing about the meaning of LONG RAW data and does no character set conversions when you transmit LONG RAW data from one system to another.

UNSIGNED

You use the UNSIGNED datatype to store unsigned integers. An unsigned integer is a binary number of 2 or 4 bytes. The order of the bytes in a word is system dependent. You must specify a length for input and output host variables. On output, if the column value is a floating–point number, Oracle truncates the fractional part.

DISPLAY

With Pro*COBOL, you use the DISPLAY datatype to store numeric character data. The DISPLAY datatype refers to a COBOL “DISPLAY SIGN LEADING SEPARATE” number, which typically requires $n + 1$ bytes of storage for PIC S9(n), and $n + d + 1$ bytes of storage for PIC S9(n)V9(d).

LONG VARCHAR

You use the LONG VARCHAR datatype to store variable–length character strings. LONG VARCHAR variables have a 4–byte length field followed by a string field. The maximum length of the string field is 2147483643 (2**31 – 5) bytes. When you specify the length of a LONG VARCHAR for use in a VAR or TYPE statement, do not include the 4 length bytes.

LONG VARRAW

You use the LONG VARRAW datatype to store variable–length binary data or byte strings. LONG VARRAW variables have a 4–byte length field followed by a data field. The maximum length of the data field is 2147483643 bytes. . When you specify the length of a LONG VARRAW for use in a VAR or TYPE statement, do not include the 4 length bytes.

CHAR

You use the CHAR datatype to store fixed–length character strings. The maximum length of a CHAR value is 255 bytes.

On Input
Oracle reads the number of bytes specified for the input host variable, does *not* strip trailing blanks, then stores the input value in the target database column.

If the input value is longer than the defined width of the database column, Oracle generates an error. If the input value is all–blank, Oracle treats it like a character value.

On Output
Oracle returns the number of bytes specified for the output host variable, doing blank–padding if necessary, then assigns the output value to the target host variable. If a null is returned, Oracle fills the host variable with blanks.

If the output value is longer than the declared length of the host variable, Oracle truncates the value before assigning it to the host variable. If an indicator variable is available, Oracle sets it to the original length of the output value.

CHARZ

By default, when DBMS=V7, Oracle assigns the CHARZ datatype to all character host variables in a Pro*C program. The CHARZ datatype indicates fixed–length, null–terminated character strings. The maximum length of a CHARZ value is 255 bytes.

On input, the CHARZ and STRING datatypes work the same way. You must null–terminate the input value. The null terminator serves only to delimit the string; it does not become part of the stored data.

On output, CHARZ host variables are blank–padded if necessary, then null terminated. The output value is always null terminated, even if data must be truncated.

CHARF

The CHARF datatype is used in EXEC SQL TYPE and EXEC SQL VAR statements. When you precompile with the DBMS option set to V7 (the default setting), specifying the external datatype CHAR in a TYPE or VAR statement equivalences the C type or variable to the fixed–length, null–terminated datatype CHARZ. When you precompile with DBMS=V6 or DBMS=V6_CHAR, the C type or variable is equivalenced to VARCHAR2.

However, you might not want either of these type equivalences, but rather an equivalence to the fixed–length external type CHAR. If you use the external type CHARF, the C type or variable is *always* equivalenced to the fixed–length ANSI datatype CHAR, regardless of the DBMS value. CHARF never allows the C type to be equivalenced to VARCHAR2 or CHARZ.

MLSLABEL

You use the MLSLABEL datatype to store variable–length, binary operating system labels. Trusted Oracle uses labels to control access to data. For more information, see the *Trusted Oracle7 Server Administrator's Guide*.

You can use the MLSLABEL datatype to define a column. However, with standard Oracle, such columns can store nulls only. With Trusted Oracle, you can insert any valid operating system label into a column of type MLSLABEL.

On Input

Trusted Oracle translates the input value into a binary label, which must be a valid operating system label. If it is not, Trusted Oracle issues an error message. If the label is valid, Trusted Oracle stores it in the target database column

On Output

Trusted Oracle converts the binary label to a character string, which can be of type CHAR, CHARZ, STRING, VARCHAR, or VARCHAR2.

Datatype Conversion

At precompile time, a default external datatype is assigned to each host variable. For example, the precompiler assigns the INTEGER external datatype to host variables of type **short** and **int**.

At run time, the datatype code of every host variable used in a SQL statement is passed to Oracle. Oracle uses the codes to convert between internal and external datatypes.

Before assigning a SELECTed column (or pseudocolumn) value to an output host variable, Oracle must convert the internal datatype of the source column to the datatype of the host variable. Likewise, before assigning or comparing the value of an input host variable to a column, Oracle must convert the external datatype of the host variable to the internal datatype of the target column.

Conversions between internal and external datatypes follow the usual data conversion rules. For example, you can convert a CHAR value of "1234" to a C **short** value. But, you cannot convert a CHAR value of "65543" (number too large) or "10F" (number not decimal) to a C **short** value. Likewise, you cannot convert a **char[n]** value that contains any alphabetic characters to a NUMBER value.

Datatype Equivalencing

Datatype equivalencing lets you control the way Oracle interprets input data, and the way Oracle formats output data. It allows you to override the default external datatypes that the precompiler assigns. On a variable–by–variable basis, you can equivalence supported C host variable datatypes to Oracle external datatypes. You can also equivalence user–defined datatypes (that you **typedef**) to Oracle external datatypes.

Host Variable Equivalencing

By default, the Pro*C Precompiler assigns a specific external datatype to every host variable. Table 3 – 6 shows the default assignments:

C Type, or Pseudotype	Oracle External Type	
char	VARCHAR2	(DBMS=V6_CHAR)
char[n]	VARCHAR2 CHARZ	(DBMS=V7)
char*	CHARZ	
int, int*	INTEGER	
short, short*	INTEGER	
long, long*	INTEGER	
float, float*	FLOAT	
double, double*	FLOAT	
VARCHAR *, VARCHAR[n]	VARCHAR	

Table 3 – 6 Default Type Assignments

With the VAR statement, you can override the default assignments by equivalencing host variables to Oracle external datatypes. The syntax you use is

```
EXEC SQL VAR host_variable IS type_name [ length ];
```

where *host_variable* is an input or output host variable (or host array) declared earlier, *type_name* is the name of a valid external datatype, and *length* is an integer literal specifying a valid length in bytes.

Host variable equivalencing is useful in several ways. For example, suppose you want to SELECT employee names from the EMP table, then pass them to a routine that expects null–terminated strings. You need not explicitly null–terminate the names. Simply equivalence a host variable to the STRING external datatype, as follows:

```
...
char  emp_name[11];
EXEC SQL VAR emp_name IS STRING (11);
```

The length of the ENAME column in the EMP table is 10 characters, so you allot the new *emp_name* 11 characters to accommodate the null terminator. When you SELECT a value from the ENAME column into *emp_name*, the program interface null–terminates the value for you.

You can use any of the datatypes listed in the external datatypes table on page 3 – 49 except the following:

- NUMBER (if you must, use VARNUM instead)
- DECIMAL (COBOL only)
- DISPLAY (COBOL only)

User–Defined Type Equivalencing

You can also equivalence user–defined datatypes to Oracle external datatypes. First, define a new datatype structured like the external datatype that suits your needs. Then, equivalence your new datatype to the external datatype using the TYPE statement.

With the TYPE statement, you can assign an Oracle external datatype to a whole class of host variables. The syntax you use is

```
EXEC SQL TYPE user_type IS type_name [ ( length ) ] [REFERENCE];
```

Suppose you need a variable–length string datatype to hold graphics characters. First, declare a struct with a **short** length component followed by a ≤ 65533–byte data component. Second, use **typedef** to define a new datatype based on the struct. Then, equivalence your new user–defined datatype to the VARRAW external datatype, as shown in the following example:

```
struct  screen
{
    short  len;
    char   buff[4000];
};
typedef struct screen graphics;

EXEC SQL TYPE graphics IS VARRAW (4000);
graphics  crt;  – host variable of type graphics
     ...
```

You specify a length of 4000 bytes for the new *graphics* type because that is the maximum length of the data component in your struct. The precompiler allows for the *len* component (and any padding) when it sends the length to the Oracle server.

REFERENCE Clause

You can declare a user–defined type to be a pointer, either explicitly, as a pointer to a scalar or struct type, or implicitly, as an array, and use this type in an EXEC SQL TYPE statement. In this case, you must use the REFERENCE clause at the end of the statement, as shown in the following example:

```
typedef unsigned char *my_raw;

EXEC SQL TYPE my_raw IS VARRAW(4000) REFERENCE;
my_raw    graphics_buffer;
...

graphics_buffer = (my_raw) malloc(4004);
```

In this example, you allocated additional memory over and above the type length (4000). This is necessary because the precompiler also returns the length (the size of a **short**), and can add padding after the length due to alignment restrictions on your system. If you do not know the alignment practices on your system, make sure to allocate sufficient extra bytes for the length and padding (9 should usually be sufficient).

CHARF External Datatype

Release 1.6 of the Pro*C Precompiler introduced a new external datatype named CHARF, which is a fixed–length character string. You can use this new datatype in VAR and TYPE statements to equivalence C datatypes to the fixed–length SQL standard datatype CHAR, regardless of the setting of the DBMS option.

When DBMS=V7, specifying the external datatype CHARACTER in a VAR or TYPE statement equivalences the C datatype to the fixed–length datatype CHAR (datatype code 96). However, when DBMS=V6_CHAR, the C datatype is equivalenced to the variable–length datatype VARCHAR2 (code 1).

Now, you can always equivalence C datatypes to the fixed–length SQL standard type CHARACTER by using the CHARF datatype in the VAR or TYPE statement. When you use CHARF, the equivalence is always made to the fixed–length character type, regardless of the setting of the DBMS option.

Using the EXEC SQL VAR and TYPE Directives

You can code an EXEC SQL VAR ... or EXEC SQL TYPE ... statement anywhere in your program. These statements are treated as executable statements, that change the datatype of any variable affected by them from the point that the TYPE or VAR statement was made to the end of the scope of the variable. If you precompile with MODE=ANSI, you should use Declare Sections. In this case, the TYPE or VAR statement must be in a Declare Section.

Sample Program: Datatype Equivalencing

The demonstration program in this section shows you how you can use datatype equivalencing in your Pro*C programs. This program is available on–line in the file *sample4.pc* in your *demo* directory.

```
/***************************************************************
sample4.pc
This program demonstrates the use of type equivalencing using the
LONG RAW external datatype.  In order to provide a useful example
that is portable across different systems, the program inserts
binary files into and retrieves them from the database.  For
example, suppose you have a file called ’hello’ in the current
directory.  You can create this file by compiling the following
source code:

#include <stdio.h>

int
main()
{
  printf(”Hello World!\n”);
}

When this program is run, we get:

$hello
Hello World!

Here is some sample output from a run of sample4:

$sample4
Connected.
Do you want to create (or recreate) the EXECUTABLES table (y/n)? y
EXECUTABLES table successfully dropped.  Now creating new table...
EXECUTABLES table created.

Sample 4 Menu.  Would you like to:
(I)nsert a new executable into the database
(R)etrieve an executable from the database
(L)ist the executables currently stored in the database
(Q)uit the program

Enter i, r, l, or q: l

Executables currently stored:
----------- --------- ------

Total: 0

Sample 4 Menu.  Would you like to:
```

```
(I)nsert a new executable into the database
(R)etrieve an executable from the database
(L)ist the executables currently stored in the database
(Q)uit the program

Enter i, r, l, or q: i
Enter the key under which you will insert this executable: hello
Enter the filename to insert under key ’hello’.
If the file is not in the current directory, enter the full
path: hello
Inserting file ’hello’ under key ’hello’...
Inserted.

Sample 4 Menu.  Would you like to:
(I)nsert a new executable into the database
(R)etrieve an executable from the database
(L)ist the executables currently stored in the database
(Q)uit the program

Enter i, r, l, or q: l

Executables currently stored:
----------- --------- ------
hello

Total: 1

Sample 4 Menu.  Would you like to:
(I)nsert a new executable into the database
(R)etrieve an executable from the database
(L)ist the executables currently stored in the database
(Q)uit the program

Enter i, r, l, or q: r
Enter the key for the executable you wish to retrieve: hello
Enter the file to write the executable stored under key hello
into.  If you
don’t want the file to be in the current directory, enter the
full path: h1
Retrieving executable stored under key ’hello’ to file ’h1’...
Retrieved.

Sample 4 Menu.  Would you like to:
(I)nsert a new executable into the database
(R)etrieve an executable from the database
(L)ist the executables currently stored in the database
(Q)uit the program

Enter i, r, l, or q: q
```

```
We now have the binary file ’h1’ created, and we can run it:

$h1
Hello World!
****************************************************************/

#include <stdio.h>
#include <sys/types.h>
#include <sys/file.h>
#include <fcntl.h>
#include <string.h>
#include <sqlca.h>

/* Oracle error code for ’table or view does not exist’. */
#define NON_EXISTENT –942

/* This is the maximum size (in bytes) of a file that
 * can be inserted and retrieved.
 * If your system cannot allocate this much contiguous
 * memory, this value might have to be lowered.
 */
#define MAX_FILE_SIZE 500000


/* This is the definition of the long varraw structure.
 * Note that the first field, len, is a long instead
 * of a short.  This is becuase the first 4
 * bytes contain the length, not the first 2 bytes.
 */
typedef struct
{
    long len;
    char buf[MAX_FILE_SIZE];
} long_varraw;


/* Type Equivalence long_varraw to long varraw.
 * All variables of type long_varraw from this point
 * on in the file will have external type 95 (long varraw)
 * associated with them.
 */
EXEC SQL type long_varraw is long varraw (MAX_FILE_SIZE);


/* This program’s functions declared. */
void do_connect();
void create_table();
void sql_error();
void list_executables();
void print_menu();



main()
{
  char reply[20], key[20], filename[100];
  int ok = 1;

/* Connect to the database. */
  do_connect();

  printf(”Do you want to create (or recreate) the EXECUTABLES
table (y/n)? ”);
  gets(reply);

  if ((reply[0] == ’y’) || (reply[0] == ’Y’))
    create_table();

/* Print the menu, and read in the user’s selection. */
  print_menu();
  gets(reply);

  while (ok)
  {
    switch(reply[0]) {
    case ’I’: case ’i’:
/* User selected insert – get the key and file name. */
      printf(”
    Enter the key under which you will insert this executable: ”);
      gets(key);
      printf(
        ”Enter the filename to insert under key ’%s’.\n”, key);
      printf(
 ”If the file is not in the current directory, enter the full\n”);
      printf(”path: ”);
      gets(filename);
      insert(key, filename);
      break;
    case ’R’: case ’r’:
/* User selected retrieve – get the key and file name. */
      printf(
       ”Enter the key for the executable you wish to retrieve: ”);
      gets(key);
      printf(
      ”Enter the file to write the executable stored under key ”);
      printf(”%s into.  If you\n”, key);
      printf(
     ”don’t want the file to be in the current directory, enter
the\n”);
      printf(”full path: ”);
      gets(filename);
      retrieve(key, filename);
      break;
    case ’L’: case ’l’:
/* User selected list – just call the list routine. */
      list_executables();
      break;
    case ’Q’: case ’q’:
/* User selected quit – just end the loop. */
      ok = 0;
      break;
    default:
/* Invalid selection. */
      printf(”Invalid selection.\n”);
      break;
    }

    if (ok)
    {
/* Print the menu again. */
      print_menu();
      gets(reply);
    }
  }

  EXEC SQL commit work release;
}


/* Connect to the database. */
void
do_connect()
{

/* Note this declaration: uid is a char *
 * pointer, so Oracle will do a strlen() on it
 * at runtime to determine the length.
 */
  char *uid = ”scott/tiger”;

  EXEC SQL whenever sqlerror do sql_error(”Connect”);
  EXEC SQL connect :uid;

  printf(”Connected.\n”);
}
```

```
/* Creates the executables table. */
void
create_table()
{
/* We are going to check for errors ourselves
 * for this statement. */
  EXEC SQL whenever sqlerror continue;

  EXEC SQL drop table executables;
  if (sqlca.sqlcode == 0)
  {
    printf(”EXECUTABLES table successfully dropped.  ”);
    printf(”Now creating new table...\n”);
  }
  else if (sqlca.sqlcode == NON_EXISTENT)
  {
    printf(”EXECUTABLES table does not exist.  ”);
    printf(”Now creating new table...\n”);
  }
  else
    sql_error(”create_table”);

/* Reset error handler. */
  EXEC SQL whenever sqlerror do sql_error(”create_table”);

  EXEC SQL create table executables
    (name varchar2(20),
     binary long raw);

  printf(”EXECUTABLES table created.\n”);
}



/* Opens the binary file identified by ’filename’ for
 * reading, and copies it into ’buf’.
 * ’bufsize’ should contain the maximum size of
 * ’buf’.  Returns the actual length of the file read in,
 * or –1 if there is an error.
 */
int
read_file(filename, buf, bufsize)
char *filename, *buf;
long bufsize;
{

/* We will read in the file LOCAL_BUFFERSIZE bytes at a time. */
#define LOCAL_BUFFERSIZE 512
/* Buffer to store each section of the file. */
  char local_buffer[LOCAL_BUFFERSIZE];

/* Number of bytes read each time. */
  int number_read;

/* Total number of bytes read (the size of the file). */
  int total_size = 0;

/* File descriptor for the input file. */
  int in_fd;

/* Open the file for reading. */
  in_fd = open(filename, O_RDONLY, 0);
  if (in_fd == –1)
    return(–1);

/* While loop to actually read in the file,
 * LOCAL_BUFFERSIZE bytes at a time.
 */
  while ((number_read = read(in_fd, local_buffer,
          LOCAL_BUFFERSIZE)) > 0)
  {
    if (total_size + number_read > bufsize)
    {
/* The number of bytes we have read in so far exceeds the buffer
 * size – close the file and return an error. */
      close(in_fd);
      return(–1);
    }

/* Copy the bytes just read in from the local buffer
   into the output buffer. */
    memcpy(buf+total_size, local_buffer, number_read);

/* Increment the total number of bytes read by the number
   we just read. */
    total_size += number_read;
  }

/* Close the file, and return the total file size. */
  close(in_fd);
  return(total_size);
}


/* Generic error handler.  The ’routine’ parameter
 * should contain the name of the routine executing when
 * the error occured.  This would be specified in the
 * ’EXEC SQL whenever sqlerror do sql_error()’ statement.
 */
void
sql_error(routine)
char *routine;
{
  char message_buffer[512];
  int buffer_size;
  int message_length;

/* Turn off the call to sql_error() to avoid
 * a possible infinite loop.
 */
  EXEC SQL WHENEVER SQLERROR CONTINUE;

  printf(”\nOracle error while executing %s!\n”, routine);

/* Use sqlglm() to get the full text of the error message. */
  buffer_size = sizeof(message_buffer);
  sqlglm(message_buffer, &buffer_size, &message_length);
  printf(”%.*s\n”, message_length, message_buffer);

  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}


/* Opens the binary file identified by ’filename’ for
 * writing, and copies the contents of ’buf’ into it.
 * ’bufsize’ should contain the size of ’buf’.
 * Returns the number of bytes written (should be == bufsize),
 * or –1 if there is an error.
 */
int
write_file(filename, buf, bufsize)
char *filename, *buf;
long bufsize;
{
  int out_fd;        /* File descriptor for the output file. */
  int num_written;   /* Number of bytes written. */

/* Open the file for writing.  This command replaces
 * any existing version. */
  out_fd = creat(filename, 0755);
  if (out_fd == –1) {
/* Can’t create the output file – return an error. */
    return(–1);
  }
```

```
/* Write the contents of buf to the file. */
  num_written = write(out_fd, buf, bufsize);

/* Close the file, and return the number of bytes written. */
  close(out_fd);
  return(num_written);
}



/* Inserts the binary file identified by file into the
 * executables table identified by key.
 */
int
insert(key, file)
char *key, *file;
{
  long_varraw lvr;

  printf(”Inserting file ’%s’ under key ’%s’...\n”, file, key);
  lvr.len = read_file(file, lvr.buf, MAX_FILE_SIZE);
  if (lvr.len == –1)
  {
/* File size is too big for the buffer we have –
 * exit with an error.
 */
    fprintf(stderr,
       ”\n\nError while reading file ’%s’:\n”, file);
    fprintf(stderr,
      ”The file you selected to read is
        too large for the buffer.\n”);
    fprintf(stderr,
       ”Increase the MAX_FILE_SIZE macro in the source code,\n”);
    fprintf(stderr,
       ”reprecompile, compile, and link, and try again.\n”);
    fprintf(stderr,
       ”The current value of MAX_FILE_SIZE is %d bytes.\n”,
            MAX_FILE_SIZE);

    EXEC SQL rollback work release;

    exit(1);
  }

  EXEC SQL whenever sqlerror do sql_error(”insert”);
  EXEC SQL insert into executables (name, binary)
    values (:key, :lvr);
  EXEC SQL commit;
  printf(”Inserted.\n”);
}


/* Retrieves the executable identified by key into file */
int
retrieve(key, file)
char *key, *file;
{

/* Type equivalence key to the string external datatype.*/
  EXEC SQL VAR key is string(21);

  long_varraw lvr;
  short ind;
  int num_written;

  printf(”Retrieving executable stored under key ’%s’ to file
’%s’...\n”,
          key, file);

  EXEC SQL whenever sqlerror do sql_error(”retrieve”);
  EXEC SQL select binary
    into :lvr :ind
    from executables
    where name = :key;

  num_written = write_file(file, lvr.buf, lvr.len);
  if (num_written != lvr.len) {
/* Error while writing – exit with an error. */
    fprintf(stderr,
       ”\n\nError while writing file ’%s’:\n”, file);
    fprintf(stderr,
       ”Can’t create the output file.  Check to be sure that
you\n”);
    fprintf(stderr,
       ”have write permissions in the directory into which
you\n”);
    fprintf(stderr,
       ”are writing the file, and that there is enough disk
space.\n”);

    EXEC SQL rollback work release;

    exit(1);
  }

  printf(”Retrieved.\n”);
```

```
}

void
list_executables()
{
  char key[21];
/* Type equivalence key to the string external
 * datatype, so we don’t have to null–terminate it.
 */
  EXEC SQL VAR key is string(21);

  EXEC SQL whenever sqlerror do sql_error(”list_executables”);

  EXEC SQL declare key_cursor cursor for
    select name from executables;

  EXEC SQL open key_cursor;

  printf(”\nExecutables currently stored:\n”);
  printf(”––––––––––– ––––––––– ––––––\n”);

  while (1)
  {
    EXEC SQL whenever not found do break;
    EXEC SQL fetch key_cursor into :key;

    printf(”%s\n”, key);
  }

  EXEC SQL whenever not found continue;

  EXEC SQL close key_cursor;

  printf(”\nTotal: %d\n”, sqlca.sqlerrd[2]);
}

/* Prints the menu selections. */
void
print_menu()
{
  printf(”\nSample 4 Menu.  Would you like to:\n”);
  printf(”(I)nsert a new executable into the database\n”);
  printf(”(R)etrieve an executable from the database\n”);
  printf(”(L)ist the executables currently stored in the
database\n”);
  printf(”(Q)uit the program\n\n”);
  printf(”Enter i, r, l, or q: ”);
}
```

National Language Support

Although the widely–used 7– or 8–bit ASCII and EBCDIC character sets are adequate to represent the Roman alphabet, some Asian languages, such as Japanese, contain thousands of characters. These languages can require at least 16 bits (two bytes) to represent each character. How does Oracle deal with such dissimilar languages?

Oracle provides National Language Support (NLS), which lets you process single–byte and multibyte character data and convert between character sets. It also lets your applications run in different language environments. With NLS, number and date formats adapt automatically to the language conventions specified for a user session. Thus, NLS allows users around the world to interact with Oracle in their native languages.

You control the operation of language–dependent features by specifying various NLS parameters. Default values for these parameters can be set in the Oracle initialization file. Table 3 – 7 shows what each NLS parameter specifies.

NLS Parameter	Specifies ...
NLS_LANGUAGE	language–dependent conventions
NLS_TERRITORY	territory–dependent conventions
NLS_DATE_FORMAT	date format
NLS_DATE_LANGUAGE	language for day and month names
NLS_NUMERIC_CHARACTERS	decimal character and group separator
NLS_CURRENCY	local currency symbol
NLS_ISO_CURRENCY	ISO currency symbol
NLS_SORT	sort sequence

Table 3 – 7 NLS Parameters

The main parameters are NLS_LANGUAGE and NLS_TERRITORY. NLS_LANGUAGE specifies the default values for language–dependent features, which include

- language for Server messages
- language for day and month names
- sort sequence

NLS_LANGUAGE specifies the default values for territory–dependent features, which include

- date format
- decimal character
- group separator
- local currency symbol
- ISO currency symbol

You can control the operation of language–dependent NLS features for a user session by specifying the parameter NLS_LANG as follows

```
NLS_LANG = <language>_<territory>.<character set>
```

where *language* specifies the value of NLS_LANGUAGE for the user session, *territory* specifies the value of NLS_TERRITORY, and *character set* specifies the encoding scheme used for the terminal. An *encoding scheme* (usually called a character set or code page) is a range of numeric codes that corresponds to the set of characters a terminal can display. It also includes codes that control communication with the terminal.

You define NLS_LANG as an environment variable (or the equivalent on your system). For example, on UNIX using the C shell, you might define NLS_LANG as follows:

```
setenv NLS_LANG French_France.WE8ISO8859P1
```

To change the values of NLS parameters during a session, you use the ALTER SESSION statement as follows:

```
ALTER SESSION SET <nls_parameter> = <value>
```

The Pro*C Precompiler fully supports all the NLS features that allow your applications to process multilingual data stored in an Oracle database. For example, you can declare foreign–language character variables and pass them to string functions such as INSTRB, LENGTHB, and SUBSTRB. These functions have the same syntax as the INSTR, LENGTH, and SUBSTR functions, respectively, but operate on a per–byte basis rather than a per–character basis.

You can use the functions NLS_INITCAP, NLS_LOWER, and NLS_UPPER to handle special instances of case conversion. And, you can use the function NLSSORT to specify WHERE–clause comparisons based on linguistic rather than binary ordering. You can even pass NLS parameters to the TO_CHAR, TO_DATE, and TO_NUMBER functions. For more information about NLS, see the *Oracle7 Server Application Developer's Guide.*

Multi–byte Character Sets

The Pro*C/C++ Precompiler extends support for multi–byte NLS character sets through

- recognition of multi–byte character strings by the precompiler in embedded SQL statements.
- a Pro*C precompiler option that allows host character variables to be interpreted by the precompiler as multibyte character strings

The current release (Oracle7 Server 7.3 with Pro*C/C++ 2.2) supports multi–byte strings through the precompiler runtime library, SQLLIB.

Character Strings in Embedded SQL

A multibyte character string in an embedded SQL statement consists of a character literal that identifies the string as multibyte, immediately followed by the string. The string is enclosed in the usual single quotes.

For example, an embedded SQL statement like

```
EXEC SQL SELECT empno INTO :emp_num FROM emp
    WHERE ename=N'Kuroda';
```

contains a multibyte character string ('Kuroda' would actually be in Kanji), since the N character literal preceding the string 'Kuroda' identifies it as a multibyte string.

Dynamic SQL

Since dynamic SQL statements are not processed at precompile time, and since release 7.2 of the Oracle Server does not itself process multibyte strings, you cannot embed multibyte strings in dynamic SQL statements. If you do so, an error is returned by the server at runtime.

Host Variable Multibyte Character Strings

Release 2.1 of Pro*C does not provide character string host variables that directly support multibyte character sets. Instead, you use normal Pro*C character string host variables such as **char** or VARCHAR. You tell the precompiler which host variables should be interpreted as multibyte strings by using the NLS_CHAR precompiler option.

For more information, see the documentation of the following options in Chapter 7:

- DEF_SQLCODE
- NLS_CHAR
- NLS_LOCAL
- VARCHAR

Restrictions

You cannot use datatype equivalencing (the TYPE or VAR commands) with NLS multibyte character strings.

Dynamic SQL is not available for NLS multibyte character string host variables in Pro*C Release 2.1.

Blank Padding

When a Pro*C character variable is defined as an NLS multibyte variable, using the NLS_CHAR option, the following blank padding and blank stripping rules apply, depending on the external datatype of the variable. See the section "External Datatypes" on page 3 – 49.

CHARZ

This is the default character type when a character string is defined as multibyte. Input data must contain a null terminator at the end of the string (which does not become part of the entry in the database column). The string is stripped of any trailing double–byte spaces. However, if a string consists only of double–byte spaces, one double–byte space is left in the string.

Output host variables are padded with double–byte spaces, and contain a null terminator at the end of the string.

VARCHAR

On input, host variables are not stripped of trailing double–byte spaces. The length component of the structure is the length of the data in characters, not bytes.

On output, the host variable is not blank padded at all. The length of the buffer is set to the length of the data in characters, not in bytes.

STRING and LONG VARCHAR

These host variables are not supported for NLS data, since they can be specified only by datatype equivalencing or dynamic SQL, neither of which are supported for NLS data.

Indicator Variables

You can use indicator variables with host character variables that are NLS multibyte (as specified using the NLS_CHAR option).

Possible indicator values, and their meanings, are

0 The operation was successful.

–1 A null was returned, inserted, or updated.

–2 Output to a character host variable from a “long” type was truncated, but the original column length cannot be determined.

> 0 The result of a SELECT or FETCH into a character host variable was truncated. In this case, if the host variable is an NLS multibyte variable, the indicator value is the original column length in characters. If the host variable is not an NLS variable, then the indicator length is the original column length in bytes.

Cursor Variables

Starting with release 2.1 of the Pro*C/C++ Precompiler, you can use *cursor variables* in your Pro*C/C++ program for cursors for queries. A cursor variable is a handle for a cursor that must be defined and opened on the Oracle 7.2 or later server, using PL/SQL. See the *PL/SQL User's Guide and Reference* for complete information about cursor variables.

The advantages of cursor variables are:

- *Ease of maintenance*: queries are centralized, in the stored procedure that opens the cursor variable. If you need to change the cursor, you only need to make the change in one place: the stored procedure. There is no need to change each application.
- *Convenient security*: the user of the application is the username used when the Pro*C/C++ application connects to the server. The user must have *execute* permission on the stored procedure that opens the cursor but does not *read* permission on the tables used in the query. This capability can be used to limit access to the columns in the table, and access to other stored procedures.

Declaring a Cursor Variable

You declare a cursor variable in your Pro*C/C++ program using the Pro*C/C++ pseudotype SQL_CURSOR. For example:

```
EXEC SQL BEGIN DECLARE SECTION;
    sql_cursor     emp_cursor;             /* a cursor variable */
    SQL_CURSOR     dept_cursor;      /* another cursor variable */
    sql_cursor     *ecp;      /* a pointer to a cursor variable */
    ...
EXEC SQL END DECLARE SECTION;
ecp = &emp_cursor;             /* assign a value to the pointer */
```

You can declare a cursor variable using the type specification SQL_CURSOR, in all upper case, or sql_cursor, in all lower case; you cannot use mixed case.

A cursor variable is just like any other host variable in the Pro*C/C++ program. It has scope, following the scoping rules of C. You can pass it as a parameter to other functions, even functions external to the source file in which you declared it. You can also define functions that return cursor variables, or pointers to cursor variables.

> **Caution:** A SQL_CURSOR is implemented as a C **struct** in the code that Pro*C/C++ generates. So you can always pass it by pointer to another function, or return a pointer to a cursor variable from a function. But you can only pass it or return it by value if your C compiler supports these operations.

Allocating a Cursor Variable

Before you can use a cursor variable, either to open it or to FETCH using it, you must allocate the cursor. You do this using the new precompiler command ALLOCATE. For example, to allocate the SQL_CURSOR *emp_cursor* that was declared in the example above, you write the statement:

```
EXEC SQL ALLOCATE :emp_cursor;
```

Allocating a cursor does *not* require a call to the server, either at precompile time or at runtime. If the ALLOCATE statement contains an error (for example, an undeclared host variable), Pro*C/C++ issues a precompile time (PCC) error. Allocating a cursor variable *does* cause heap memory to be used. For this reason, you should normally avoid allocating a cursor variable in a program loop. Memory allocated for cursor variables is *not* freed when the cursor is closed, but only when the connection is closed.

Opening a Cursor Variable

You must open a cursor variable on the Oracle Server. You cannot use the embedded SQL OPEN command to open a cursor variable. You can open a cursor variable either by calling a PL/SQL stored procedure that opens the cursor (and defines it in the same statement). Or, you can open and define a cursor variable using an anonymous PL/SQL block in your Pro*C/C++ program.

For example, consider the following PL/SQL package, stored in the database:

```
CREATE PACKAGE demo_cur_pkg AS
    TYPE EmpName IS RECORD (name VARCHAR2(10));
    TYPE cur_type IS REF CURSOR RETURN EmpName;
    PROCEDURE open_emp_cur (
               curs     IN OUT cur_type,
               dept_num IN     NUMBER);
END;

CREATE PACKAGE BODY demo_cur_pkg AS
    CREATE PROCEDURE open_emp_cur (
               curs     IN OUT cur_type,
               dept_num IN     NUMBER) IS
    BEGIN
        OPEN curs FOR
            SELECT ename FROM emp
                WHERE deptno = dept_num
                ORDER BY ename ASC;
    END;
END;
```

After this package has been stored, you can open the cursor *curs* by calling the *open_emp_cur* stored procedure from your Pro*C/C++ program, and FETCH from the cursor in the program. For example:

```
...
sql_cursor    emp_cursor;
char          emp_name[11];
...
EXEC SQL ALLOCATE :emp_cursor;  /* allocate the cursor variable */
...
/* Open the cursor on the server side. */
EXEC SQL EXECUTE
    begin
        demo_cur_pkg.open_emp_cur(:emp_cursor, :dept_num);
    end;
END–EXEC;
EXEC SQL WHENEVER NOT FOUND DO break;
for (;;)
{
    EXEC SQL FETCH :emp_cursor INTO :emp_name;
    printf(”%s\n”, emp_name);
}
...
```

To open a cursor using a PL/SQL anonymous block in your Pro*C/C++ program, you define the cursor in the anonymous block. For example:

```
int dept_num = 10;
...
EXEC SQL EXECUTE
    BEGIN
        OPEN :emp_cursor FOR SELECT ename FROM emp
             WHERE deptno = :dept_num;
    END;
END–EXEC;
...
```

Opening in a Stand–Alone Stored Procedure

In the example above, a reference cursor was defined inside a package, and the cursor was opened in a procedure in that package. But it is not always necessary to define a reference cursor inside the package that contains the procedures that open the cursor.

If you need to open a cursor inside a stand–alone stored procedure, you can define the cursor in a separate package, and then reference that package in the stand–alone stored procedure that opens the cursor. Here is an example:

```
PACKAGE dummy IS
    TYPE EmpName IS RECORD (name VARCHAR2(10));
    TYPE emp_cursor_type IS REF CURSOR RETURN EmpName;
END;
–– and then define a stand–alone procedure:
PROCEDURE open_emp_curs (
      emp_cursor IN OUT dummy.emp_cursor_type;
      dept_num   IN     NUMBER) IS
    BEGIN
        OPEN emp_cursor FOR
            SELECT ename FROM emp WHERE deptno = dept_num;
    END;
END;
```

Return Types

When you define a reference cursor in a PL/SQL stored procedure, you must declare the type that the cursor returns. See the *PL/SQL User's Guide and Reference* for complete information on the reference cursor type and its return types.

Closing a Cursor Variable

Use the CLOSE command to close a cursor variable. For example, to close the *emp_cursor* cursor variable that was OPENed in the examples above, use the embedded SQL statement:

```
EXEC SQL CLOSE :emp_cursor;
```

Note that the cursor variable is a host variable, and so you must precede it with a colon.

You can re–use ALLOCATEd cursor variables. You can open, FETCH, and CLOSE as many times as needed for your application. However, if you disconnect from the server, then reconnect, you must re–ALLOCATE cursor variables.

> **Note:** Cursors are automatically deallocated by the SQLLIB runtime library upon exiting the current connection.

Using Cursor Variables with the OCI

You can share a Pro*C / C++ cursor variable with an OCI function. To do so, you must use the SQLLIB conversion functions, *sqlcda()* and *sqlcur()*. These functions convert between OCI cursor data areas and Pro*C / C++ cursor variables.

The *sqlcda()* function translates an allocated cursor variable to an OCI cursor data area. The syntax is:

```
void sqlcda(Cda_Def *cda, void *cur, int *retval);
```

where the parameters are

cda	A pointer to the destination OCI cursor data area.
cur	A pointer to the source Pro*C cursor variable.
retval	0 if no error, otherwise a SQLLIB (RTL) error number.

Note: In the case of an error, the *V2* and *rc* return code fields in the CDA also receive the error codes. The *rows processed count* field in the CDA is not set.

The *sqlcur()* function translates an OCI cursor data area to a Pro*C cursor variable. The syntax is:

```
void sqlcur(void *cur, Cda_Def *cda, int *retval);
```

where the parameters are

cur	A pointer to the destination Pro*C cursor variable.
cda	A pointer to the source OCI cursor data area.
retval	0 if no error, otherwise an error code.

Note: The SQLCA structure is not updated by this routine. The SQLCA components are only set after a database operation is performed using the translated cursor.

ANSI and K&R prototypes for these functions are provided in the *sqlapr.h* and *slqkpr.h* header files, respectively. Memory for both *cda* and *cur* must be allocated prior to calling these functions.

Restrictions

The following restrictions apply to the use of cursor variables:

- You cannot use cursor variables with dynamic SQL.
- You can only use cursor variables with the ALLOCATE, FETCH, and CLOSE commands

The DECLARE CURSOR command does not apply to cursor variables.

- You cannot FETCH from a CLOSEd cursor variable.
- You cannot FETCH from a non–ALLOCATEd cursor variable.
- If you precompile with MODE=ANSI, it is an error to close a cursor variable that is already closed.
- You cannot use the AT clause with the ALLOCATE command, nor with the FETCH and CLOSE commands if they reference a cursor variable.
- Cursor variables cannot be stored in columns in the database.
- A cursor variable itself cannot be declared in a package specification. Only the *type* of the cursor variable can be declared in the package specification.
- A cursor variable cannot be a component of a PL/SQL record.

A Sample Program

The following sample programs—a SQL script and a Pro*C program—demonstrate how you can use cursor variables in Pro*C. These sources are available on–line in your *demo* directory.

cv_demo.sql

```
-- PL/SQL source for a package that declares and
-- opens a ref cursor
CONNECT SCOTT/TIGER
CREATE OR REPLACE PACKAGE emp_demo_pkg as
   TYPE emp_cur_type IS REF CURSOR RETURN emp%ROWTYPE;
     PROCEDURE open_cur(curs IN OUT emp_cur_type, dno IN NUMBER);
END emp_demo_pkg;


CREATE OR REPLACE PACKAGE BODY emp_demo_pkg AS
    PROCEDURE open_cur(curs IN OUT emp_cur_type, dno IN NUMBER) IS
    BEGIN
        OPEN curs FOR SELECT *
            FROM emp WHERE deptno = dno
            ORDER BY ename ASC;
    END;
END emp_demo_pkg;
```

sample11.pc

```
/*
 *  Fetch from the EMP table, using a cursor variable.
 *  The cursor is opened in the stored PL/SQL procedure
 *  open_cur, in the EMP_DEMO_PKG package.
 *
 *  This package is available on–line in the file
 *  sample11.sql, in the demo directory.
 *
 */

#include <stdio.h>
#include <sqlca.h>

/* Error handling function. */
void sql_error();

main()
{
    char temp[32];

    EXEC SQL BEGIN DECLARE SECTION;
        char *uid = ”scott/tiger”;
        SQL_CURSOR emp_cursor;
        int dept_num;
        struct
        {
            int   emp_num;
            char  emp_name[11];
            char  job[10];
            int   manager;
            char  hire_date[10];
            float salary;
            float commission;
            int   dept_num;
        } emp_info;
        struct
        {
            short emp_num_ind;
            short emp_name_ind;
            short job_ind;
            short manager_ind;
            short hire_date_ind;
            short salary_ind;
            short commission_ind;
            short dept_num_ind;
        } emp_info_ind;
    EXEC SQL END DECLARE SECTION;

    EXEC SQL WHENEVER SQLERROR do sql_error(”Oracle error”);

/* Connect to Oracle. */
    EXEC SQL CONNECT :uid;

/* Allocate the cursor variable. */
    EXEC SQL ALLOCATE :emp_cursor;

/* Exit the inner for (;;) loop when NO DATA FOUND. */
    EXEC SQL WHENEVER NOT FOUND DO break;

    for (;;)
    {
        printf(”\nEnter department number  (0 to exit): ”);
        gets(temp);
        dept_num = atoi(temp);
        if (dept_num <= 0)
            break;

        EXEC SQL EXECUTE
            begin
                emp_demo_pkg.open_cur(:emp_cursor, :dept_num);
            end;
        END–EXEC;

        printf(”\nFor department %d––\n”, dept_num);
        printf(”ENAME\t             SAL\t             COMM\n”);
        printf(”–––––\t             –––\t             ––––\n”);
/* Fetch each row in the EMP table into the data struct.
   Note the use of a parallel indicator struct. */
        for (;;)
        {
             EXEC SQL FETCH :emp_cursor
                 INTO :emp_info INDICATOR :emp_info_ind;

             printf(”%s\t”, emp_info.emp_name);
             printf(”%8.2f\t\t”, emp_info.salary);
             if (emp_info_ind.commission_ind != 0)
                 printf(”    NULL\n”);
             else
                 printf(”%8.2f\n”, emp_info.commission);
        }

    }


/* Close the cursor. */
    EXEC SQL CLOSE :emp_cursor;
    exit(0);
}


void
sql_error(msg)
char *msg;
{
    long clen, fc;
    char cbuf[128];

    clen = (long) sizeof (cbuf);
    sqlgls(cbuf, &clen, &fc);

    printf(”\n%s\n”, msg);
    printf(”Statement is––\n%s\n”, cbuf);
    printf(”Function code is %ld\n\n”, fc);

    sqlglm(cbuf, (int *) &clen, (int *) &clen);
    printf (”\n%.*s\n”, clen, cbuf);

    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK WORK;
    exit(–1);
}
```

Connecting to Oracle

Your Pro*C/C++ program must connect to Oracle before querying or manipulating data. To log on, simply use the CONNECT statement

```
EXEC SQL CONNECT :username IDENTIFIED BY :password;
```

where *username* and *password* are **char** or VARCHAR host variables.

Or, you can use the statement

```
EXEC SQL CONNECT :usr_pwd;
```

where the host variable *usr_pwd* contains your username and password separated by a slash character (/).

The CONNECT statement must be the first SQL statement executed by the program. That is, other SQL statements can physically but not logically precede the CONNECT statement in the precompilation unit.

To supply the Oracle username and password separately, you define two host variables as character strings or VARCHARs. (If you supply a username containing both username and password, only one host variable is needed.)

Make sure to set the username and password variables before the CONNECT is executed, or it will fail. Your program can prompt for the values, or you can hardcode them as follows:

```
char *username = ”SCOTT”;
char *password = ”TIGER”;
...
EXEC SQL WHENEVER SQLERROR ...
EXEC SQL CONNECT :username IDENTIFIED BY :password;
```

However, you cannot hardcode a username and password into the CONNECT statement. Nor can you use quoted literals. For example, both of the following statements are *invalid*:

```
EXEC SQL CONNECT SCOTT IDENTIFIED BY TIGER;
EXEC SQL CONNECT ’SCOTT’ IDENTIFIED BY ’TIGER’;
```

Connecting Using SQL*Net Version 2

Some of the examples in this and the following section might use SQL*Net V1 connection strings to show network protocols and server names. To connect using a SQL*Net V2 driver, substitute a service name, as defined in your *tnsnames.ora* configuration file or in Oracle Names, in place of the SQL*Net V1 connect string.

If you are using Oracle Names, the name server obtains the service name from the network definition database.

See *Understanding SQL*Net* for more information about SQL*Net V2.

Automatic Connects

You can automatically connect to Oracle with the username

```
OPS$username
```

where *username* is the current operating system username, and OPS$*username* is a valid Oracle database username. (The actual value for OPS$ is defined in the INIT.ORA parameter file.) You simply pass to the Pro*C Precompiler a slash character, as follows:

```
...
char  oracleid = ’/’;
...
EXEC SQL CONNECT :oracleid;
```

This automatically connects you as user OPS$*username*. For example, if your operating system username is RHILL, and OPS$RHILL is a valid Oracle username, connecting with ’/’ automatically logs you on to Oracle as user OPS$RHILL.

You can also pass a ’/’ in a string to the precompiler. However, the string cannot contain trailing blanks. For example, the following CONNECT statement will fail:

```
...
char  oracleid[10] = ”/    ”;
...
EXEC SQL CONNECT :oracleid;
```

The AUTO_CONNECT Precompiler Option

If AUTO_CONNECT=YES, and the application is not already connected to a database when it processes the first executable SQL statement, it attempts to connect using the userid

```
OPS$<username>
```

where *username* is your current operating system user or task name and OPS$username is a valid Oracle userid.

When AUTO_CONNECT=NO, you must use the CONNECT statement in your program to connect to Oracle.

Concurrent Connections

The Pro*C/C++ Precompiler supports distributed processing via SQL*Net. Your application can concurrently access any combination of local and remote databases or make multiple connections to the same database. In Figure 3 – 1, an application program communicates with one local and three remote Oracle databases. ORA2, ORA3, and ORA4 are simply logical names used in CONNECT statements.

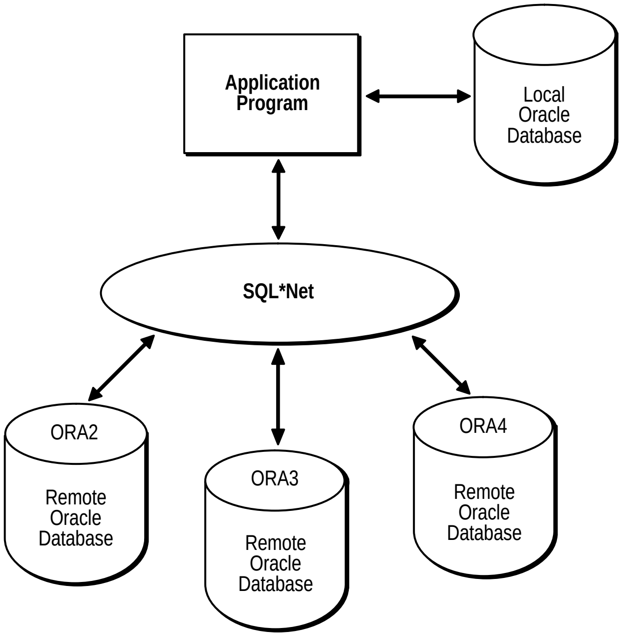


Figure 3 – 1 Connecting via SQL*Net

By eliminating the boundaries in a network between different machines and operating systems, SQL*Net provides a distributed processing environment for Oracle tools. This section shows you how Pro*C/C++ supports distributed processing via SQL*Net. You learn how your application can

- directly or indirectly access other databases
- concurrently access any combination of local and remote databases
- make multiple connections to the same database

For details on installing SQL*Net and identifying available databases, refer to the *SQL*Net User's Guide* and your system–specific Oracle documentation.

Some Preliminaries

The communicating points in a network are called *nodes*. SQL*Net lets you transmit information (SQL statements, data, and status codes) over the network from one node to another.

A *protocol* is a set of rules for accessing a network. The rules establish such things as procedures for recovering after a failure and formats for transmitting data and checking errors.

The SQL*Net syntax for connecting to the default database in the local domain is simply to use the service name for the database.

If the service name is not in the default (local) domain, you must use a global specification (all domains specified). For example:

```
HR.US.ORACLE.COM
```

Default Databases and Connections

Each node has a *default* database. If you specify a database name, but no domain, in your CONNECT statement, you connect to the default database on the named local or remote node.

A *default* connection is made by a CONNECT statement that has no AT clause. The connection can be to any default or non–default database at any local or remote node. SQL statements without an AT clause are executed against the default connection. Conversely, a *non–default* connection is made by a CONNECT statement that has an AT clause. SQL statements with an AT clause are executed against the non–default connection.

All database names must be unique, but two or more database names can specify the same connection. That is, you can have multiple connections to any database on any node.

Explicit Connections

Usually, you establish a connection to Oracle as follows:

```
EXEC SQL CONNECT :username IDENTIFIED BY :password;
```

You can also use

```
EXEC SQL CONNECT :usr_pwd;
```

where *usr_pwd* contains *username/password*.

You can automatically connect to Oracle with the userid

```
OPS$username
```

where *username* is your current operating system user or task name and OPS$*username* is a valid Oracle userid. You simply pass to the precompiler a slash (/) character, as follows:

```
char oracleid = '/';
...
EXEC SQL CONNECT :oracleid;
```

This automatically connects you as user OPS$*username*.

If you do not specify a database and node, you are connected to the default database at the current node. If you want to connect to a different database, you must explicitly identify that database.

With *explicit connections*, you connect to another database directly, giving the connection a name that will be referenced in SQL statements. You can connect to several databases at the same time and to the same database multiple times.

Single Explicit Connections

In the following example, you connect to a single non–default database at a remote node:

```
/* declare needed host variables */
char  username[10]  = ”scott”;
char  password[10]  = ”tiger”;
char  db_string[20] = ”NYNON”;

/* give the database connection a unique name */
EXEC SQL DECLARE DB_NAME DATABASE;

/* connect to the non–default database  */
EXEC SQL CONNECT :username IDENTIFIED BY :password
   AT DB_NAME USING :db_string;
```

The identifiers in this example serve the following purposes:

- The host variables *username* and *password* identify a valid user.
- The host variable *db_string* contains the SQL*Net syntax for connecting to a non–default database at a remote node.
- The undeclared identifier DB_NAME names a non–default connection; it is an identifier used by Oracle, *not* a host or program variable.

The USING clause specifies the network, machine, and database to be associated with DB_NAME. Later, SQL statements using the AT clause (with DB_NAME) are executed at the database specified by *db_string*.

Alternatively, you can use a character host variable in the AT clause, as the following example shows:

```
/* declare needed host variables */
char  username[10]  = ”scott”;
char  password[10]  = ”tiger”;
char  db_name[10]   = ”oracle1”;
char  db_string[20] = ”NYNON”;

/* connect to the non–default database using db_name */
EXEC SQL CONNECT :username IDENTIFIED BY :password
   AT :db_name USING :db_string;
...
```

If *db_name* is a host variable, the DECLARE DATABASE statement is not needed. Only if DB_NAME is an undeclared identifier must you execute a DECLARE DB_NAME DATABASE statement before executing a CONNECT ... AT DB_NAME statement.

SQL Operations If granted the privilege, you can execute any SQL data manipulation statement at the non–default connection. For example, you might execute the following sequence of statements:

```
EXEC SQL AT DB_NAME SELECT ...
EXEC SQL AT DB_NAME INSERT ...
EXEC SQL AT DB_NAME UPDATE ...
```

In the next example, *db_name* is a host variable:

```
EXEC SQL AT :db_name DELETE ...
```

If *db_name* is a host variable, all database tables referenced by the SQL statement must be defined in DECLARE TABLE statements. Otherwise, the precompiler issues a warning.

PL/SQL Blocks You can execute a PL / SQL block using the AT clause. The following example shows the syntax:

```
EXEC SQL AT :db_name EXECUTE
    begin
        /* PL/SQL block here */
    end;
END–EXEC;
```

Cursor Control Cursor control statements such as OPEN, FETCH, and CLOSE are exceptions—they never use an AT clause. If you want to associate a cursor with an explicitly identified database, use the AT clause in the DECLARE CURSOR statement, as follows:

```
EXEC SQL AT :db_name DECLARE emp_cursor CURSOR FOR ...
EXEC SQL OPEN emp_cursor ...
EXEC SQL FETCH emp_cursor ...
EXEC SQL CLOSE emp_cursor;
```

If *db_name* is a host variable, its declaration must be within the scope of all SQL statements that refer to the DECLAREd cursor. For example, if you OPEN the cursor in one subprogram, then FETCH from it in another subprogram, you must declare *db_name* globally.

When OPENing, CLOSing, or FETCHing from the cursor, you do not use the AT clause. The SQL statements are executed at the database named in the AT clause of the DECLARE CURSOR statement or at the default database if no AT clause is used in the cursor declaration.

The AT :*host_variable* clause allows you to change the connection associated with a cursor. However, you cannot change the association while the cursor is open. Consider the following example:

```
EXEC SQL AT :db_name DECLARE emp_cursor CURSOR FOR ...
strcpy(db_name, ”oracle1”);
EXEC SQL OPEN emp_cursor;
EXEC SQL FETCH emp_cursor INTO ...
strcpy(db_name, ”oracle2”);
EXEC SQL OPEN emp_cursor;   /*  illegal, cursor still open */
EXEC SQL FETCH emp_cursor INTO ...
```

This is illegal because *emp_cursor* is still open when you try to execute the second OPEN statement. Separate cursors are not maintained for different connections; there is only one *emp_cursor*, which must be closed before it can be reopened for another connection. To debug the last example, simply close the cursor before reopening it, as follows:

```
...
EXEC SQL CLOSE emp_cursor;  –– close cursor first
strcpy(db_name, ”oracle2”);
EXEC SQL OPEN emp_cursor;
EXEC SQL FETCH emp_cursor INTO ...
```

Dynamic SQL Dynamic SQL statements are similar to cursor control statements in that some never use the AT clause.

For dynamic SQL Method 1, you must use the AT clause if you want to execute the statement at a non–default connection. An example follows:

```
EXEC SQL AT :db_name EXECUTE IMMEDIATE :slq_stmt;
```

For Methods 2, 3, and 4, you use the AT clause only in the DECLARE STATEMENT statement if you want to execute the statement at a non–default connection. All other dynamic SQL statements such as PREPARE, DESCRIBE, OPEN, FETCH, and CLOSE never use the AT clause. The next example shows Method 2:

```
EXEC SQL AT :db_name DECLARE slq_stmt STATEMENT;
EXEC SQL PREPARE slq_stmt FROM :sql_string;
EXEC SQL EXECUTE slq_stmt;
```

The following example shows Method 3:

```
EXEC SQL AT :db_name DECLARE slq_stmt STATEMENT;
EXEC SQL PREPARE slq_stmt FROM :sql_string;
EXEC SQL DECLARE emp_cursor CURSOR FOR slq_stmt;
EXEC SQL OPEN emp_cursor ...
EXEC SQL FETCH emp_cursor INTO ...
EXEC SQL CLOSE emp_cursor;
```

Multiple Explicit Connections

You can use the AT *db_name* clause for multiple explicit connections, just as you would for a single explicit connection. In the following example, you connect to two non–default databases concurrently:

```
/* declare needed host variables */
char  username[10]   = ”scott”;
char  password[10]   = ”tiger”;
char  db_string1[20] = ”NYNON1”;
char  db_string2[20] = ”CHINON”;
...
/* give each database connection a unique name */
EXEC SQL DECLARE DB_NAME1 DATABASE;
EXEC SQL DECLARE DB_NAME2 DATABASE;
/* connect to the two non–default databases */
EXEC SQL CONNECT :username IDENTIFIED BY :password
   AT DB_NAME1 USING :db_string1;
EXEC SQL CONNECT :username IDENTIFIED BY :password
   AT DB_NAME2 USING :db_string2;
```

The undeclared identifiers DB_NAME1 and DB_NAME2 are used to name the default databases at the two non–default nodes so that later SQL statements can refer to the databases by name.

Alternatively, you can use a host variable in the AT clause, as the following example shows:

```
/* declare needed host variables */
char  username[10]   = ”scott”;
char  password[10]   = ”tiger”;
char  db_name[20];
char  db_string[20];
int   n_defs = 3;    /* number of connections to make */
...
for (i = 0; i < n_defs; i++)
{
    /* get next database name and SQL*Net string */
    printf(”Database name: ”);
    gets(db_name);
    printf(”SQL*Net string: ”);
    gets(db_string);
    /* do the connect */
    EXEC SQL CONNECT :username IDENTIFIED BY :password
        AT :db_name USING :db_string;
}
```

You can also use this method to make multiple connections to the same database, as the following example shows:

```
strcpy(db_string, ”NYNON”);
for (i = 0; i < ndefs; i++)
{
    /* connect to the non–default database */
    printf(”Database name: ”);
    gets(db_name);
    EXEC SQL CONNECT :username IDENTIFIED BY :password
       AT :db_name USING :db_string;
}
...
```

You must use different database names for the connections, even though they use the same SQL*Net string. However, you can connect twice to the same database using just one database name because that name identifies the default and non–default databases.

Ensuring Data Integrity

Your application program must ensure the integrity of transactions that manipulate data at two or more remote databases. That is, the program must commit or roll back *all* SQL statements in the transactions. This might be impossible if the network fails or one of the systems crashes.

For example, suppose you are working with two accounting databases. You debit an account on one database and credit an account on the other database, then issue a COMMIT at each database. It is up to your program to ensure that both transactions are committed or rolled back.

Implicit Connections

Implicit connections are supported through the Oracle distributed query facility, which does not require explicit connections, but only supports the SELECT statement. A distributed query allows a single SELECT statement to access data on one or more non–default databases.

The distributed query facility depends on database links, which assign a name to a CONNECT *statement* rather than to the connection itself.   At run time, the embedded SELECT statement is executed by the specified Oracle Server, which *implicitly* connects to the non–default database(s) to get the required data.

Single Implicit Connections

In the next example, you connect to a single non–default database. First, your program executes the following statement to define a database link (database links are usually established interactively by the DBA or user):

```
EXEC SQL CREATE DATABASE LINK db_link
     CONNECT TO username IDENTIFIED BY password
         USING ’NYNON’;
```

Then, the program can query the non–default EMP table using the database link, as follows:

```
EXEC SQL SELECT ENAME, JOB INTO :emp_name, :job_title
     FROM emp@db_link
     WHERE DEPTNO = :dept_number;
```

The database link is not related to the database name used in the AT clause of an embedded SQL statement. It simply tells Oracle where the non–default database is located, the path to it, and what Oracle username and password to use. The database link is stored in the data dictionary until it is explicitly dropped.

In our example, the default Oracle Server logs on to the non–default database via SQL*Net using the database link *db_link*. The query is submitted to the default Server, but is “forwarded” to the non–default database for execution.

To make referencing the database link easier, you can create a synonym as follows (again, this is usually done interactively):

```
EXEC SQL CREATE SYNONYM emp FOR emp@db_link;
```

Then, your program can query the non–default EMP table, as follows:

```
EXEC SQL SELECT ENAME, JOB INTO :emp_name, :job_title
     FROM emp
     WHERE DEPTNO = :dept_number;
```

This provides location transparency for *emp*.

Multiple Implicit Connections

In the following example, you connect to two non–default databases concurrently. First, you execute the following sequence of statements to define two database links and create two synonyms:

```
EXEC SQL CREATE DATABASE LINK db_link1
     CONNECT TO username1 IDENTIFIED BY password1
         USING ’NYNON’;
EXEC SQL CREATE DATABASE LINK db_link2
     CONNECT TO username2 IDENTIFIED BY password2
         USING ’CHINON’;
EXEC SQL CREATE SYNONYM emp FOR emp@db_link1;
EXEC SQL CREATE SYNONYM dept FOR dept@db_link2;
```

Then, your program can query the non–default EMP and DEPT tables, as follows:

```
EXEC SQL SELECT ENAME, JOB, SAL, LOC
     FROM emp, dept
     WHERE emp.DEPTNO = dept.DEPTNO AND DEPTNO = :dept_number;
```

Oracle executes the query by performing a join between the non–default EMP table at *db_link1* and the non–default DEPT table at *db_link2*.

Embedding Oracle Call Interface (OCI) Calls

To embed OCI calls in your Pro*C/C++ program, take the following steps:

1. Declare an OCI Logon Data Area (LDA) in your Pro*C/C++ program (outside the Declare Section if you precompile with MODE=ANSI). The LDA is a structure defined in the OCI header file *ocidfn.h*. For details, see the *Programmer's Guide to the Oracle Call Interface.*
2. Connect to Oracle using the embedded SQL statement CONNECT, not the OCI *orlon*() or *onblon()* calls.
3. Call the Oracle runtime library function *sqllda*() to set up the LDA.

That way, the Pro*C Precompiler and the OCI "know" that they are working together. However, there is no sharing of Oracle cursors.

You need not worry about declaring the OCI Host Data Area (HDA) because the Oracle runtime library manages connections and maintains the HDA for you.

Setting Up the LDA

You set up the LDA by issuing the OCI call

```
sqllda(&lda);
```

where *lda* identifies the LDA data structure.

If the setup fails, the *lda_rc* field in the *lda* is set to 1012 to indicate the error.

Remote and Multiple Connections

A call to *sqllda()* sets up an LDA for the connection used by the most recently executed SQL statement. To set up the different LDAs needed for additional connections, just call *sqllda()* with a different *lda* after each CONNECT. In the following example, you connect to two non–default databases concurrently:

```
#include <ocidfn.h>
Lda_Def lda1;
Lda_Def lda2;

char username[10], password[10], db_string1[20], dbstring2[20];
...
strcpy(username, ”scott”);
strcpy(password, ”tiger”);
strcpy(db_string1, ”NYNON”);
strcpy(db_string2, ”CHINON”);
/* give each database connection a unique name */
EXEC SQL DECLARE DB_NAME1 DATABASE;
EXEC SQL DECLARE DB_NAME2 DATABASE;
```

```
/* connect to first non–default database */
EXEC SQL CONNECT :username IDENTIFIED BY :password;
    AT DB_NAME1 USING :db_string1;
/* set up first LDA */
sqllda(&lda1);
/* connect to second non–default database */
EXEC SQL CONNECT :username IDENTIFIED BY :password;
    AT DB_NAME2 USING :db_string2;
/* set up second LDA */
sqllda(&lda2);
```

DB_NAME1 and DB_NAME2 are *not* C variables; they are SQL identifiers. You use them only to name the default databases at the two non–default nodes, so that later SQL statements can refer to the databases by name.

Developing Multi–threaded Applications

Multi–threaded applications have multiple *threads* executing in a shared address space. Threads are "lightweight" processes that execute within a process; they share data segments but have their own program counter, machine registers and stack. For more detailed information about multi–threaded applications, refer to your thread–package specific reference material.

The Pro*C/C++ Precompiler supports development of multi–threaded Oracle7 Server applications (on platforms that support multi–threaded applications) using the following:

- a command–line option to generate thread–safe code
- embedded SQL statements and directives to support multi–threading
- thread–safe SQLLIB and other client–side Oracle libraries

☞ **Attention:** If your development platform does not support multi–threading, the information in this section does not apply.

Note: While your platform may support a particular thread package, see your platform–specific Oracle documentation to determine whether Oracle supports it.

The following topics discuss how to use the preceding features to develop multi–threaded Pro*C/C++ applications:

- runtime contexts for multi–threaded applications
- two models for using runtime contexts
- user–interface features for multi–threaded applications
- programming considerations for writing multi–threaded applications with Pro*C/C++
- a sample multi–threaded Pro*C/C++ application

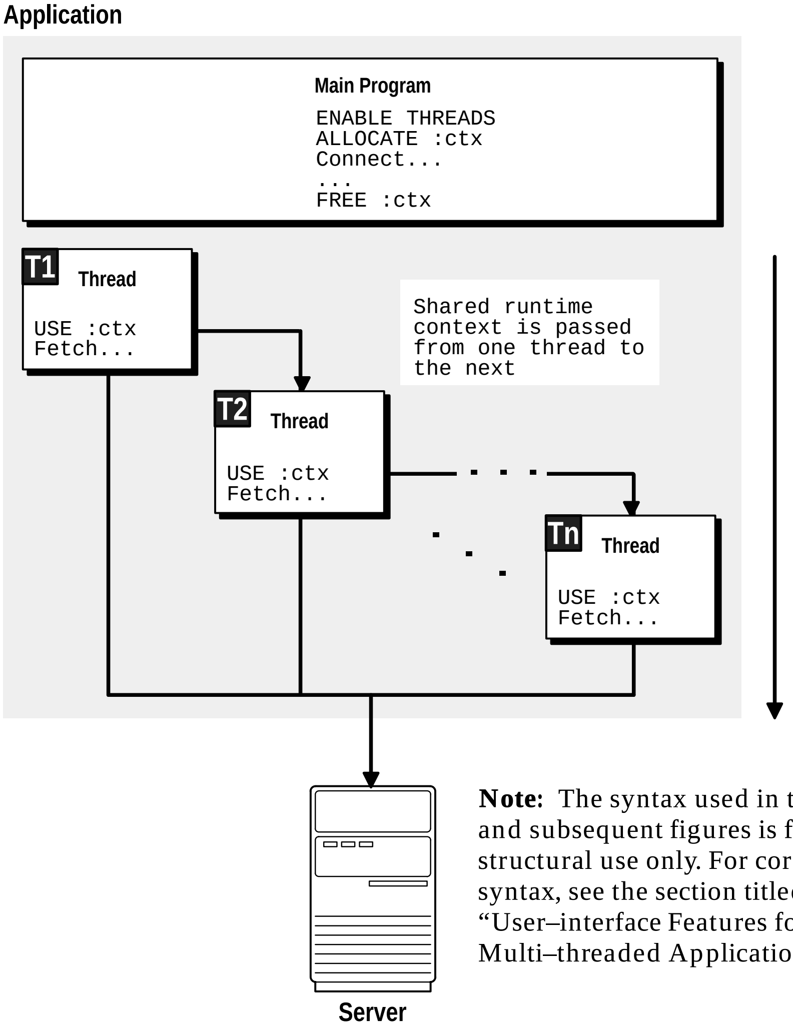
Runtime Contexts in Pro*C/C++

To loosely couple a thread and a connection, the Oracle Pro*C/C++ Precompiler introduces the notion of a *runtime context*. The runtime context includes the following resources and their current states:

- zero or more connections to one or more Oracle7 Servers
- zero or more cursors used for the server connections
- inline options, such as MODE, HOLD_CURSOR, RELEASE_CURSOR, and SELECT_ERROR

Rather than simply supporting a loose coupling between threads and connections, the Pro*C/C++ Precompiler allows developers to loosely couple threads with runtime contexts. Pro*C/C++ allows an application to define a handle to a runtime context, and pass that handle from one thread to another.

For example, an interactive application spawns a thread, T1, to execute a query and return the first 10 rows to the application. T1 then terminates. After obtaining the necessary user input, another thread, T2, is spawned (or an existing thread is used) and the runtime context for T1 is passed to T2 so it can fetch the next 10 rows by processing the same cursor.



Note: The syntax used in this and subsequent figures is for structural use only. For correct syntax, see the section titled, "User–interface Features for Multi–threaded Applications."

Figure 3 – 2 Loosely Coupling Connections and Threads

Runtime Context Usage Models

Two possible models for using runtime contexts in multi–threaded Pro*C / C++ applications are shown here:

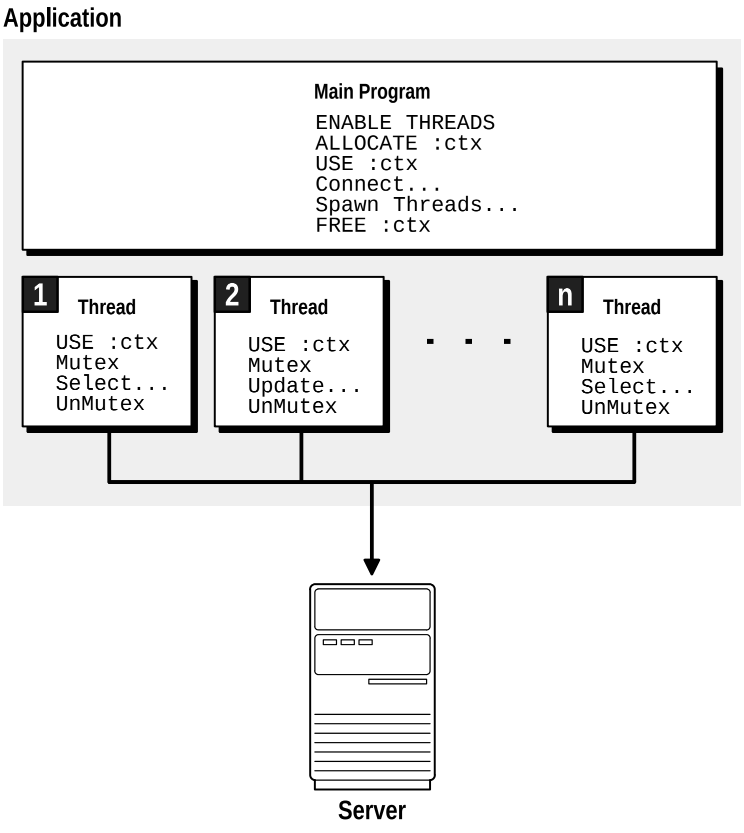
- multiple threads sharing a single runtime context
- multiple threads using separate runtime contexts

Regardless of the model you use for runtime contexts, you *cannot* share a runtime context between multiple threads *at the same time*. If two or more threads attempt to use the same runtime context simultaneously, the following runtime error occurs:

```
SQL–02131:  Runtime context in use
```

Multiple Threads Sharing a Single Runtime Context

Figure 3 – 3 shows an application running in a multi–threaded environment. The various threads share a single runtime context to process one or more SQL statements. Again, runtime contexts cannot be shared by multiple threads at the same time; the mutexes in Figure 3 – 3 show how to prevent concurrent usage.



```
ENABLE THREADS
ALLOCATE :ctx
USE :ctx
Connect...
Spawn Threads...
FREE :ctx
```

```
USE :ctx
Mutex
Select...
UnMutex
```

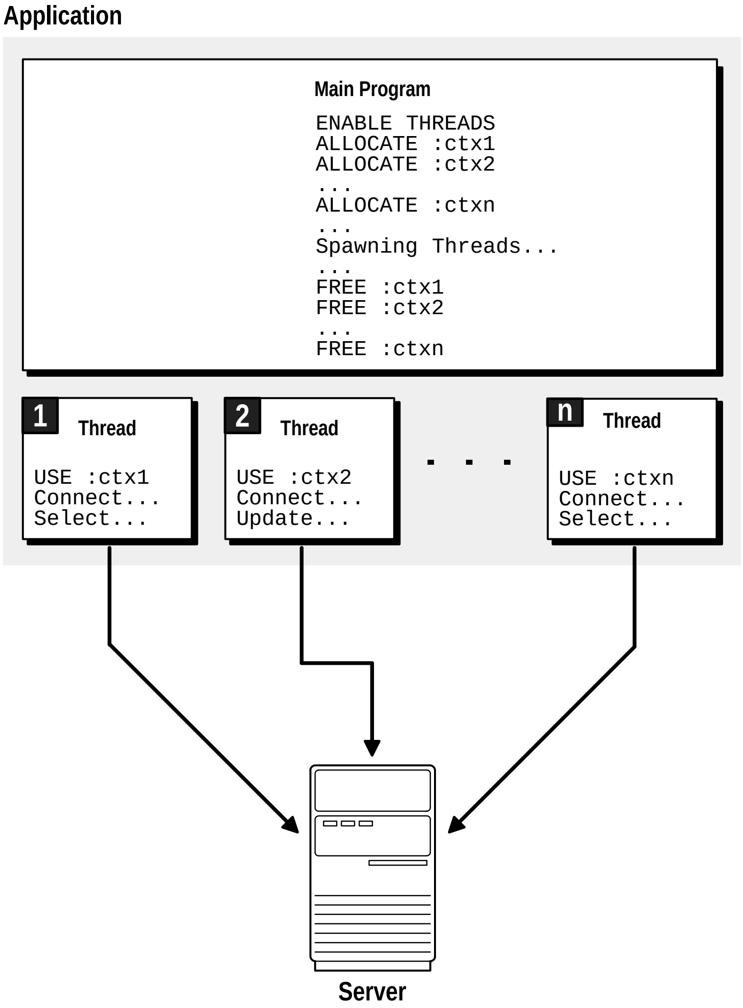
```
USE :ctx
Mutex
Update...
UnMutex
```

```
USE :ctx
Mutex
Select...
UnMutex
```

Figure 3 – 3 Context Sharing Among Threads

Multiple Threads Sharing Multiple Runtime Contexts

Figure 3 – 4 shows an application that executes multiple threads using multiple runtime contexts. In this situation, the application does not require mutexes, because each thread has a dedicated runtime context.



```
ENABLE THREADS
ALLOCATE :ctx1
ALLOCATE :ctx2
...
ALLOCATE :ctxn
...
Spawning Threads...
...
FREE :ctx1
FREE :ctx2
...
FREE :ctxn
```

```
USE :ctx1
Connect...
Select...
```

```
USE :ctx2
Connect...
Update...
```

```
USE :ctxn
Connect...
Select...
```

Figure 3 – 4 No Context Sharing Among Threads

User–interface Features for Multi–threaded Applications

The Pro*C / C++ Precompiler provides the following user–interface features to support multi–threaded applications:

- command–line option, THREADS=YES | NO
- embedded SQL statements and directives
- thread–safe SQLLIB public functions

THREADS Option

With THREADS=YES specified on the command line, the Pro*C / C++ Precompiler ensures that the generated code is thread–safe, given that you follow the guidelines described in "Programming Considerations" on page 3 – 108. With THREADS=YES specified, Pro*C / C++ verifies that all SQL statements execute within the scope of a user–defined runtime context. If your program does not meet this requirement, the following precompiler error is returned:

```
PCC–02390:  No EXEC SQL CONTEXT USE statement encountered
```

For more information about the THREADS option, see "THREADS" on page 7 – 37.

Embedded SQL Statements and Directives

The following embedded SQL statements and directives support the definition and usage of runtime contexts and threads:

- EXEC SQL ENABLE THREADS;
- EXEC SQL CONTEXT ALLOCATE :*context_var*;
- EXEC SQL CONTEXT USE :*context_var*;
- EXEC SQL CONTEXT FREE :*context_var*;

For these EXEC SQL statements, *context_var* is the handle to the runtime context and must be declared of type **sql_context** as follows:

```
sql_context <context_variable>;
```

EXEC SQL ENABLE THREADS
This executable SQL statement initializes a process that supports multiple threads. This must be the first executable SQL statement in your multi–threaded application. For more detailed information, see "ENABLE THREADS" on page F – 29.

EXEC SQL CONTEXT ALLOCATE
This executable SQL statement initializes and allocates memory for the specified runtime context; the runtime–context variable must be declared of type sql_context. For more detailed information, see "CONTEXT ALLOCATE" on page F – 14.

EXEC SQL CONTEXT USE

This directive instructs the precompiler to use the specified runtime context for subsequent executable SQL statements. The runtime context specified must be previously allocated using an EXEC SQL CONTEXT ALLOCATE statement.

The EXEC SQL CONTEXT USE directive works similarly to the EXEC SQL WHENEVER directive in that it affects all executable SQL statements which positionally follow it in a given source file without regard to standard C scope rules. In the following example, the UPDATE statement in *function2()* uses the global runtime context, *ctx1*:

```
sql_context ctx1;            /* declare global context ctx1     */

function1()
{  sql_context ctx1;         /* declare local context ctx1      */
   EXEC SQL CONTEXT USE :ctx1;
   EXEC SQL INSERT INTO ...  /* local ctx1 used for this stmt   */
   ...
}

function2()
{  EXEC SQL UPDATE ...       /* global ctx1 used for this stmt */
}
```

In the next example, there is no global runtime context. The precompiler refers to the *ctx1* runtime context in the generated code for the UPDATE statement. However, there is no context variable in scope for *function2()*, so errors are generated at compile time.

```
function1()
{  sql_context ctx1;         /* local context variable declared */
   EXEC SQL CONTEXT USE :ctx1;
   EXEC SQL INSERT INTO ...     /* ctx1 used for this statement */
   ...
}
function2()
{  EXEC SQL UPDATE ...   /* Error! No context variable in scope */
}
```

For more detailed information, see “CONTEXT USE” on page F – 16.

EXEC SQL CONTEXT FREE

This executable SQL statement frees the memory associated with the specified runtime context and places a null pointer in the host program variable. For more detailed information, see “CONTEXT FREE” on page F – 15.

Examples

The following code fragments show how to use embedded SQL statements and precompiler directives for two typical programming models; they use *thread_create()* to create threads.

The first example showing multiple threads using multiple runtime contexts:

```
main()
{  sql_context ctx1,ctx2;           /* declare runtime contexts */
   EXEC SQL ENABLE THREADS;
   EXEC SQL ALLOCATE :ctx1;
   EXEC SQL ALLOCATE :ctx2;
   ...
/* spawn thread, execute function1 (in the thread) passing ctx1 */
   thread_create(..., function1, ctx1);
/* spawn thread, execute function2 (in the thread) passing ctx2 */
   thread_create(..., function2, ctx2);
   ...
   EXEC SQL CONTEXT FREE :ctx1;
   EXEC SQL CONTEXT FREE :ctx2;
   ...
}

void function1(sql_context ctx)
{  EXEC SQL CONTEXT USE :ctx;
/* execute executable SQL statements on runtime context ctx1!!! */
   ...
}

void function2(sql_context ctx)
{  EXEC SQL CONTEXT USE :ctx;
/* execute executable SQL statements on runtime context ctx2!!! */
   ...
}
```

The next example shows how to use multiple threads that share a common runtime context. Because the SQL statements executed in *function1()* and *function2()* potentially execute at the same time, you must place mutexes around every *executable* EXEC SQL statement to ensure serial, therefore safe, manipulation of the data.

```
main()
{  sql_context ctx;                  /* declare runtime context */
   EXEC SQL ALLOCATE :ctx;
   ...
/* spawn thread, execute function1 (in the thread) passing ctx  */
   thread_create(..., function1, ctx);
/* spawn thread, execute function2 (in the thread) passing ctx  */
   thread_create(..., function2, ctx);
   ...
}

void function1(sql_context ctx)
{  EXEC SQL CONTEXT USE :ctx;
/* Execute SQL statements on runtime context ctx.               */
   ...
}

void function2(sql_context ctx)
{  EXEC SQL CONTEXT USE :ctx;
/* Execute SQL statements on runtime context ctx.               */
   ...
}
```

Thread–safe SQLLIB Public Functions

Each standard SQLLIB public function is accompanied by a thread–safe counterpart, suffixed with a *t*, that accepts the runtime context as an additional argument at the beginning of the argument list. For example, the syntax for *sqlglm()* is:

```
void sqlglm(char   *message_buffer,
            size_t *buffer_size,
            size_t *message_length);
```

while the thread–safe version, *sqlglmt()*, is:

```
void sqlglmt(void    *context
             char    *message_buffer,
             size_t  *buffer_size,
             size_t  *message_length);
```

You *must* use the thread–safe versions of the SQLLIB functions when writing multi–threaded applications.

Table 3 – 8 is a list of all the thread–safe SQLLIB public functions and their corresponding syntax. Cross–references to the related non–thread functions are provided to help you find more complete descriptions.

Thread–safe Function Calls	Cross–reference

struct SQLDA *sqlaldt(void   *context,
                      int   maximum_variables
                      int   maximum_name_length
                      int   maximum_ind_name_length);
	see also sqlald() on page 12 – 4

void sqlcdat(void   *context,
             Cda_Def *cda,
             void   *cursor,
             int   *return_value);
	see also sqlcda() on page 3 – 81

void sqlclut(void *context, struct SQLDA *descriptor_name);	*see also sqlclu()* on page 12 – 32

void sqlcurt(void   *context,
             void   *cursor,
             Cda_Def *cda,
             int   *return_value)
	see also sqlcur() on page 3 – 81

void sqlglmt(void   *context,
             char   *message_buffer,
             size_t *buffer_size,
             size_t *message_length);
	see also sqlglm() on page 9 – 22

void sqlglst(void   *context,
             char   *statement_buffer,
             size_t *statement_length,
             size_t *sqlfc);
	see also sqlgls() on page 9 – 29

void sqlld2t(void   *context,
             Lda_Def *lda,
             text   *cname,
             sb4   *cname_length);
	see also sqlld2() on page 3 – 115

void sqlldat(void *context, Lda_Def *lda);	*see also sqllda()* on page 3 – 97

void sqlnult(void   *context,
             unsigned short *value_type,
             unsigned short *type_code,
             int   *null_status);
	see also sqlnul() on page 12 – 15

void sqlprct(void   *context,
             long   *length,
             int   *precision,
             int   *scale);
	see also sqlprc() on page 12 – 13

void sqlpr2t(void   *context,
             long   *length,
             int   *precision,
             int   *scale);
	see also sqlpr2() on page 12 – 15

void sqlvcpt(void   *context,
             size_t *data_length,
             size_t *total_length);
	see also sqlvcp() on page 3 – 37

Table 3 – 8 Thread–safe SQLLIB Public Functions

Attention: For the specific datatypes used in the argument lists for these functions, refer to your platform–specific *sqlcpr.h* file.

Programming Considerations

While Oracle ensures that the SQLLIB code is thread–safe, you are responsible for ensuring that your Pro*C / C++ source code is designed to work properly with threads; for example, carefully consider your use of static and global variables.

In addition, multi–threaded requires design decisions regarding the following:

- declaring the SQLCA as a thread–safe **struct**, typically an auto variable and one for each runtime context
- declaring the SQLDA as a thread–safe **struct**, like the SQLCA, typically an auto variable and one for each runtime context
- declaring host variables in a thread–safe fashion, in other words, carefully consider your use of static and global host variables.
- avoiding simultaneous use of a runtime context in multiple threads
- whether or not to use default database connections or to explicitly define them using the AT clause

Also, no more than one executable embedded SQL statement, for example, EXEC SQL UPDATE, may be outstanding on a runtime context at a given time.

Existing requirements for precompiled applications also apply. For example, all references to a given cursor must appear in the same source file.

Example

The following program is one approach to writing a multi–threaded embedded SQL application; it is written specifically for the DCE thread package and is, therefore, not portable. The program creates as many sessions as there are threads. Each thread executes zero or more transactions, that are specified in a transient structure called ”records.”

```
/*
 * Requirements:
 *             The program requires a table ”ACCOUNTS” to be in
 *             the schema SCOTT/TIGER. The description of
 *             ACCOUNTS is:
 *  SQL> desc accounts
 *  Name                            Null?    Type
 *  ------------------------------- -------  ------------
 *  ACCOUNT                                  NUMBER(10)
 *  BALANCE                                  NUMBER(12,2)
 *
 *  For proper execution, the table should be filled with the
 *  accounts 10001 to 1008.
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#ifndef ORA_PROC
#include <pthread.h>
#endif
#include <sqlca.h>
```

```
/* Function prototypes                                          */
void err_report();
void do_transaction();
void get_transaction();
void logon();
void logoff();

#define CONNINFO ”SCOTT/TIGER”
#define THREADS  3

struct parameters
{  sql_context * ctx;
   int thread_id;
};
typedef struct parameters parameters;

struct record_log
{  char action;
   unsigned int from_account;
   unsigned int to_account;
   float  amount;
};
typedef struct record_log record_log;

record_log records[]= { { ’M’, 10001, 10002, 12.50 },
                        { ’M’, 10001, 10003, 25.00 },
                        { ’M’, 10001, 10003, 123.00 },
                        { ’M’, 10001, 10003, 125.00 },
                        { ’M’, 10002, 10006, 12.23 },
                        { ’M’, 10007, 10008, 225.23 },
                        { ’M’, 10002, 10008, 0.70 },
                        { ’M’, 10001, 10003, 11.30 },
                        { ’M’, 10003, 10002, 47.50 },
                        { ’M’, 10002, 10006, 125.00 },
                        { ’M’, 10007, 10008, 225.00 },
                        { ’M’, 10002, 10008, 0.70 },
                        { ’M’, 10001, 10003, 11.00 },
                        { ’M’, 10003, 10002, 47.50 },
                        { ’M’, 10002, 10006, 125.00 },
                        { ’M’, 10007, 10008, 225.00 },
                        { ’M’, 10002, 10008, 0.70 },
                        { ’M’, 10001, 10003, 11.00 },
                        { ’M’, 10003, 10002, 47.50 },
                        { ’M’, 10008, 10001, 1034.54}};

static unsigned int trx_nr=0;
pthread_mutex_t mutex;
```

```
/*
 * Function: main()
 */

main()
{  sql_context ctx[THREADS];
   pthread_t thread[THREADS];
   parameters params[THREADS];

   int i;
   pthread_addr_t status;

/* Initialize a process in which to spawn threads.              */
   EXEC SQL ENABLE THREADS;

   EXEC SQL WHENEVER SQLERROR DO err_report(sqlca);

/* Create THREADS sessions by connecting THREADS times, each
   connection in a separate runtime context.                    */
   for(i=0; i<THREADS; i++)
   {  printf(”Start Session %d....”,i);
      EXEC SQL CONTEXT ALLOCATE :ctx[i];
      logon(ctx[i],CONNINFO);
   }

/* Create mutex for transaction retrieval.                      */
   if (pthread_mutex_init(&mutex,pthread_mutexattr_default))
   {  printf(”Can’t initialize mutex\n”);
      exit(1);
   }

/* Spawn threads.                                               */
   for(i=0; i<THREADS; i++)
   {  params[i].ctx=ctx[i];
      params[i].thread_id=i;

      printf(”Thread %d... ”,i);
      if (pthread_create(&thread[i],pthread_attr_default,
            (pthread_startroutine_t)do_transaction,
            (pthread_addr_t) &params[i]))
         printf(”Cant create thread %d\n”,i);
      else
         printf(”Created\n”);
   }
```

```
/* Logoff sessions.                                              */
   for(i=0;i<THREADS;i++)
   {  printf(”Thread %d ....”,i);  /* waiting for thread to end */
      if (pthread_join(thread[i],&status))
         printf(”Error waiting for thread % to terminate\n”, i);
      else
         printf(”stopped\n”);

      printf(”Detach thread...”);
      if (pthread_detach(&thread[i]))
         printf(”Error detaching thread! \n”);
      else
         printf(”Detached!\n”);

      printf(”Stop Session %d....”,i);
      logoff(ctx[i]);

      EXEC SQL CONTEXT FREE :ctx[i];
   }

/* Destroy mutex.                                                */
   if (pthread_mutex_destroy(&mutex))
   {  printf(”Can’t destroy mutex\n”);
      exit(1);
   }
} /* end main()                                                  */


/*
 * Function: do_transaction()
 *
 * Description:  This functions executes one transaction out of
 *               the records array. The records array is managed
 *               by get_transaction().
 */

void do_transaction(params)
parameters *params;
{  struct sqlca sqlca;
   record_log *trx;
   sql_context ctx;
   ctx = params–>ctx;
```

```
/* Done all transactions ?                                       */
   while (trx_nr < (sizeof(records)/sizeof(record_log)))
   {  get_transaction(&trx);
      EXEC SQL WHENEVER SQLERROR DO err_report(sqlca);

   /* Use the specified SQL context to perform the executable SQL
      statements that follow.                                    */
      EXEC SQL CONTEXT USE :ctx;

      printf(”Thread %d executing transaction\n”,
              params–>thread_id);
      switch(trx–>action)
      {  case ’M’:  EXEC SQL UPDATE ACCOUNTS
                             SET    BALANCE=BALANCE+:trx–>amount
                             WHERE  ACCOUNT=:trx–>to_account;
                    EXEC SQL UPDATE ACCOUNTS
                             SET    BALANCE=BALANCE–:trx–>amount
                             WHERE  ACCOUNT=:trx–>from_account;
                    break;
         default:   break;
      }
      EXEC SQL COMMIT;
   }
}

/*
 * Function: err_report()
 *
 * Description: This routine prints the most recent error.
 */

void err_report(sqlca)
struct sqlca sqlca;
{  if (sqlca.sqlcode < 0)
      printf(”\n%.*s\n\n”,sqlca.sqlerrm.sqlerrml,
              sqlca.sqlerrm.sqlerrmc);
   exit(1);
}
```

Developing X/Open Applications

X/Open applications run in a distributed transaction processing (DTP) environment. In an abstract model, an X/Open application calls on *resource managers* (RMs) to provide a variety of services. For example, a database resource manager provides access to data in a database. Resource managers interact with a *transaction manager* (TM), which controls all transactions for the application.

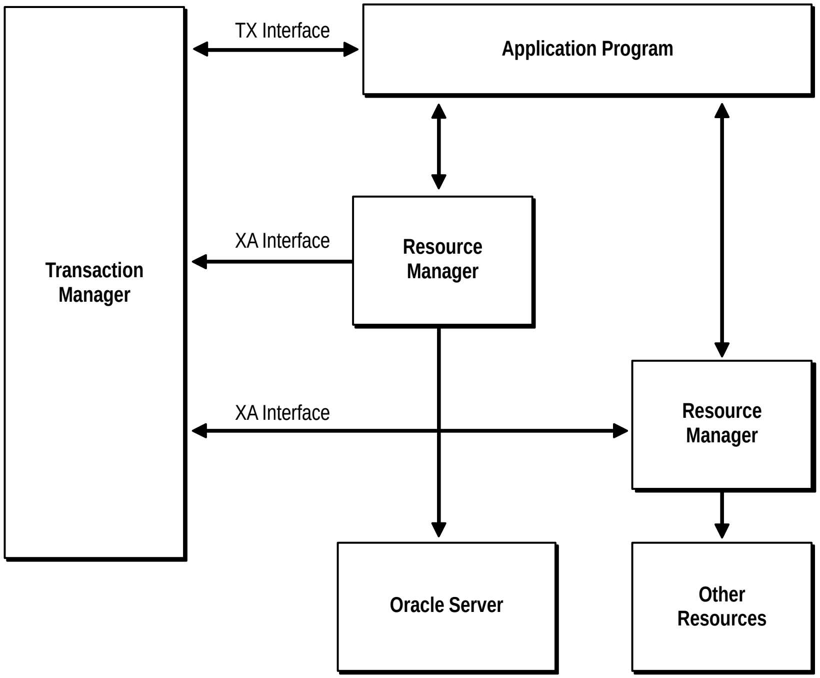


Figure 3 – 5 Hypothetical DTP Model

Figure 3 – 5 shows one way that components of the DTP model can interact to provide efficient access to data in an Oracle database. The DTP model specifies the *XA interface* between resource managers and the transaction manager. Oracle supplies an XA–compliant library, which you must link to your X/Open application. Also, you must specify the *native interface* between your application program and the resource managers.

The DTP model that specifies how a transaction manager and resource managers interact with an application program is described in the X/Open guide *Distributed Transaction Processing Reference Model* and related publications, which you can obtain by writing to

X/Open Company Ltd.
1010 El Camino Real, Suite 380
Menlo Park, CA 94025

For instructions on using the XA interface, see your Transaction Processing (TP) Monitor user's guide.

Oracle–Specific Issues

You can use the precompiler to develop applications that comply with the X/Open standards. However, you must meet the following requirements.

Connecting to Oracle

The X/Open application does not establish and maintain connections to a database. Instead, the transaction manager and the XA interface, which is supplied by Oracle, handle database connections and disconnections transparently. So, normally an X/Open–compliant application does not execute CONNECT statements.

Transaction Control

The X/Open application must not execute statements such as COMMIT, ROLLBACK, SAVEPOINT, and SET TRANSACTION that affect the state of global transactions. For example, the application must not execute the COMMIT statement because the transaction manager handles commits. Also, the application must not execute SQL data definition statements such as CREATE, ALTER, and RENAME because they issue an implicit COMMIT.

The application can execute an internal ROLLBACK statement if it detects an error that prevents further SQL operations. However, this might change in later releases of the XA interface.

OCI Calls

If you want your X/Open application to issue OCI calls, you must use the runtime library routine *sqlld2()*, which sets up an LDA for a specified connection established through the XA interface. For a description of the *sqlld2()* call, see the *Programmer's Guide to the Oracle Call Interface*. Note that the following OCI calls cannot be issued by an X/Open application: OCOM, OCON, OCOF, ONBLON, ORLON, OLON, OLOGOF.

Linking

To get XA functionality, you must link the XA library to your X/Open application object modules. For instructions, see your system–specific Oracle documentation.

CHAPTER

4

Using Embedded SQL

This chapter helps you to understand and apply the basic techniques of embedded SQL programming. You learn how to use

- host variables
- indicator variables
- the fundamental SQL commands that insert, update, select, and delete Oracle data

Using Host Variables

Oracle uses host variables to pass data and status information to your program; your program uses host variables to pass data to Oracle.

Output versus Input Host Variables

Depending on how they are used, host variables are called output or input host variables.

Host variables in the INTO clause of a SELECT or FETCH statement are called *output* host variables because they hold column values output by Oracle. Oracle assigns the column values to corresponding output host variables in the INTO clause.

All other host variables in a SQL statement are called *input* host variables because your program inputs their values to Oracle. For example, you use input host variables in the VALUES clause of an INSERT statement and in the SET clause of an UPDATE statement. They are also used in the WHERE, HAVING, and FOR clauses. Input host variables can appear in a SQL statement wherever a value or expression is allowed.

☞ **Attention:** In an ORDER BY clause, you *can* use a host variable, but it is treated as a constant or literal, and hence the contents of the host variable have no effect. For example, the SQL statement

```
EXEC SQL SELECT ename, empno INTO :name, :number
    FROM emp
    ORDER BY :ord;
```

appears to contain an input host variable, *:ord*. However, the host variable in this case is treated as a constant, and regardless of the value of *:ord*, no ordering is done.

You cannot use input host variables to supply SQL keywords or the names of database objects. Thus, you cannot use input host variables in data definition statements such as ALTER, CREATE, and DROP. In the following example, the DROP TABLE statement is *invalid*:

```
char table_name[30];

printf(”Table name? ”);
gets(table_name);

EXEC SQL DROP TABLE :table_name;  –– host variable not allowed
```

If you need to change database object names at runtime, use dynamic SQL. See Chapter NO TAG for more information about dynamic SQL.

Before Oracle executes a SQL statement containing input host variables, your program must assign values to them. An example follows:

```
int     emp_number;
char    temp[20];
VARCHAR emp_name[20];
/* get values for input host variables */
printf(”Employee number? ”);
gets(temp);
emp_number = atoi(temp);
printf(”Employee name? ”);
gets(emp_name.arr);
emp_name.len = strlen(emp_name.arr);

EXEC SQL INSERT INTO EMP (EMPNO, ENAME)
    VALUES (:emp_number, :emp_name);
```

Notice that the input host variables in the VALUES clause of the INSERT statement are prefixed with colons.

Using Indicator Variables

You can associate any host variable with an optional indicator variable. Each time the host variable is used in a SQL statement, a result code is stored in its associated indicator variable. Thus, indicator variables let you monitor host variables.

You use indicator variables in the VALUES or SET clause to assign nulls to input host variables and in the INTO clause to detect nulls or truncated values in output host variables.

On Input

The values your program can assign to an indicator variable have the following meanings:

–1 Oracle will assign a null to the column, ignoring the value of the host variable.

>=0 Oracle will assign the value of the host variable to the column.

On Output

The values Oracle can assign to an indicator variable have the following meanings:

–1 The column value is null, so the value of the host variable is indeterminate.

0 Oracle assigned an intact column value to the host variable.

>0 Oracle assigned a truncated column value to the host variable. The integer returned by the indicator variable is the original length of the column value, and SQLCODE in SQLCA is set to zero.

–2 Oracle assigned a truncated column variable to the host variable, but the original column value could not be determined (a LONG column, for example).

Remember, an indicator variable must be defined in the Declare Section as a 2–byte integer and, in SQL statements, must be prefixed with a colon and must immediately follow its host variable.

Inserting Nulls

You can use indicator variables to INSERT nulls. Before the INSERT, for each column you want to be null, set the appropriate indicator variable to –1, as shown in the following example:

```
set ind_comm = –1;

EXEC SQL INSERT INTO emp (empno, comm)
    VALUES (:emp_number, :commission:ind_comm);
```

The indicator variable *ind_comm* specifies that a null is to be stored in the COMM column.

You can hardcode the null instead, as follows:

```
EXEC SQL INSERT INTO emp (empno, comm)
    VALUES (:emp_number, NULL);
```

While this is less flexible, it might be more readable. Typically, you insert nulls conditionally, as the next example shows:

```
printf(”Enter employee number or 0 if not available: ”);
scanf(”%d”, &emp_number);

if (emp_number == 0)
    ind_empnum = –1;
else
    ind_empnum = 0;

EXEC SQL INSERT INTO emp (empno, sal)
VALUES (:emp_number:ind_empnum, :salary);
```

Handling Returned Nulls

You can also use indicator variables to manipulate returned nulls, as the following example shows:

```
EXEC SQL SELECT ename, sal, comm
    INTO :emp_name, :salary, :commission:ind_comm
    FROM emp
    WHERE empno = :emp_number;
 if (ind_comm == –1)
    pay = salary;   /* commission is null; ignore it */
else
    pay = salary + commission;
```

Fetching Nulls

When DBMS=V6, you can SELECT or FETCH nulls into a host variable not associated with an indicator variable, as the following example shows:

```
/* assume that commission is NULL */
EXEC SQL SELECT ename, sal, comm
    INTO :emp_name, :salary, :commission
    FROM emp
    WHERE empno = :emp_number;
```

SQLCODE in the SQLCA is set to zero indicating that Oracle executed the statement without detecting an error or exception.

However, when DBMS=V7 or DBMS=V6_CHAR, if you SELECT or FETCH nulls into a host variable not associated with an indicator variable, Oracle issues the following error message:

```
ORA–01405: fetched column value is NULL
```

For more information about the DBMS option, see page 7 – 9, "Using the Precompiler Options".

Testing for Nulls

You can use indicator variables in the WHERE clause to test for nulls, as the following example shows:

```
EXEC SQL SELECT ename, sal
    INTO :emp_name, :salary
    FROM emp
    WHERE :commission INDICATOR :ind_comm IS NULL ...
```

However, you cannot use a relational operator to compare nulls with each other or with other values. For example, the following SELECT statement fails if the COMM column contains one or more nulls:

```
EXEC SQL SELECT ename, sal
    INTO :emp_name, :salary
    FROM emp
    WHERE comm = :commission;
```

The next example shows how to compare values for equality when some of them might be nulls:

```
EXEC SQL SELECT ename, sal
    INTO :emp_name, :salary
    FROM emp
    WHERE (comm = :commission) OR ((comm IS NULL) AND
          (:commission INDICATOR :ind_comm IS NULL));
```

Fetching Truncated Values

When DBMS=V6, if you SELECT or FETCH a truncated column value into a host variable not associated with an indicator variable, Oracle issues the following error message:

```
ORA–01406: fetched column value was truncated
```

However, when DBMS=V7 or V6_CHAR, a warning is generated instead of an error.

The Basic SQL Statements

Executable SQL statements let you query, manipulate, and control Oracle data and create, define, and maintain Oracle objects such as tables, views, and indexes. This chapter focuses on the statements that query and manipulate data.

When executing a data manipulation statement such as INSERT, UPDATE, or DELETE, your only concern, besides setting the values of any input host variables, is whether the statement succeeds or fails. To find out, you simply check the SQLCA. (Executing any SQL statement sets the SQLCA variables.) You can check in the following two ways:

- implicit checking with the WHENEVER statement
- explicit checking of SQLCA variables

For more information about the SQLCA and the WHENEVER statement, see Chapter 9, “Handling Runtime Errors.”

When executing a SELECT statement (query), however, you must also deal with the rows of data it returns. Queries can be classified as follows:

- queries that return no rows (that is, merely check for existence)
- queries that return only one row
- queries that return more than one row

Queries that return more than one row require explicitly declared cursors or the use of *host arrays* (host variables declared as arrays).

Note: Host arrays let you process "batches" of rows. For more information, see Chapter 10 "Using Host Arrays." This chapter assumes the use of scalar host variables.

The following embedded SQL statements let you query and manipulate Oracle data:

SELECT	Returns rows from one or more tables.
INSERT	Adds new rows to a table.
UPDATE	Modifies rows in a table.
DELETE	Removes unwanted rows from a table.

The following embedded SQL statements let you define and manipulate an explicit cursor:

DECLARE	Names the cursor and associates it with a query.
ALLOCATE	Allocates memory for a cursor variable.
OPEN	Executes the query and identifies the active set. (Cannot be used for cursor variables, which must be OPENed on the server.)
FETCH	Advances the cursor and retrieves each row in the active set, one by one.
CLOSE	Disables the cursor (the active set becomes undefined).

In the coming sections, first you learn how to code INSERT, UPDATE, DELETE, and single–row SELECT statements. Then, you progress to multirow SELECT statements. For a detailed discussion of each statement and its clauses, see the *Oracle7 Server SQL Reference.*

Using the SELECT Statement

Querying the database is a common SQL operation. To issue a query you use the SELECT statement. In the following example, you query the EMP table:

```
EXEC SQL SELECT ename, job, sal + 2000
    INTO :emp_name, :job_title, :salary
    FROM emp
    WHERE empno = :emp_number;
```

The column names and expressions following the keyword SELECT make up the *select list*. The select list in our example contains three items. Under the conditions specified in the WHERE clause (and following clauses, if present), Oracle returns column values to the host variables in the INTO clause.

The number of items in the select list should equal the number of host variables in the INTO clause, so there is a place to store every returned value.

In the simplest case, when a query returns one row, its form is that shown in the last example. However, if a query can return more than one row, you must FETCH the rows using a cursor or SELECT them into a host–variable array. Cursors and the FETCH statement are discussed later in this chapter; array processing is discussed in Chapter 10, "Using Host Arrays."

If a query is written to return only one row but might actually return several rows, the result of the SELECT is indeterminate. Whether this causes an error depends on how you specify the SELECT_ERROR option. The default value, YES, generates an error if more than one row is returned.

Available Clauses

You can use all of the following standard SQL clauses in your SELECT statements:

- INTO
- FROM
- WHERE
- CONNECT BY
- START WITH
- GROUP BY
- HAVING

- ORDER BY
- FOR UPDATE OF

Except for the INTO clause, the text of embedded SELECT statements can be executed and tested interactively using SQL*Plus. In SQL*Plus, you use substitution variables or constants instead of input host variables.

Using the INSERT Statement

You use the INSERT statement to add rows to a table or view. In the following example, you add a row to the EMP table:

```
EXEC SQL INSERT INTO emp (empno, ename, sal, deptno)
    VALUES (:emp_number, :emp_name, :salary, :dept_number);
```

Each column you specify in the *column list* must belong to the table named in the INTO clause. The VALUES clause specifies the row of values to be inserted. The values can be those of constants, host variables, SQL expressions, SQL functions such as USER and SYSDATE, or user–defined PL/SQL functions.

The number of values in the VALUES clause must equal the number of names in the column list. However, you can omit the column list if the VALUES clause contains a value for each column in the table, in the order that they are defined in the table.

Using Subqueries

A *subquery* is a nested SELECT statement. Subqueries let you conduct multipart searches. They can be used to

- supply values for comparison in the WHERE, HAVING, and START WITH clauses of SELECT, UPDATE, and DELETE statements
- define the set of rows to be inserted by a CREATE TABLE or INSERT statement
- define values for the SET clause of an UPDATE statement

The following example uses a subquery in an INSERT statement to copy rows from one table to another:

```
EXEC SQL INSERT INTO emp2 (empno, ename, sal, deptno)
    SELECT empno, ename, sal, deptno FROM emp
    WHERE job = :job_title;
```

Notice how the INSERT statement uses the subquery to obtain intermediate results.

Using the UPDATE Statement

You use the UPDATE statement to change the values of specified columns in a table or view. In the following example, you UPDATE the SAL and COMM columns in the EMP table:

```
EXEC SQL UPDATE emp
    SET sal = :salary, comm = :commission
    WHERE empno = :emp_number;
```

You can use the optional WHERE clause to specify the conditions under which rows are UPDATEd. See the section "Using the WHERE Clause" on page 4 – 11.

The SET clause lists the names of one or more columns for which you must provide values. You can use a subquery to provide the values, as the following example shows:

```
EXEC SQL UPDATE emp
    SET sal = (SELECT AVG(sal)*1.1 FROM emp WHERE deptno = 20)
    WHERE empno = :emp_number;
```

Using the DELETE Statement

You use the DELETE statement to remove rows from a table or view. In the following example, you delete all employees in a given department from the EMP table:

```
EXEC SQL DELETE FROM emp
    WHERE deptno = :dept_number;
```

You can use the optional WHERE clause to specify the condition under which rows are DELETEd.

Using the WHERE Clause

You use the WHERE clause to SELECT, UPDATE, or DELETE only those rows in a table or view that meet your search condition. The WHERE–clause *search condition* is a Boolean expression, which can include scalar host variables, host arrays (not in SELECT statements), subqueries, and user–defined stored functions.

If you omit the WHERE clause, all rows in the table or view are processed. If you omit the WHERE clause in an UPDATE or DELETE statement, Oracle sets *sqlwarn[4]* in the SQLCA to ‘W’ to warn that all rows were processed.

Using Cursors

When a query returns multiple rows, you can explicitly define a cursor to

- process beyond the first row returned by the query
- keep track of which row is currently being processed

Or, you can use host arrays; see Chapter 10, “Using Host Arrays.”

A cursor identifies the current row in the set of rows returned by the query. This allows your program to process the rows one at a time. The following statements let you define and manipulate a cursor:

- DECLARE
- OPEN
- FETCH
- CLOSE

First you use the DECLARE statement to name the cursor and associate it with a query.

The OPEN statement executes the query and identifies all the rows that meet the query search condition. These rows form a set called the active set of the cursor. After OPENing the cursor, you can use it to retrieve the rows returned by its associated query.

Rows of the active set are retrieved one by one (unless you use host arrays). You use a FETCH statement to retrieve the current row in the active set. You can execute FETCH repeatedly until all rows have been retrieved.

When done FETCHing rows from the active set, you disable the cursor with a CLOSE statement, and the active set becomes undefined.

The following sections show you how to use these cursor control statements in your application program.

Using the DECLARE Statement

You use the DECLARE statement to define a cursor by giving it a name and associating it with a query, as the following example shows:

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
    SELECT ename, empno, sal
    FROM emp
    WHERE deptno = :dept_number;
```

The cursor name is an identifier used by the precompiler, *not* a host or program variable, and should not be defined in the Declare Section. Cursor names cannot be hyphenated. They can be any length, but only the first 31 characters are significant. For ANSI compatibility, use cursor names no longer than 18 characters.

The SELECT statement associated with the cursor cannot include an INTO clause. Rather, the INTO clause and list of output host variables are part of the FETCH statement.

Because it is declarative, the DECLARE statement must physically (not just logically) precede all other SQL statements referencing the cursor. That is, forward references to the cursor are not allowed. In the following example, the OPEN statement is misplaced:

```
...
EXEC SQL OPEN emp_cursor;

EXEC SQL DECLARE emp_cursor CURSOR FOR
SELECT ename, empno, sal
    FROM emp
    WHERE ename = :emp_name;
```

The cursor control statements (DECLARE, OPEN, FETCH, CLOSE) must all occur within the same precompiled unit. For example, you cannot DECLARE a cursor in file A, then OPEN it in file B.

Your host program can DECLARE as many cursors as it needs. However, in a given file, every DECLARE statement must be unique. That is, you cannot DECLARE two cursors with the same name in one precompilation unit, even across blocks or procedures, because the scope of a cursor is global within a file.

If you will be using many cursors, you might want to specify the MAXOPENCURSORS option. For more information, see Chapter 7, “Running the Pro*C/C++ Precompiler,” and Appendix C, “Performance Tuning.”

Using the OPEN Statement

You use the OPEN statement to execute the query and identify the active set. In the following example, you OPEN a cursor named *emp_cursor*:

```
EXEC SQL OPEN emp_cursor;
```

OPEN positions the cursor just before the first row of the active set. It also zeroes the rows–processed count kept by the third element of SQLERRD in the SQLCA. However, none of the rows is actually retrieved at this point. That will be done by the FETCH statement.

Once you OPEN a cursor, the query’s input host variables are not re–examined until you reOPEN the cursor. Thus, the active set does not change. To change the active set, you must reOPEN the cursor.

Generally, you should CLOSE a cursor before reOPENing it. However, if you specify MODE=ORACLE (the default), you need not CLOSE a cursor before reOPENing it. This can increase performance; for details, see Appendix C, “Performance Tuning.”

The amount of work done by OPEN depends on the values of three precompiler options: HOLD_CURSOR, RELEASE_CURSOR, and MAXOPENCURSORS. For more information, see the section “Using the Precompiler Options” on page 7 – 9.

Using the FETCH Statement

You use the FETCH statement to retrieve rows from the active set and specify the output host variables that will contain the results. Recall that the SELECT statement associated with the cursor cannot include an INTO clause. Rather, the INTO clause and list of output host variables are part of the FETCH statement. In the following example, you FETCH INTO three host variables:

```
EXEC SQL FETCH emp_cursor
    INTO :emp_name, :emp_number, :salary;
```

The cursor must have been previously DECLAREd and OPENed. The first time you execute FETCH, the cursor moves from before the first row in the active set to the first row. This row becomes the current row. Each subsequent execution of FETCH advances the cursor to the next row in the active set, changing the current row. The cursor can only move forward in the active set. To return to a row that has already been FETCHed, you must reOPEN the cursor, then begin again at the first row of the active set.

If you want to change the active set, you must assign new values to the input host variables in the query associated with the cursor, then reOPEN the cursor. When MODE=ANSI, you must CLOSE the cursor before reOPENing it.

As the next example shows, you can FETCH from the same cursor using different sets of output host variables. However, corresponding host variables in the INTO clause of each FETCH statement must have the same datatype.

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
SELECT ename, sal FROM emp WHERE deptno = 20;
...
EXEC SQL OPEN emp_cursor;

EXEC SQL WHENEVER NOT FOUND GOTO ...
for (;;)
{
    EXEC SQL FETCH emp_cursor INTO :emp_name1, :salary1;
    EXEC SQL FETCH emp_cursor INTO :emp_name2, :salary2;
    EXEC SQL FETCH emp_cursor INTO :emp_name3, :salary3;
    ...
}
```

If the active set is empty or contains no more rows, FETCH returns the “no data found” error code to *sqlcode* in the SQLCA, or to the SQLCODE or SQLSTATE status variables. The status of the output host variables is indeterminate. (In a typical program, the WHENEVER NOT FOUND statement detects this error.) To re–use the cursor, you must reOPEN it.

It is an error to FETCH on a cursor under the following conditions:

- before OPENing the cursor
- after a “no data found” condition
- after CLOSEing it

Using the CLOSE Statement

When done FETCHing rows from the active set, you CLOSE the cursor to free the resources, such as storage, acquired by OPENing the cursor. When a cursor is closed, parse locks are released. What resources are freed depends on how you specify the HOLD_CURSOR and RELEASE_CURSOR options. In the following example, you CLOSE the cursor named *emp_cursor*:

```
EXEC SQL CLOSE emp_cursor;
```

You cannot FETCH from a closed cursor because its active set becomes undefined. If necessary, you can reOPEN a cursor (with new values for the input host variables, for example).

When MODE=ORACLE, issuing a COMMIT or ROLLBACK closes cursors referenced in a CURRENT OF clause. Other cursors are unaffected by COMMIT or ROLLBACK and if open, remain open. However, when MODE=ANSI, issuing a COMMIT or ROLLBACK closes *all* explicit cursors. For more information about COMMIT and ROLLBACK, see Chapter 7, "Defining and Controlling Transactions." For more information about the CURRENT OF clause, see the next section.

Optimizer Hints

The Pro*C/C++ Precompiler supports optimizer hints in SQL statements. An *optimizer hint* is a suggestion to the Oracle SQL optimizer that can override the optimization approach that would normally be taken. You can use hints to specify the

- optimization approach for a SQL statement
- access path for each referenced table
- join order for a join
- method used to join tables

Hints allow you to choose between rule–based and cost–based optimization. With cost–based optimization, you can use further hints to maximize throughput or response time.

Issuing Hints

You can issue an optimizer hint inside a C or C++ style comment, immediately after a SELECT, DELETE, or UPDATE command. You indicate that the comment contains one or more hints by following the comment opener with a plus sign, leaving no space between the opener and the '+'. For example, the following statement uses the ALL_ROWS hint to let the cost–based approach optimize the statement for the goal of best throughput:

```
SELECT /*+ ALL_ROWS (cost-based) */ empno, ename, sal, job
    INTO :emp_rec FROM emp
    WHERE deptno = :dept_number;
```

As shown in this statement, the comment can contain optimizer hints as well as other comments.

For more information about the cost–based optimizer, and optimizer hints, see the *Oracle7 Server Application Developer's Guide.*

Using the CURRENT OF Clause

You use the CURRENT OF *cursor_name* clause in a DELETE or UPDATE statement to refer to the latest row FETCHed from the named cursor. The cursor must be open and positioned on a row. If no FETCH has been done or if the cursor is not open, the CURRENT OF clause results in an error and processes no rows.

The FOR UPDATE OF clause is optional when you DECLARE a cursor that is referenced in the CURRENT OF clause of an UPDATE or DELETE statement. The CURRENT OF clause signals the precompiler to add a FOR UPDATE clause if necessary. For more information, see page 8 – 11, "Using the FOR UPDATE OF Clause".

In the following example, you use the CURRENT OF clause to refer to the latest row FETCHed from a cursor named *emp_cursor*:

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
SELECT ename, sal FROM emp WHERE job = 'CLERK'
FOR UPDATE OF sal;
...
EXEC SQL OPEN emp_cursor;
EXEC SQL WHENEVER NOT FOUND GOTO ...
for (;;) {
    EXEC SQL FETCH emp_cursor INTO :emp_name, :salary;
    ...
    EXEC SQL UPDATE emp SET sal = :new_salary
    WHERE CURRENT OF emp_cursor;
}
```

Restrictions

An explicit FOR UPDATE OF or an implicit FOR UPDATE acquires exclusive row locks. All rows are locked at the OPEN, not as they are FETCHed, and are released when you COMMIT or ROLLBACK. Therefore, you cannot FETCH from a FOR UPDATE cursor after a COMMIT. If you try to do this, Oracle returns a 1002 error code.

Also, you cannot use host arrays with the CURRENT OF clause. For an alternative, see page 10 – 13, "Mimicking CURRENT OF".

Furthermore, you cannot reference multiple tables in an associated FOR UPDATE OF clause, which means that you cannot do joins with the CURRENT OF clause.

Finally, you cannot use dynamic SQL with the CURRENT OF clause.

Using All the Cursor Statements

The following example shows the typical sequence of cursor control statements in an application program:

```
   ...
/* define a cursor */
   EXEC SQL DECLARE emp_cursor CURSOR FOR
      SELECT ename, job
         FROM emp
         WHERE empno = :emp_number
         FOR UPDATE OF job;

/* open the cursor and identify the active set */
   EXEC SQL OPEN emp_cursor;

/* break if the last row was already fetched */
   EXEC SQL WHENEVER NOT FOUND DO break;

/* fetch and process data in a loop */
   for (;;)
   {
      EXEC SQL FETCH emp_cursor INTO :emp_name, :job_title;

/* optional host-language statements that operate on
   the FETCHed data */

      EXEC SQL UPDATE emp
         SET job = :new_job_title
         WHERE CURRENT OF emp_cursor;
   }
...
/* disable the cursor */
   EXEC SQL CLOSE emp_cursor;
   EXEC SQL COMMIT WORK RELEASE;
   ...
```

A Complete Example

The following complete program illustrates the use of a cursor and the FETCH statement. The program prompts for a department number, then displays the names of all employees in that department.

All FETCHes except the final one return a row and, if no errors were detected during the FETCH, a success status code. The final FETCH fails and returns the “no data found” Oracle error code to *sqlca.sqlcode*. The cumulative number of rows actually FETCHed is found in *sqlerrd[2]* in the SQLCA.

```
#include <stdio.h>

/* declare host variables */
char userid[12] = ”SCOTT/TIGER”;
char emp_name[10];
int  emp_number;
int  dept_number;
char temp[32];
void sql_error();

/* include the SQL Communications Area */
#include <sqlca.h>

main()
{  emp_number = 7499;
/* handle errors */
   EXEC SQL WHENEVER SQLERROR do sql_error(”Oracle error”);

/* connect to Oracle */
   EXEC SQL CONNECT :userid;
   printf(”Connected.\n”);

/* declare a cursor */
   EXEC SQL DECLARE emp_cursor CURSOR FOR
   SELECT ename
      FROM emp
      WHERE deptno = :dept_number;

   printf(”Department number? ”);
   gets(temp);
   dept_number = atoi(temp);

/* open the cursor and identify the active set */
   EXEC SQL OPEN emp_cursor;

   printf(”Employee Name\n”);
   printf(”-------------\n”);
/* fetch and process data in a loop
   exit when no more data */
   EXEC SQL WHENEVER NOT FOUND DO break;
   while (1)
   {
      EXEC SQL FETCH emp_cursor INTO :emp_name;
      printf(”%s\n”, emp_name);
   }
   EXEC SQL CLOSE emp_cursor;
   EXEC SQL COMMIT WORK RELEASE;
   exit(0);
}

void
sql_error(msg)
char *msg;
{
   char buf[500];
   int buflen, msglen;

   EXEC SQL WHENEVER SQLERROR CONTINUE;
   EXEC SQL ROLLBACK WORK RELEASE;
   buflen = sizeof (buf);
   sqlglm(buf, &buflen, &msglen);
   printf(”%s\n”, msg);
   printf(”%*.s\n”, msglen, buf);
   exit(1);
}
```

CHAPTER

5

Using Embedded PL/SQL

This chapter shows you how to improve performance by embedding PL/SQL transaction processing blocks in your program. After pointing out the advantages of PL/SQL, this chapter discusses the following subjects:

- embedding PL/SQL blocks
- using host variables
- using indicator variables
- using host arrays
- using cursors
- creating and calling PL/SQL stored subprograms
- using dynamic SQL

Advantages of PL/SQL

This section looks at some of the features and benefits offered by PL/SQL, such as

- better performance
- integration with Oracle
- cursor FOR loops
- procedures and functions
- packages
- PL/SQL tables
- user–defined records

For more information about PL/SQL, see the *PL/SQL User's Guide and Reference.*

Better Performance

PL/SQL can help you reduce overhead, improve performance, and increase productivity. For example, without PL/SQL, Oracle must process SQL statements one at a time. Each SQL statement results in another call to the Server and higher overhead. However, with PL/SQL, you can send an entire block of SQL statements to the Server. This minimizes communication between your application and Oracle.

Integration with Oracle

PL/SQL is tightly integrated with the Oracle Server. For example, most PL/SQL datatypes are native to the Oracle data dictionary. Furthermore, you can use the %TYPE attribute to base variable declarations on column definitions stored in the data dictionary, as the following example shows:

```
job_title  emp.job%TYPE;
```

That way, you need not know the exact datatype of the column. Furthermore, if a column definition changes, the variable declaration changes accordingly and automatically. This provides data independence, reduces maintenance costs, and allows programs to adapt as the database changes.

Cursor FOR Loops

With PL/SQL, you need not use the DECLARE, OPEN, FETCH, and CLOSE statements to define and manipulate a cursor. Instead, you can use a cursor FOR loop, which implicitly declares its loop index as a record, opens the cursor associated with a given query, repeatedly fetches data from the cursor into the record, then closes the cursor. An example follows:

```
DECLARE
...
BEGIN
   FOR emprec IN (SELECT empno, sal, comm FROM emp) LOOP
   IF emprec.comm / emprec.sal > 0.25 THEN ...
   ...
END LOOP;
END;
```

Notice that you use dot notation to reference components in the record.

Procedures and Functions

PL/SQL has two types of subprograms called *procedures* and *functions*, which aid application development by letting you isolate operations. Generally, you use a procedure to perform an action and a function to compute a value.

Procedures and functions provide *extensibility*. That is, they let you tailor the PL/SQL language to suit your needs. For example, if you need a procedure that creates a new department, just write your own as follows:

```
PROCEDURE create_dept
  (new_dname  IN CHAR(14),
   new_loc    IN CHAR(13),
   new_deptno OUT NUMBER(2)) IS
BEGIN
   SELECT deptno_seq.NEXTVAL INTO new_deptno FROM dual;
   INSERT INTO dept VALUES (new_deptno, new_dname, new_loc);
END create_dept;
```

When called, this procedure accepts a new department name and location, selects the next value in a department–number database sequence, inserts the new number, name, and location into the *dept* table, then returns the new number to the caller.

You use *parameter modes* to define the behavior of formal parameters. There are three parameter modes: IN (the default), OUT, and IN OUT. An IN parameter lets you pass values to the subprogram being called. An OUT parameter lets you return values to the caller of a subprogram. An IN OUT parameter lets you pass initial values to the subprogram being called and return updated values to the caller.

The datatype of each actual parameter must be convertible to the datatype of its corresponding formal parameter. Table 3 – 2 in Chapter 3 shows the legal conversions between datatypes.

Packages

PL/SQL lets you bundle logically related types, program objects, and subprograms into a *package*. With the Procedural Database Extension, packages can be compiled and stored in an Oracle database, where their contents can be shared by many applications.

Packages usually have two parts: a specification and a body. The *specification* is the interface to your applications; it declares the types, constants, variables, exceptions, cursors, and subprograms available for use. The *body* defines cursors and subprograms; it implements the specification. In the following example, you "package" two employment procedures:

```
PACKAGE emp_actions IS  –– package specification
  PROCEDURE hire_employee (empno NUMBER, ename CHAR, ...);

  PROCEDURE fire_employee (emp_id NUMBER);
END emp_actions;

PACKAGE BODY emp_actions IS  –– package body
  PROCEDURE hire_employee (empno NUMBER, ename CHAR, ...) IS
  BEGIN
    INSERT INTO emp VALUES (empno, ename, ...);
  END hire_employee;

  PROCEDURE fire_employee (emp_id NUMBER) IS
  BEGIN
    DELETE FROM emp WHERE empno = emp_id;
  END fire_employee;
END emp_actions;
```

Only the declarations in the package specification are visible and accessible to applications. Implementation details in the package body are hidden and inaccessible.

PL/SQL Tables

PL/SQL provides a composite datatype named TABLE. Objects of type TABLE are called *PL/SQL tables*, which are modelled as (but not the same as) database tables. PL/SQL tables have only one column and use a primary key to give you array–like access to rows. The column can belong to any scalar type (such as CHAR, DATE, or NUMBER), but the primary key must belong to type BINARY_INTEGER.

You can declare PL/SQL table types in the declarative part of any block, procedure, function, or package. In the following example, you declare a TABLE type called *NumTabTyp*:

```
...
DECLARE
   TYPE NumTabTyp IS TABLE OF NUMBER
      INDEX BY BINARY_INTEGER;
...
BEGIN
   ...
END;
...
```

Once you define type *NumTabTyp*, you can declare PL/SQL tables of that type, as the next example shows:

```
num_tab  NumTabTyp;
```

The identifier *num_tab* represents an entire PL/SQL table.

You reference rows in a PL/SQL table using array–like syntax to specify the primary key value. For example, you reference the ninth row in the PL/SQL table named *num_tab* as follows:

```
num_tab(9) ...
```

User–Defined Records

You can use the %ROWTYPE attribute to declare a record that represents a row in a table or a row fetched by a cursor. However, you cannot specify the datatypes of components in the record or define components of your own. The composite datatype RECORD lifts those restrictions.

Objects of type RECORD are called *records*. Unlike PL/SQL tables, records have uniquely named components, which can belong to different datatypes. For example, suppose you have different kinds of data about an employee such as name, salary, hire date, and so on. This data is dissimilar in type but logically related. A record that contains such components as the name, salary, and hire date of an employee would let you treat the data as a logical unit.

You can declare record types and objects in the declarative part of any block, procedure, function, or package. In the following example, you declare a RECORD type called *DeptRecTyp*:

```
DECLARE
TYPE DeptRecTyp IS RECORD
    (deptno  NUMBER(4) NOT NULL, –– default is NULL allowed
     dname   CHAR(9),
     loc     CHAR(14));
```

Notice that the component declarations are like variable declarations. Each component has a unique name and specific datatype. You can add the NOT NULL option to any component declaration and so prevent the assigning of nulls to that component.

Once you define type *DeptRecTyp*, you can declare records of that type, as the next example shows:

```
dept_rec  DeptRecTyp;
```

The identifier *dept_rec* represents an entire record.

You use dot notation to reference individual components in a record. For example, you reference the *dname* component in the *dept_rec* record as follows:

```
dept_rec.dname ...
```

Embedding PL/SQL Blocks

The Pro*C/C++ Precompiler treats a PL/SQL block like a single embedded SQL statement. So, you can place a PL/SQL block anywhere in a program that you can place a SQL statement.

To embed a PL/SQL block in your Pro*C/C++ program, simply bracket the PL/SQL block with the keywords EXEC SQL EXECUTE and END–EXEC as follows:

```
EXEC SQL EXECUTE
DECLARE
...
BEGIN
   ...
END;
END–EXEC;
```

The keyword END–EXEC must be followed by a semicolon.

After writing your program, you precompile the source file in the usual way.

When the program contains embedded PL/SQL, you must use the SQLCHECK=SEMANTICS command–line option, since the PL/SQL must be parsed by the Oracle Server. SQLCHECK=SEMANTICS requires the USERID option also, to connect to a server. For more information, see the section “Using the Precompiler Options” on page 7 – 9.

Using Host Variables

Host variables are the key to communication between a host language and a PL/SQL block. Host variables can be shared with PL/SQL, meaning that PL/SQL can set and reference host variables.

For example, you can prompt a user for information and use host variables to pass that information to a PL/SQL block. Then, PL/SQL can access the database and use host variables to pass the results back to your host program.

Inside a PL/SQL block, host variables are treated as global to the entire block and can be used anywhere a PL/SQL variable is allowed. Like host variables in a SQL statement, host variables in a PL/SQL block must be prefixed with a colon. The colon sets host variables apart from PL/SQL variables and database objects.

An Example

The following example illustrates the use of host variables with PL/SQL. The program prompts the user for an employee number, then displays the job title, hire date, and salary of that employee.

```
char username[100], password[20];
char job_title[20], hire_date[9], temp[32];
int emp_number;
float salary;

EXEC SQL INCLUDE SQLCA;

printf(”Username? \n”);
gets(username);
printf(”Password? \n”);
gets(password);

EXEC SQL WHENEVER SQLERROR GOTO sql_error;

EXEC SQL CONNECT :username IDENTIFIED BY :password;
printf(”Connected to Oracle\n”);
for (;;)
{
   printf(”Employee Number (0 to end)? ”);
   gets(temp);
   emp_number = atoi(temp);

   if (emp_number == 0)
   {
      EXEC SQL COMMIT WORK RELEASE;
      printf(”Exiting program\n”);
      break;
   }
/*-------------- begin PL/SQL block -----------------*/
   EXEC SQL EXECUTE
   BEGIN
      SELECT job, hiredate, sal
         INTO :job_title, :hire_date, :salary
         FROM emp
         WHERE empno = :emp_number;
   END;
   END-EXEC;
/*-------------- end PL/SQL block -----------------*/

   printf("Number  Job Title  Hire Date  Salary\n");
   printf("------------------------------------\n");
   printf("%6d  %8.8s  %9.9s  %6.2f\n",
   emp_number, job_title, hire_date, salary);
}
...
exit(0);

sql_error:
EXEC SQL WHENEVER SQLERROR CONTINUE;
EXEC SQL ROLLBACK WORK RELEASE;
printf("Processing error\n");
exit(1);
```

Notice that the host variable *emp_number* is set before the PL/SQL block is entered, and the host variables *job_title*, *hire_date*, and *salary* are set inside the block.

A More Complex Example

In the example below, you prompt the user for a bank account number, transaction type, and transaction amount, then debit or credit the account. If the account does not exist, you raise an exception. When the transaction is complete, you display its status.

```
#include <stdio.h>
#include <sqlca.h>

char username[20];
char password[20];
char status[80];
char temp[32];
int  acct_num;
double trans_amt;
void sql_error();
```

```
main()
{
   char trans_type;

/* printf(”Username? ”);
   gets(username);
   printf(”Password? ”);
   gets(password);
*/
   strcpy(password, ”TIGER”);
   strcpy(username, ”SCOTT”);

   EXEC SQL WHENEVER SQLERROR DO sql_error();
   EXEC SQL CONNECT :username IDENTIFIED BY :password;
   printf(”Connected to Oracle\n”);

   for (;;)
   {
      printf(”Account Number (0 to end)? ”);
      gets(temp);
      acct_num = atoi(temp);

      if(acct_num == 0)
      {
         EXEC SQL COMMIT WORK RELEASE;
         printf(”Exiting program\n”);
         break;
      }

      printf(”Transaction Type – D)ebit or C)redit? ”);
      gets(temp);
      trans_type = temp[0];

      printf(”Transaction Amount? ”);
      gets(temp);
      trans_amt = atof(temp);

/*––––––––––––––––– begin PL/SQL block –––––––––––––––––––*/
      EXEC SQL EXECUTE
      DECLARE
         old_bal      NUMBER(9,2);
         err_msg      CHAR(70);
         nonexistent  EXCEPTION;
      BEGIN
         :trans_type := UPPER(:trans_type);
         IF :trans_type = ’C’ THEN       –– credit the account
            UPDATE accts SET bal = bal + :trans_amt
            WHERE acctid = :acct_num;
            IF SQL%ROWCOUNT = 0 THEN    –– no rows affected
               RAISE nonexistent;
            ELSE
               :status := ’Credit applied’;
            END IF;
         ELSIF :trans_type = ’D’ THEN    –– debit the account
            SELECT bal INTO old_bal FROM accts
               WHERE acctid = :acct_num;
            IF old_bal >= :trans_amt THEN   –– enough funds
               UPDATE accts SET bal = bal – :trans_amt
                  WHERE acctid = :acct_num;
               :status := ’Debit applied’;
            ELSE
               :status := ’Insufficient funds’;
            END IF;
         ELSE
            :status := ’Invalid type: ’ || :trans_type;
         END IF;
         COMMIT;
      EXCEPTION
         WHEN NO_DATA_FOUND OR nonexistent THEN
            :status := ’Nonexistent account’;
         WHEN OTHERS THEN
            err_msg := SUBSTR(SQLERRM, 1, 70);
            :status := ’Error: ’ || err_msg;
      END;
      END–EXEC;
/*----------------- end PL/SQL block ––––––––––––––––––––––– */

      printf(”\nStatus: %s\n”, status);
   }
   exit(0);
}


void
sql_error()
{
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK WORK RELEASE;
    printf(”Processing error\n”);
    exit(1);
}
```

VARCHAR Pseudotype

Recall from Chapter 3 that you can use the VARCHAR datatype to declare variable–length character strings. If the VARCHAR is an input host variable, you must tell Oracle what length to expect. So, set the length component to the actual length of the value stored in the string component.

If the VARCHAR is an output host variable, Oracle automatically sets the length component. However, to use a VARCHAR output host variable in your PL/SQL block, you must initialize the length component *before* entering the block. So, set the length component to the declared (maximum) length of the VARCHAR, as shown here:

```
int     emp_number;
varchar emp_name[10];
float   salary;
...
emp_name.len = 10;   /* initialize length component */

EXEC SQL EXECUTE
  BEGIN
    SELECT ename, sal INTO :emp_name, :salary
        FROM emp
        WHERE empno = :emp_number;
    ...
  END;
END–EXEC;
...
```

Restriction

Do not use C pointer or array syntax in PL/SQL blocks. The PL/SQL compiler does not understand C host–variable expressions and is, therefore, unable to parse them. For example, the following is *invalid*:

```
EXEC SQL EXECUTE
    BEGIN
        :x[5].name := 'SCOTT';
        ...
    END;
END–EXEC;
```

To avoid syntax errors, use a placeholder (a temporary variable), to hold the address of the structure field to populate structures as shown in the following *valid* example:

```
name = &employee.name
EXEC SQL EXECUTE
    BEGIN
        :name := ...;
        ...
    END;
END–EXEC;
```

Using Indicator Variables

PL/SQL does not need indicator variables because it can manipulate nulls. For example, within PL/SQL, you can use the IS NULL operator to test for nulls, as follows:

```
IF variable IS NULL THEN ...
```

And, you can use the assignment operator (:=) to assign nulls, as follows:

```
variable := NULL;
```

However, a host language such as C needs indicator variables because it cannot manipulate nulls. Embedded PL/SQL meets this need by letting you use indicator variables to

- accept nulls input from a host program
- output nulls or truncated values to a host program

When used in a PL/SQL block, indicator variables are subject to the following rules:

- You cannot refer to an indicator variable by itself; it must be appended to its associated host variable.
- If you refer to a host variable with its indicator variable, you must always refer to it that way in the same block.

In the following example, the indicator variable *ind_comm* appears with its host variable *commission* in the SELECT statement, so it must appear that way in the IF statement:

```
...
EXEC SQL EXECUTE
BEGIN
    SELECT ename, comm
        INTO :emp_name, :commission :ind_comm
        FROM emp
        WHERE empno = :emp_number;
    IF :commission :ind_comm IS NULL THEN ...
    ...
END;
END–EXEC;
```

Notice that PL/SQL treats *:commission :ind_comm* like any other simple variable. Though you cannot refer directly to an indicator variable inside a PL/SQL block, PL/SQL checks the value of the indicator variable when entering the block and sets the value correctly when exiting the block.

Handling Nulls

When entering a block, if an indicator variable has a value of –1, PL/SQL automatically assigns a null to the host variable. When exiting the block, if a host variable is null, PL/SQL automatically assigns a value of –1 to the indicator variable. In the next example, if *ind_sal* had a value of –1 before the PL/SQL block was entered, the *salary_missing* exception is raised. An *exception* is a named error condition.

```
...
EXEC SQL EXECUTE
BEGIN
    IF :salary :ind_sal IS NULL THEN
    RAISE salary_missing;
END IF;
...
END;
END–EXEC;
...
```

Handling Truncated Values

PL/SQL does not raise an exception when a truncated string value is assigned to a host variable. However, if you use an indicator variable, PL/SQL sets it to the original length of the string. In the following example, the host program will be able to tell, by checking the value of *ind_name*, if a truncated value was assigned to *emp_name*:

```
...
EXEC SQL EXECUTE
DECLARE
...
new_name  CHAR(10);
BEGIN
    ...
    :emp_name:ind_name := new_name;
    ...
END;
END–EXEC;
```

Using Host Arrays

You can pass input host arrays and indicator arrays to a PL/SQL block. They can be indexed by a PL/SQL variable of type BINARY_INTEGER or by a host variable compatible with that type. Normally, the entire host array is passed to PL/SQL, but you can use the ARRAYLEN statement (discussed later) to specify a smaller array dimension.

Furthermore, you can use a procedure call to assign all the values in a host array to rows in a PL/SQL table. Given that the array subscript range is m .. n, the corresponding PL/SQL table index range is always 1 .. $n – m$ + 1. For example, if the array subscript range is 5 .. 10, the corresponding PL/SQL table index range is 1 .. (10 – 5 + 1) or 1 .. 6.

In the example below, you pass an array named *salary* to a PL/SQL block, which uses the array in a function call. The function is named *median* because it finds the middle value in a series of numbers. Its formal parameters include a PL/SQL table named *num_tab*. The function call assigns all the values in the actual parameter *salary* to rows in the formal parameter *num_tab*.

```
...
float salary[100];

/* populate the host array */

EXEC SQL EXECUTE
  DECLARE
    TYPE NumTabTyp IS TABLE OF REAL
        INDEX BY BINARY_INTEGER;
    median_salary  REAL;
    n  BINARY_INTEGER;
...
  FUNCTION median (num_tab NumTabTyp, n INTEGER)
    RETURN REAL IS
  BEGIN
    -- compute median
  END;
  BEGIN
    n := 100;
    median_salary := median(:salary, n);
    ...
  END;
END-EXEC;
...
```

⚠ **Warning:** In dynamic SQL Method 4, you cannot bind a host array to a PL/SQL procedure with a parameter of type “table.” For more information, see “Using Method 4” on page 11 – 23.

You can also use a procedure call to assign all row values in a PL/SQL table to corresponding elements in a host array. For an example, see the section “Stored Subprograms” on page 5 – 18.

Table 5 – 1 shows the legal conversions between row values in a PL/SQL table and elements in a host array. For example, a host array of type LONG is compatible with a PL/SQL table of type VARCHAR2, LONG, RAW, or LONG RAW. Notably, it is not compatible with a PL/SQL table of type CHAR.

	PL/SQL Table							
Host Array	CHAR	DATE	LONG	LONG RAW	NUMBER	RAW	ROWID	VARCHAR2
CHARF	✓							
CHARZ	✓							
DATE		✓						
DECIMAL					✓			
DISPLAY					✓			
FLOAT					✓			
INTEGER					✓			
LONG	✓		✓					
LONG VARCHAR			✓	✓		✓		✓
LONG VARRAW				✓		✓		
NUMBER					✓			
RAW				✓		✓		
ROWID							✓	
STRING			✓	✓		✓		✓
UNSIGNED					✓			
VARCHAR			✓	✓		✓		✓
VARCHAR2			✓	✓		✓		✓
VARNUM					✓			
VARRAW				✓		✓		

Table 5 – 1 Legal Datatype Conversions

The Pro*C/C++ Precompiler does not check your usage of host arrays. For instance, no index range–checking is done.

ARRAYLEN Statement

Suppose you must pass an input host array to a PL/SQL block for processing. By default, when binding such a host array, the Pro*C/C++ Precompiler uses its declared dimension. However, you might not want to process the entire array. In that case, you can use the ARRAYLEN statement to specify a smaller array dimension. ARRAYLEN associates the host array with a host variable, which stores the smaller dimension. The statement syntax is

```
EXEC SQL ARRAYLEN host_array (dimension);
```

where *dimension* is a 4–byte integer host variable, *not* a literal or expression.

The ARRAYLEN statement must appear along with, but somewhere after, the declarations of *host_array* and *dimension*. You cannot specify an offset into the host array. However, you might be able to use C features for that purpose. The following example uses ARRAYLEN to override the default dimension of a C host array named *bonus*:

```
float bonus[100];
int dimension;
EXEC SQL ARRAYLEN bonus (dimension);
/* populate the host array */
...
dimension = 25;  /* set smaller array dimension */
EXEC SQL EXECUTE
DECLARE
    TYPE NumTabTyp IS TABLE OF REAL
    INDEX BY BINARY_INTEGER;
    median_bonus  REAL;
    FUNCTION median (num_tab NumTabTyp, n INTEGER)
        RETURN REAL IS
  BEGIN
    –– compute median
  END;
  BEGIN
    median_bonus := median(:bonus, :my_dimension);
    ...
  END;
END–EXEC;
```

Only 25 array elements are passed to the PL/SQL block because ARRAYLEN downsizes the array from 100 to 25 elements. As a result, when the PL/SQL block is sent to Oracle for execution, a much smaller host array is sent along. This saves time and, in a networked environment, reduces network traffic.

Using Cursors

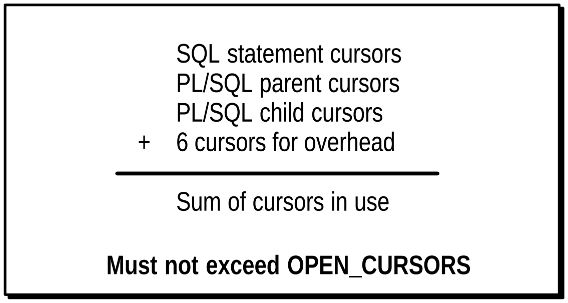
Every embedded SQL statement is assigned a cursor, either explicitly by you in a DECLARE CURSOR statement or implicitly by the precompiler. Internally, the precompiler maintains a cache, called the *cursor cache*, to control the execution of embedded SQL statements. When executed, every SQL statement is assigned an entry in the cursor cache. This entry is linked to a private SQL area in your Program Global Area (PGA) within Oracle.

Various precompiler options, including MAXOPENCURSORS, HOLD_CURSOR, and RELEASE_CURSOR, let you manage the cursor cache to improve performance. For example, RELEASE_CURSOR controls what happens to the link between the cursor cache and private SQL area. If you specify RELEASE_CURSOR=YES, the link is removed after Oracle executes the SQL statement. This frees memory allocated to the private SQL area and releases parse locks. See the "Cursor Control" section on page C – 7 for more information.

For purposes of cursor cache management, an embedded PL/SQL block is treated just like a SQL statement. At run time, a cursor, called a *parent cursor*, is associated with the entire PL/SQL block. A corresponding entry is made to the cursor cache, and this entry is linked to a private SQL area in the PGA.

Each SQL statement inside the PL/SQL block also requires a private SQL area in the PGA. So, PL/SQL manages a separate cache, called the *child cursor cache*, for these SQL statements. Their cursors are called *child cursors*. Because PL/SQL manages the child cursor cache, you do not have direct control over child cursors.

The maximum number of cursors your program can use simultaneously is set by the Oracle initialization parameter OPEN_CURSORS. Figure 5 – 1 shows you how to calculate the maximum number of cursors in use:



SQL statement cursors
PL/SQL parent cursors
PL/SQL child cursors
+ 6 cursors for overhead

Sum of cursors in use

Must not exceed OPEN_CURSORS

Figure 5 – 1 Maximum Cursors in Use

If your program exceeds the limit imposed by OPEN_CURSORS, you get the following Oracle error:

```
ORA–01000: maximum open cursors exceeded
```

You can avoid this error by specifying the RELEASE_CURSOR=YES and HOLD_CURSOR=NO options. If you do not want to precompile the entire program with RELEASE_CURSOR set to YES, simply reset it to NO after each PL/SQL block, as follows:

```
EXEC ORACLE OPTION (RELEASE_CURSOR=YES);
–– first embedded PL/SQL block
EXEC ORACLE OPTION (RELEASE_CURSOR=NO);
–– embedded SQL statements
EXEC ORACLE OPTION (RELEASE_CURSOR=YES);
–– second embedded PL/SQL block
EXEC ORACLE OPTION (RELEASE_CURSOR=NO);
–– embedded SQL statements
```

An Alternative

The MAXOPENCURSORS option specifies the initial size of the cursor cache. For example, when MAXOPENCURSORS=10, the cursor cache can hold up to 10 entries. If a new cursor is needed and HOLD_CURSOR=NO, and there are no free cache entries, the precompiler tries to reuse an entry. If you specify a very low value for MAXOPENCURSORS, the precompiler is forced to reuse the parent cursor more often. All the child cursors are released as soon as the parent cursor is reused.

Stored Subprograms

Unlike anonymous blocks, PL/SQL subprograms (procedures and functions) can be compiled separately, stored in an Oracle database, and invoked. A subprogram explicitly CREATEd using an Oracle tool such as SQL*Plus or SQL*DBA is called a *stored* subprogram. Once compiled and stored in the data dictionary, it is a database object, which can be re–executed without being recompiled.

When a subprogram within a PL/SQL block or stored procedure is sent to Oracle by your application, it is called an *inline* subprogram. Oracle compiles the inline subprogram and caches it in the System Global Area (SGA) but does not store the source or object code in the data dictionary.

Subprograms defined within a package are considered part of the package, and so are called *packaged* subprograms. Stored subprograms not defined within a package are called *stand–alone* subprograms.

Creating Stored Subprograms

You can embed the SQL statements CREATE FUNCTION, CREATE PROCEDURE, and CREATE PACKAGE in a host program, as the following example shows:

```
EXEC SQL CREATE
FUNCTION sal_ok (salary REAL, title CHAR)
RETURN BOOLEAN AS
min_sal  REAL;
max_sal  REAL;
  BEGIN
    SELECT losal, hisal INTO min_sal, max_sal
        FROM sals
        WHERE job = title;
    RETURN (salary >= min_sal) AND
           (salary <= max_sal);
  END sal_ok;
END–EXEC;
```

Notice that the embedded CREATE {FUNCTION | PROCEDURE | PACKAGE} statement is a hybrid. Like all other embedded CREATE statements, it begins with the keywords EXEC SQL (not EXEC SQL EXECUTE). But, unlike other embedded CREATE statements, it ends with the PL/SQL terminator END–EXEC.

In the example below, you create a package that contains a procedure named *get_employees*, which fetches a batch of rows from the EMP table. The batch size is determined by the caller of the procedure, which might be another stored subprogram or a client application.

The procedure declares three PL/SQL tables as OUT formal parameters, then fetches a batch of employee data into the PL/SQL tables. The matching actual parameters are host arrays. When the procedure finishes, it automatically assigns all row values in the PL/SQL tables to corresponding elements in the host arrays.

```
EXEC SQL CREATE OR REPLACE PACKAGE emp_actions AS
    TYPE CharArrayTyp IS TABLE OF VARCHAR2(10)
        INDEX BY BINARY_INTEGER;
    TYPE NumArrayTyp IS TABLE OF FLOAT
        INDEX BY BINARY_INTEGER;
  PROCEDURE get_employees(
    dept_number IN     INTEGER,
    batch_size  IN     INTEGER,
    found       IN OUT INTEGER,
    done_fetch  OUT    INTEGER,
    emp_name    OUT    CharArrayTyp,
    job–title   OUT    CharArrayTyp,
    salary      OUT    NumArrayTyp);
  END emp_actions;
END–EXEC;
```

```
EXEC SQL CREATE OR REPLACE PACKAGE BODY emp_actions AS

    CURSOR get_emp (dept_number IN INTEGER) IS
        SELECT ename, job, sal FROM emp
            WHERE deptno = dept_number;

  PROCEDURE get_employees(
    dept_number IN     INTEGER,
    batch_size  IN     INTEGER,
    found       IN OUT INTEGER,
    done_fetch  OUT    INTEGER,
    emp_name    OUT    CharArrayTyp,
    job_title   OUT    CharArrayTyp,
    salary      OUT    NumArrayTyp) IS

  BEGIN
    IF NOT get_emp%ISOPEN THEN
        OPEN get_emp(dept_number);
    END IF;
    done_fetch := 0;
    found := 0;
    FOR i IN 1..batch_size LOOP
        FETCH get_emp INTO emp_name(i),
        job_title(i), salary(i);
        IF get_emp%NOTFOUND THEN
            CLOSE get_emp;
            done_fetch := 1;
            EXIT;
        ELSE
            found := found + 1;
        END IF;
    END LOOP;
  END get_employees;
END emp_actions;
END–EXEC;
```

You specify the REPLACE clause in the CREATE statement to redefine an existing package without having to drop the package, recreate it, and regrant privileges on it. For the full syntax of the CREATE statement see the *Oracle7 Server SQL Reference.*

If an embedded CREATE {FUNCTION | PROCEDURE | PACKAGE} statement fails, Oracle generates a warning, not an error.

Calling a Stored Subprogram

To invoke (call) a stored subprogram from your host program, you must use an anonymous PL/SQL block. In the following example, you call a stand–alone procedure named *raise_salary*:

```
EXEC SQL EXECUTE
  BEGIN
    raise_salary(:emp_id, :increase);
  END;
END–EXEC;
```

Notice that stored subprograms can take parameters. In this example, the actual parameters *emp_id* and *increase* are C host variables.

In the next example, the procedure *raise_salary* is stored in a package named *emp_actions*, so you must use dot notation to fully qualify the procedure call:

```
EXEC SQL EXECUTE
BEGIN
    emp_actions.raise_salary(:emp_id, :increase);
END;
END–EXEC;
```

An actual IN parameter can be a literal, scalar host variable, host array, PL/SQL constant or variable, PL/SQL table, PL/SQL user–defined record, procedure call, or expression. However, an actual OUT parameter cannot be a literal, procedure call, or expression.

In the following example, three of the formal parameters are PL/SQL tables, and the corresponding actual parameters are host arrays. The program calls the stored procedure *get_employees* (see page 5 – 19) repeatedly, displaying each batch of employee data, until no more data is found. This program is available on–line in the *demo* directory, in the file *sample9.pc*. A SQL script to create the CALLDEMO stored package is available in the file *calldemo.sql*.

```
/*************************************************************
Sample Program 9:  Calling a stored procedure

This program connects to ORACLE using the SCOTT/TIGER
account.  The program declares several host arrays, then
calls a PL/SQL stored procedure (GET_EMPLOYEES in the
CALLDEMO package) that fills the table OUT parameters.  The
PL/SQL procedure returns up to ASIZE values.

Sample9 keeps calling GET_EMPLOYEES, getting ASIZE arrays
each time, and printing the values, until all rows have been
retrieved.  GET_EMPLOYEES sets the done_flag to indicate ”no
more data.”
*************************************************************/
```

```
#include <stdio.h>
#include <string.h>

EXEC SQL INCLUDE sqlca.h;


typedef char asciz[20];
typedef char vc2_arr[11];

EXEC SQL BEGIN DECLARE SECTION;
/* User–defined type for null–terminated strings */
EXEC SQL TYPE asciz  IS STRING(20) REFERENCE;

/* User–defined type for a VARCHAR array element. */
EXEC SQL TYPE vc2_arr IS VARCHAR2(11) REFERENCE;

asciz     username;
asciz     password;
int       dept_no;              /* which department to query? */
vc2_arr   emp_name[10];            /* array of returned names */
vc2_arr   job[10];
float     salary[10];
int       done_flag;
int       array_size;
int       num_ret;                 /* number of rows returned */
EXEC SQL END DECLARE SECTION;

long      SQLCODE;



void print_rows();            /* produces program output      */
void sql_error();             /* handles unrecoverable errors */
```

```
main()
{
   int   i;
   char  temp_buf[32];

/* Connect to ORACLE. */
   EXEC SQL WHENEVER SQLERROR DO sql_error();
   strcpy(username, ”scott”);
   strcpy(password, ”tiger”);
   EXEC SQL CONNECT :username IDENTIFIED BY :password;
   printf(”\nConnected to ORACLE as user: %s\n\n”, username);
   printf(”Enter department number: ”);
   gets(temp_buf);
   dept_no = atoi(temp_buf);/* Print column headers. */
   printf(”\n\n”);
   printf(”%–10.10s%–10.10s%s\n”, ”Employee”, ”Job”, ”Salary”);
   printf(”%–10.10s%–10.10s%s\n”, ”––––––––”, ”–––”, ”––––––”);

/* Set the array size. */
   array_size = 10;

   done_flag = 0;
   num_ret = 0;

/*  Array fetch loop.
 *  The loop continues until the OUT parameter done_flag is set.
 *  Pass in the department number, and the array size––
 *  get names, jobs, and salaries back.
 */
   for (;;)
   {
      EXEC SQL EXECUTE
         BEGIN calldemo.get_employees
            (:dept_no, :array_size, :num_ret, :done_flag,
             :emp_name, :job, :salary);
         END;
      END–EXEC;

      print_rows(num_ret);

      if (done_flag)
         break;
   }

/* Disconnect from the database. */
   EXEC SQL COMMIT WORK RELEASE;
   exit(0);
}
```

```
void
print_rows(n)
int n;
{
   int i;

    if (n == 0)
    {
        printf(”No rows retrieved.\n”);
        return;
    }

    for (i = 0; i < n; i++)
        printf(”%10.10s%10.10s%6.2f\n”,
               emp_name[i], job[i], salary[i]);
}

/* Handle errors. Exit on any error. */
void
sql_error()
{
   char msg[512];
   int buf_len, msg_len;


   EXEC SQL WHENEVER SQLERROR CONTINUE;

   buf_len = sizeof(msg);
   sqlglm(msg, &buf_len, &msg_len);

   printf(”\nORACLE error detected:”);
   printf(”\n%.*s \n”, msg_len, msg);

   EXEC SQL ROLLBACK WORK RELEASE;
   exit(1);
}
```

Remember, the datatype of each actual parameter must be convertible to the datatype of its corresponding formal parameter. Also, before a stored procedure is exited, all OUT formal parameters must be assigned values. Otherwise, the values of corresponding actual parameters are indeterminate.

Remote Access

PL/SQL lets you access remote databases via *database links*. Typically, database links are established by your DBA and stored in the Oracle data dictionary. A database link tells Oracle where the remote database is located, the path to it, and what Oracle username and password to use. In the following example, you use the database link *dallas* to call the *raise_salary* procedure:

```
EXEC SQL EXECUTE
BEGIN
    raise_salary@dallas(:emp_id, :increase);
END;
END–EXEC;
```

You can create synonyms to provide location transparency for remote subprograms, as the following example shows:

```
CREATE PUBLIC SYNONYM raise_salary
FOR raise_salary@dallas;
```

Getting Information about Stored Subprograms

Chapter 3 described how to embed OCI calls in your host program. After calling the library routine SQLLDA to set up the LDA, you can use the OCI call *odessp* to get useful information about a stored subprogram. When you call *odessp*, you must pass it a valid LDA and the name of the subprogram. For packaged subprograms, you must also pass the name of the package. *odessp* returns information about each subprogram parameter such as its datatype, size, position, and so on. For details, see Chapter 6 in the *Programmer's Guide to the Oracle Call Interface*.

You can also use the DESCRIBE_PROCEDURE stored procedure, in the DBMS_DESCRIBE package. See the *Oracle7 Server Application Developer's Guide* for more information about this procedure.

Using Dynamic SQL

Recall that the precompiler treats an entire PL/SQL block like a single SQL statement. Therefore, you can store a PL/SQL block in a string host variable. Then, if the block contains no host variables, you can use dynamic SQL Method 1 to EXECUTE the PL/SQL string. Or, if the block contains a known number of host variables, you can use dynamic SQL Method 2 to PREPARE and EXECUTE the PL/SQL string. If the block contains an unknown number of host variables, you must use dynamic SQL Method 4.

For more information, refer to Chapter 11, "Using Dynamic SQL," and Chapter 12, "Implementing Dynamic SQL Method 4."

⚠ **Warning:** In dynamic SQL Method 4, you cannot bind a host array to a PL/SQL procedure with a parameter of type "table." For more information, see "Using Method 4" on page 11 – 23.

CHAPTER

6

Using C++

This chapter describes how you can use the Pro*C/C++ Precompiler to precompile your C++ embedded SQL application, and how Pro*C/C++ generates C++ compatible code.

Understanding C++ Support

To understand how Pro*C/C++ supports C++, you must understand the basic functional capabilities of Pro*C/C++. In particular, you must be aware of how Pro*C/C++ differs from Pro*C Version 1.

The basic capabilities of Pro*C/C++ are:

- Full C preprocessor support. You can use **#define**, **#include**, **#ifdef**, and other preprocessor directives in your Pro*C/C++ program, to handle constructs that the precompiler itself must process. See page 3 – 2 for more information.
- No need for EXEC SQL ... DECLARE statements to surround host variables. This allows full ANSI C standard function prototyping when function parameters are host variables. See the section "Declaring Host Variables'' on page 3 – 15.
- Use of native C structures as host variables, including the ability to pass structs (or pointers to structs) as host variables to functions, and write functions that return host structures or struct pointers. See page 3 – 21 for more information.

To support its C preprocessor capabilities and to enable host variables to be declared outside a special Declare Section, Pro*C/C++ incorporates a complete C parser. The Pro*C/C++ parser is a C parser; it cannot parse C++ code.

This means that for C++ support, you must be able to disable the C parser, or at least partially disable it. To disable the C parser, the Pro*C/C++ Precompiler includes command–line options to give you control over the extent of C parsing that Pro*C/C++ performs on your source code. The section "Precompiling for C++'' on page 6 – 3 fully describes these options.

No Special Macro Processing

Using C++ with Pro*C/C++ does not require any special preprocessing or special macro processors that are external to Pro*C/C++. There is no need to run a macro processor on the output of the precompiler to achieve C++ compatibility.

If you are a user of a Pro*C Version 1 Precompiler, and you did use macro processors on the precompiler output, you should be able to precompile your C++ applications using Pro*C/C++ with no changes to your code.

Precompiling for C++

To control precompilation so that it accommodates C++, there are four considerations:

- Code emission by the precompiler
- Parsing capability
- The output filename extension
- The location of system header files

Code Emission

You must be able to specify what kind of code, C compatible code or C++ compatible code, the precompiler generates. Pro*C by default generates C code. C++ is not a perfect superset of C. Some changes are required in generated code so that it can be compiled by a C++ compiler.

For example, in addition to emitting your application code, the precompiler interposes calls to its runtime library, SQLLIB. The functions in SQLLIB are C functions. There is no special C++ version of SQLLIB. For this reason, if you want to compile the generated code using a C++ compiler, Pro*C/C++ must declare the functions called in SQLLIB as C functions.

For C output, the precompiler would generate a prototype such as

```
void sqlora(unsigned long *, void *);
```

But for C++ compatible code, the precompiler must generate

```
extern ”C” {
void sqlora(unsigned long *, void *);
};
```

You control the kind of code Pro*C/C++ generates using the precompiler option CODE. In previous releases of Pro*C, there were two values for this option: KR_C and ANSI_C. Pro*C/C++ adds a third option, CPP. The differences between these options can be illustrated by considering how the declaration of the SQLLIB function *sqlora* differs among the three values for the CODE option:

```
void sqlora( /*_ unsigned long *, void * _*/);  /* K&R C */
void sqlora(unsigned long *, void *);           /* ANSI C */
extern ”C” {                                    /* CPP */
void sqlora(unsigned long *, void *);
};
```

When you specify CODE=CPP, the precompiler

- Generates C++ compilable code.
- Gives the output file a platform–specific file extension (suffix), such as ".C" or ".cc", rather than the standard ".c" extension. (You can override this by using the CPP_SUFFIX option.)
- Causes the value of the PARSE option to default to PARTIAL. You can also specify PARSE=NONE. If you specify PARSE=FULL, an error is issued at precompile time.
- Allows the use of the C++ style // comments in your code. This style of commenting is also permitted inside SQL statements and PL/SQL blocks when CODE=CPP.
- Pro*C recognizes SQL optimizer hints that begin with //+.

See Chapter 7 for information about the KR_C and ANSI_C values for the CODE option.

Parsing Code

You must be able to control the effect of the Pro*C / C++ C parser on your code. You do this by using the new PARSE precompiler option, which controls how the precompiler's C parser treats your code.

The values and effects of the PARSE option are:

PARSE=NONE

The value NONE has the following effects:

- C preprocessor directives are understood only inside a declare section.
- You must declare all host variables inside a Declare Section.
- Give precompiler V1.x behavior.

PARSE=PARTIAL

The value PARTIAL has the following effects:

- All preprocessor directives are understood.
- You must declare all host variables inside a Declare Section.

This option value is the default if CODE=CPP.

PARSE=FULL

The value FULL has the following effects:

- The precompiler C parser runs on your code.
- All preprocessor directives are understood.
- You can declare host variables at any place that they can be declared legally in C.

This option value is the default if the value of the CODE option is anything other than CPP. It is an error to specify PARSE=FULL when CODE=CPP.

To generate C++ compatible code, the PARSE option must be either NONE or PARTIAL. If PARSE=FULL, the C parser runs, and it does not understand C++ constructs in your code, such as classes.

Output Filename Extension

Most C compilers expect a default extension of ‘‘.c’’ for their input files. Different C++ compilers, however, can expect different filename extensions. The CPP_SUFFIX option allows you to specify the filename extension that the precompiler generates. The value of this option is a string, without the quotes or the period. For example, CPP_SUFFIX=cc, or CPP_SUFFIX=C.

System Header Files

Pro*C/C++ searches for standard system header files, such as *stdio.h*, in standard locations that are platform specific. For example, on almost all UNIX systems, the file *stdio.h* has the full pathname */usr/include/stdio.h.*

But a C++ compiler has its own version of *stdio.h* that is not in the standard system location. When you are precompiling for C++, you must use the SYS_INCLUDE precompiler option to specify the directory paths that Pro*C/C++ searches to look for system header files. For example:

```
SYS_INCLUDE=(/usr/lang/SC2.0.1/include,/usr/lang/SC2.1.1/include)
```

Use the INCLUDE precompiler option to specify the location of non–system header files. See page 7 – 23. The directories specified by the SYS_INCLUDE option are searched before directories specified by the INCLUDE option.

If PARSE=NONE, the values specified in SYS_INCLUDE and INCLUDE for system files are not relevant, since there is no need for Pro*C/C++ to include system header files. (You can, of course, still include Pro*C/C++–specific headers, such *sqlca.h*, using the EXEC SQL INCLUDE statement.)

Sample Programs

This section includes three sample Pro*C/C++ programs that include C++ constructs. Each of these programs is available on-line, in your *demo* directory.

cppdemo1.pc

```
/*  cppdemo1.pc
 *
 *  Prompts the user for an employee number, then queries the
 *  emp table for the employee’s name, salary and commission.
 *  Uses indicator variables (in an indicator struct) to
 *  determine if the commission is NULL.
 */

#include <iostream.h>
#include <stdio.h>
#include <string.h>

// Parse=partial by default when code=cpp,
// so preprocessor directives are recognized and parsed fully.
#define     UNAME_LEN      20
#define     PWD_LEN        40

// Declare section is required when CODE=CPP and/or
// PARSE={PARTIAL|NONE}
EXEC SQL BEGIN DECLARE SECTION;
  VARCHAR username[UNAME_LEN];  // VARCHAR is an ORACLE pseudotype
  varchar password[PWD_LEN];    // can be in lower case also

  // Define a host structure for the output values
  // of a SELECT statement
  struct empdat {
      VARCHAR   emp_name[UNAME_LEN];
      float     salary;
      float     commission;
  } emprec;

  // Define an indicator struct to correspond to the
  // host output struct
  struct empind {
      short     emp_name_ind;
      short     sal_ind;
      short     comm_ind;
  } emprec_ind;
  // Input host variables
  int   emp_number;
  int   total_queried;
EXEC SQL END DECLARE SECTION;

// Define a C++ class object to match the desired
// struct from the above declare section.
class emp {
  char  ename[UNAME_LEN];
  float salary;
  float commission;
public:
  // Define a constructor for this C++ object that
  // takes ordinary C objects.
  emp(empdat&, empind&);
  friend ostream& operator<<(ostream&, emp&);
};

emp::emp(empdat& dat, empind& ind)
{
  strncpy(ename, (char *)dat.emp_name.arr, dat.emp_name.len);
  ename[dat.emp_name.len] = ’\0’;
  this–>salary = dat.salary;
  this–>commission = (ind.comm_ind < 0) ? 0 : dat.commission;
}

ostream& operator<<(ostream& s, emp& e)
{
  return s << e.ename << ” earns ” << e.salary <<
              ” plus ” << e.commission << ” commission.”
           << endl << endl;
}

// Include the SQL Communications Area
// You can use #include or EXEC SQL INCLUDE
#include <sqlca.h>

// Declare error handling function
void sql_error(char *msg);

main()
{
  char temp_char[32];

  // Register sql_error() as the error handler
  EXEC SQL WHENEVER SQLERROR DO sql_error(”ORACLE error:”);

  // Connect to ORACLE.  Program calls sql_error()
  // if an error occurs
  // when connecting to the default database.
  // Note the (char *) cast when
  // copying into the VARCHAR array buffer.
  username.len = strlen(strcpy((char *)username.arr, ”SCOTT”));
  password.len = strlen(strcpy((char *)password.arr, ”TIGER”));

  EXEC SQL CONNECT :username IDENTIFIED BY :password;

  // Here again, note the (char *) cast when using VARCHARs
  cout << ”\nConnected to ORACLE as user: ”
       << (char *)username.arr << endl << endl;

  // Loop, selecting individual employee’s results
  total_queried = 0;
  while (1)
  {
      emp_number = 0;
      printf(”Enter employee number (0 to quit): ”);
      gets(temp_char);
      emp_number = atoi(temp_char);
      if (emp_number == 0)
        break;

      // Branch to the notfound label when the
      // 1403 (”No data found”) condition occurs
      EXEC SQL WHENEVER NOT FOUND GOTO notfound;

      EXEC SQL SELECT ename, sal, comm
         INTO :emprec INDICATOR :emprec_ind // You can also use
                                            // C++ style
         FROM EMP                  // comments in SQL statemtents.
         WHERE EMPNO = :emp_number;

      {
        // Basic idea is to pass C objects to
        // C++ constructors thus
        // creating equivalent C++ objects used in the
        // usual C++ way
        emp e(emprec, emprec_ind);
        cout << e;
      }

      total_queried++;
      continue;
notfound:
      cout << ”Not a valid employee number – try again.”
           << endl << endl;
  } // end while(1)
  cout << endl << ”Total rows returned was ”
       << total_queried << endl;
  cout << ”Have a nice day!” << endl << endl;

  // Disconnect from ORACLE
  EXEC SQL COMMIT WORK RELEASE;
  exit(0);
}


void sql_error(char *msg)
{
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    cout << endl << msg << endl;
    cout << sqlca.sqlerrm.sqlerrmc << endl;
    EXEC SQL ROLLBACK RELEASE;
    exit(1);
}
```

cppdemo2.pc

```
/*  cppdemo2.pc:  Dynamic SQL Method 3
 *
 *  This program uses dynamic SQL Method 3 to retrieve
 *  the names of all employees in a given department
 *  from the EMP table.
 */

#include <iostream.h>
#include <stdio.h>
#include <string.h>

#define USERNAME ”SCOTT”
#define PASSWORD ”TIGER”

/* Include the SQL Communications Area, a structure through
 * which ORACLE makes runtime status information such as error
 * codes, warning flags, and diagnostic text available to the
 * program. Also include the ORACA.
 */
#include <sqlca.h>
#include <oraca.h>

// The ORACA=YES option must be specified
// to enable use of the ORACA
EXEC ORACLE OPTION (ORACA=YES);

EXEC SQL BEGIN DECLARE SECTION;
  char    *username = USERNAME;
  char    *password = PASSWORD;
  VARCHAR  sqlstmt[80];
  VARCHAR  ename[10];
  int      deptno = 10;
EXEC SQL END DECLARE SECTION;

void sql_error(char *msg);
main()
{
  // Call sql_error() function on any error
  // in an embedded SQL statement
    EXEC SQL WHENEVER SQLERROR DO sql_error(”Oracle error”);

  // Save text of SQL current statement in
  // the ORACA if an error occurs.
    oraca.orastxtf = ORASTFERR;

  // Connect to Oracle.
```

```
    EXEC SQL CONNECT :username IDENTIFIED BY :password;
    cout << endl << ”Connected to Oracle.” << endl << endl;

/* Assign a SQL query to the VARCHAR sqlstmt.  Both the
 * array and the length parts must be set properly.  Note
 * that the query contains one host–variable placeholder,
 * v1, for which an actual input host variable must be
 * supplied at OPEN time.
 */
    strcpy((char *)sqlstmt.arr,
           ”SELECT ename FROM emp WHERE deptno = :v1”);
    sqlstmt.len = strlen((char *)sqlstmt.arr);

/* Display the SQL statement and its current input host
 * variable.
 */
    cout << (char *)sqlstmt.arr << endl;
    cout << ”   v1 = ” << deptno << endl << endl <<”Employee”
         << endl << ”––––––––” << endl;

/* The PREPARE statement associates a statement name with
 * a string containing a SELECT statement.  The statement
 * name is a SQL identifier, not a host variable, and
 * therefore does not appear in the Declare Section.
 *
 * A single statement name can be PREPAREd more than once,
 * optionally FROM a different string variable.
 */
    EXEC SQL PREPARE S FROM :sqlstmt;

/* The DECLARE statement associates a cursor with a
 * PREPAREd statement.  The cursor name, like the statement
 * name, does not appear in the Declare Section.

 * A single cursor name can not be DECLAREd more than once.
 */
    EXEC SQL DECLARE C CURSOR FOR S;
/* The OPEN statement evaluates the active set of the
 * PREPAREd query USING the specified input host variables,
 * which are substituted positionally for placeholders in
 * the PREPAREd query.  For each occurrence of a
 * placeholder in the statement there must be a variable
 * in the USING clause.  That is, if a placeholder occurs
 * multiple times in the statement, the corresponding
 * variable must appear multiple times in the USING clause.
 *
 * The USING clause can be omitted only if the statement
 * contains no placeholders.  OPEN places the cursor at the
 * first row of the active set in preparation for a FETCH.
 *
 * A single DECLAREd cursor can be OPENed more than once,
 * optionally USING different input host variables.
 */
    EXEC SQL OPEN C USING :deptno;

/* Break the loop when all data have been retrieved. */

    EXEC SQL WHENEVER NOT FOUND DO break;

/* Loop until the NOT FOUND condition is detected. */

    while (1)
    {
/* The FETCH statement places the select list of the
 * current row into the variables specified by the INTO
 * clause, then advances the cursor to the next row.  If
 * there are more select–list fields than output host
 * variables, the extra fields will not be returned.
 * Specifying more output host variables than select–list
 * fields results in an ORACLE error.
 */
        EXEC SQL FETCH C INTO :ename;

/* Null–terminate the array before output. */

        ename.arr[ename.len] = ’\0’;
        cout << (char *)ename.arr << endl;
        }

/* Print the cumulative number of rows processed by the
 * current SQL statement.
 */
    printf(”\nQuery returned %d row%s.\n\n”, sqlca.sqlerrd[2],
        (sqlca.sqlerrd[2] == 1) ? ”” : ”s”);

/* The CLOSE statement releases resources associated with
 * the cursor.
 */
    EXEC SQL CLOSE C;

/* Commit any pending changes and disconnect from Oracle. */
    EXEC SQL COMMIT RELEASE;

    cout << ”Have a good day!” << endl << endl;
    exit(0);
}
```

```
void sql_error(char *msg)
{
    cout << endl << msg << endl;
    sqlca.sqlerrm.sqlerrmc[sqlca.sqlerrm.sqlerrml] = ’\0’;
    oraca.orastxt.orastxtc[oraca.orastxt.orastxtl] = ’\0’;
    oraca.orasfnm.orasfnmc[oraca.orasfnm.orasfnml] = ’\0’;
    cout << sqlca.sqlerrm.sqlerrmc << endl;
    cout << ”in ” << oraca.orastxt.orastxtc << endl;
    cout << ”on line ” << oraca.oraslnr << ” of ”
         << oraca.orasfnm.orasfnmc << endl << endl;

/* Disable ORACLE error checking to avoid an infinite loop
 * should another error occur within this routine.
 */
    EXEC SQL WHENEVER SQLERROR CONTINUE;

  // Release resources associated with the cursor.
    EXEC SQL CLOSE C;

  // Roll back any pending changes and disconnect from Oracle.
    EXEC SQL ROLLBACK RELEASE;
    exit(1);
}
```

cppdemo3.pc

```
/*
 * cppdemo3.pc : An example of C++ Inheritance
 *
 * This program finds all salesman and prints their names
 * followed by how much they earn in total (ie; including
 * any commissions).
 */

#include <iostream.h>
#include <stdio.h>
#include <sqlca.h>
#include <string.h>

#define NAMELEN 10

class employee {    // Base class is a simple employee
public:
  char ename[NAMELEN];
  int sal;
  employee(char *, int);
};

employee::employee(char *ename, int sal)
{
  strcpy(this->ename, ename);
  this->sal = sal;
}

// A salesman is a kind of employee
class salesman : public employee
{
  int comm;
public:
  salesman(char *, int, int);
  friend ostream& operator<<(ostream&, salesman&);
};

// Inherits employee attributes
salesman::salesman(char *ename, int sal, int comm)
  : employee(ename, sal), comm(comm) {}

ostream& operator<<(ostream& s, salesman& m)
{
  return s << m.ename << m.sal + m.comm << endl;
}
```

```
void print(char *ename, int sal, int comm)
{
  salesman man(ename, sal, comm);
  cout << man;
}

main()
{
  EXEC SQL BEGIN DECLARE SECTION;
    char *uid = ”scott/tiger”;
    char  ename[NAMELEN];
    int   sal, comm;
    short comm_ind;
  EXEC SQL END DECLARE SECTION;

  EXEC SQL WHENEVER SQLERROR GOTO error;

  EXEC SQL CONNECT :uid;
  EXEC SQL DECLARE c CURSOR FOR
    SELECT ename, sal, comm FROM emp WHERE job = ’SALESMAN’
      ORDER BY ename;
  EXEC SQL OPEN c;

  cout << ”Name    Salary” << endl << ”––––––  ––––––” << endl;

  EXEC SQL WHENEVER NOT FOUND DO break;
  while(1)
   {
     EXEC SQL FETCH c INTO :ename, :sal, :comm:comm_ind;
     print(ename, sal, (comm_ind < 0) ? 0 : comm);
   }

  EXEC SQL CLOSE c;
  exit(0);

error:
  cout << endl << sqlca.sqlerrm.sqlerrmc << endl;
  exit(1);
}
```

CHAPTER

7

Running the Pro*C / C++ Precompiler

This chapter tells you how to run the Pro*C / C++ precompiler, and describes the extensive set of precompiler options in detail.

Precompiler Command

To run the Pro*C precompiler, you issue the following command:

```
proc
```

The location of the precompiler differs from system to system. The system or database administrator usually defines logicals or aliases, or uses other system–specific means to make the Pro*C executable accessible.

The INAME= argument specifies the source file to be precompiled. For example, the command

```
proc INAME=test_proc
```

precompiles the file *test_proc.pc* in the current directory, since the precompiler assumes that the filename extension is *.pc*. The INAME option does not have to be the first option on the command line, but if it is, you can omit the option specification. So, the command

```
proc myfile
```

is equivalent to

```
proc INAME=myfile
```

> **Note:** The option names, and option values that do not name specific OS objects, such as filenames, are not case–sensitive. In the examples in this guide, option names are written in upper case, and option values are usually in lower case. When you enter filenames, including the name of the Pro*C precompiler executable itself, always follow the case conventions used by your operating system. In UNIX, the executable is “proc”, in lower case.

Precompiler Options

Many useful options are available at precompile time. They let you control how resources are used, how errors are reported, how input and output are formatted, and how cursors are managed.

The value of an option is a string literal, which represent text or numeric values. For example, for the option

```
...  INAME=my_test
```

the value is a string literal that specifies a filename. But for the option

```
...MAXOPENCURSORS=20
```

the value is numeric.

Some options take Boolean values, and you can represent these with the strings *yes* or *no*, *true* or *false*, or with the integer literals 1 or 0 respectively. For example, the option

```
...  SELECT_ERROR=yes
```

is equivalent to

```
...  SELECT_ERROR=true
```

or

```
...  SELECT_ERROR=1
```

all of which mean that SELECT errors should be flagged at run time.

The option value is always separated from the option name by an equals sign, with no whitespace around the equals sign.

Default Values

Many of the options have default values. The default value of an option is determined by:

- a value built in to the precompiler
- a value set in the Pro*C *system configuration file*
- a value set in a Pro*C *user configuration file*

For example, the option MAXOPENCURSORS specifies the maximum number of cached open cursors. The built–in precompiler default value for this option is 10. However, if MAXOPENCURSORS=32 is specified in the system configuration file, the default now becomes 32. The user configuration file could set it to yet another value, which then overrides the system configuration value. Then, if this option is set on the command line, the new command–line value takes precedence over the precompiler default, the system configuration file specification, and the user configuration file specification.

Finally, an inline specification takes precedence over all preceding defaults. See the section "Configuration Files" on page 7 – 4 for more information about configuration files.

Some options, such as USERID, do not have a precompiler default value. The built–in default values for options that do have them are listed in Table 7 – 1, and in the "Using the Precompiler Options" section starting on page 7 – 9.

Determining Current Values

You can interactively determine the current value for one or more options by using a question mark on the command line. For example, if you issue the command

```
proc ?
```

the complete set of options, along with their current values, is printed to your terminal. (On a UNIX system running the C shell, escape the '?' with a backslash.) In this case, the values are those built into the precompiler, overridden by any values in the system configuration file. But if you issue the command

```
proc config=my_config_file.h ?
```

and there is a file named *my_config_file.h* in the current directory, all options are listed. Values in the user configuration file supply missing values, and supersede values built–in to the Pro*C precompiler, or values specified in the system configuration file.

You can also determine the current value of a single option, by simply specifying that option name, followed by =?. For example

```
proc maxopencursors=?
```

prints the current default value for the MAXOPENCURSORS option.

Case Sensitivity

In general, you can use either uppercase or lowercase for precompiler option names and values. However, if your operating system is case sensitive, like UNIX, you must specify filename values, including the name of the Pro*C executable, using the correct combination of upper and lowercase letters.

Configuration Files

A configuration file is a text file that contains precompiler options. Each record (line) in the file contains one option, with its associated value or values. For example, a configuration file might contain the lines

```
FIPS=YES
MODE=ANSI
CODE=ANSI_C
```

to set defaults for the FIPS, MODE, and CODE options.

There is a single system configuration file for each Oracle installation. The name of the system configuration file is *pmscfg.h*. The location of the file is system specific. On most UNIX systems, the file specification is *$ORACLE_HOME/proc/pmscfg.h*.

Each Pro*C user can have one or more private configuration files. The name of the configuration file must be specified using the CONFIG= precompiler option. See the "Using the Precompiler Options" section starting on page 7 – 9.

> **Note:** You cannot nest configuration files. This means that CONFIG= is not a valid option inside a configuration file.

What Occurs during Precompilation?

During precompilation, Pro*C generates C or C++ code that replaces the SQL statements embedded in your host program. The generated code contains data structures that indicate the datatype, length, and address of host variables, as well as other information required by the Oracle runtime library, SQLLIB. The generated code also contains the calls to SQLLIB routines that perform the embedded SQL operations.

> **Note:** The precompiler does *not* generate calls to Oracle Call Interface (OCI) routines.

The precompiler can issue warnings and error messages. These messages have the prefix PCC–, and are described in the *Oracle7 Server Messages* manual.

Table 7 – 1 is a quick reference to the major precompiler options. It summarizes the section "Using the Precompiler Options" starting on page 7 – 9. The options that are accepted, but do not have any affect, are not included in this table.

Scope of Options

A precompilation unit is a file containing C code and one or more embedded SQL statements. The options specified for a given precompilation unit affect only that unit; they have no effect on other units. For example, if you specify HOLD_CURSOR=YES and RELEASE_CURSOR=YES for unit A but not for unit B, SQL statements in unit A run with these HOLD_CURSOR and RELEASE_CURSOR values, but SQL statements in unit B run with the default values.

Syntax	Default	Specifics
AUTO_CONNECT=YES|NO	NO	Automatic OPS$ logon
CODE=ANSI_C | KR_C | CPP	KR_C	kind of C code generated
COMP_CHARSET= MULTI_BYTE|SINGLE_BYTE	MULTI_BYTE	the character set type the C/C++ compiler supports
CONFIG=<filename>	none	user’s private configuration file
CPP_SUFFIX=<extension>	none	specify the default filename extension for output files
DBMS=V6 | V7 | NATIVE|V6_CHAR	NATIVE	compatibility (V6, Oracle7, or the database version to which you are connected at precompile time)
DEFINE=<name>	none	a name for use by the Pro*C preprocessor
DEF_SQLCODE=YES|NO	NO	generate a macro to #define SQLCODE
ERRORS=YES | NO	YES	where to direct error messages (NO means only to listing file, and not to terminal)
FIPS=NONE | SQL89 | SQL2	none	whether to flag ANSI/ISO non–compliance
HOLD_CURSOR=YES |NO	NO	how cursor cache handles SQL statement
INAME=<filename>	none	name of the input file
INCLUDE=<pathname>	none	directory path for EXEC SQL INCLUDE or #include statements
LINES=YES | NO	NO	whether #line directives are generated
LNAME=<filename>	none	name of listing file
LTYPE=NONE|SHORT| LONG	LONG	type of listing file to be generated, if any
MAXLITERAL=10..1024	1024	maximum length (bytes) of string literals in generated C code
MAXOPENCURSORS=5..255	10	number of concurrent cached open cursors
MODE=ANSI|ISO|ORACLE	ORACLE	ANSI/ISO or Oracle behavior
NLS_CHAR=(<var1>, ..., <varn>)	none	specify NLS character variables
NLS_LOCAL=YES|NO	NO	control NLS character semantics
ONAME=<filename>	NONE	name of the output (code) file
ORACA=YES|NO	NO	whether to use the ORACA
PARSE=NONE | PARTIAL | FULL	FULL	whether Pro*C parses (with a C parser) the .pc source.
RELEASE_CURSOR=YES|NO	NO	control release of cursors from cursor cache
SELECT_ERROR=YES|NO	YES	flagging of SELECT errors
SQLCHECK=SEMANTICS|SYNTAX	SYNTAX	kind of compile time SQL checking
SYS_INCLUDE=<pathname>	none	directory where system header files, such as iostream.h, are found
UNSAFE_NULL=YES|NO	NO	UNSAFE_NULL=YES disables the ORA–01405 message
USERID=<username>/<password>	none	username/password[@dbname] connect string
VARCHAR=YES|NO	NO	allow the use of implicit VARCHAR structures

Table 7 – 1 Precompiler Options

Entering Options

You can enter any precompiler option on the command line; many can also be entered inline in the precompiler program source file, using the EXEC ORACLE OPTION statement.

On the Command Line

You enter precompiler options on the command line using the following syntax:

```
... [OPTION_NAME=value] [OPTION_NAME=value] ...
```

Separate each option=value specification with one or more spaces. For example, you might enter the following:

```
... CODE=ANSI_C MODE=ANSI
```

Inline

You enter options inline by coding EXEC ORACLE statements, using the following syntax:

```
EXEC ORACLE OPTION (OPTION_NAME=value);
```

For example, you might code the following:

```
EXEC ORACLE OPTION (RELEASE_CURSOR=yes);
```

An option entered inline overrides the same option entered on the command line, or specified in a configuration file.

Uses for EXEC ORACLE

The EXEC ORACLE feature is especially useful for changing option values during precompilation. For example, you might want to change HOLD_CURSOR and RELEASE_CURSOR on a statement–by–statement basis. Appendix C shows you how to optimize runtime performance using inline options.

Specifying options inline or in a configuration file is also helpful if your operating system limits the number of characters you can enter on the command line.

Scope of EXEC ORACLE

An EXEC ORACLE statement stays in effect until textually superseded by another EXEC ORACLE statement specifying the same option. In the following example, HOLD_CURSOR=NO stays in effect until superseded by HOLD_CURSOR=YES:

```
char emp_name[20];
int  emp_number, dept_number;
float salary;

EXEC SQL WHENEVER NOT FOUND DO break;
EXEC ORACLE OPTION (HOLD_CURSOR=NO);

EXEC SQL DECLARE emp_cursor CURSOR FOR
SELECT empno, deptno FROM emp;

EXEC SQL OPEN emp_cursor;
printf(
”Employee Number  Department\n--------------------------\n”);
for (;;)
{
   EXEC SQL FETCH emp_cursor INTO :emp_number, :dept_number;
   printf(”%d\t%d\n”, emp_number, dept_number);
}

EXEC SQL WHENEVER NOT FOUND CONTINUE;
for (;;)
{
   printf(”Employee number: ”);
   scanf(”%d”, &emp_number);
   if (emp_number == 0)
      break;
   EXEC ORACLE OPTION (HOLD_CURSOR=YES);
   EXEC SQL SELECT ename, sal
      INTO :emp_name, :salary
      FROM emp WHERE empno = :emp_number;
   printf(”Salary for %s is %6.2f.\n”, emp_name, salary);
}
```

Using the Precompiler Options

This section is organized for easy reference. It lists the precompiler options alphabetically, and for each option gives its purpose, syntax, and default value. Usage notes that help you understand how the option works are also provided.

AUTO_CONNECT

Purpose

Allows automatic connection to the OPS$ account.

Syntax

AUTO_CONNECT={YES | NO}

Default

NO

Usage Notes

Can be entered only on the command line or in a configuration file.

If AUTO_CONNECT=YES, and the application is not already connected to a database when it processes the first executable SQL statement, it attempts to connect using the userid

```
OPS$<username>
```

where *username* is your current operating system user or task name and OPS$username is a valid Oracle userid.

When AUTO_CONNECT=NO, you must use the CONNECT statement in your program to connect to Oracle.

CODE

Purpose

Specifies the format of C function prototypes generated by the Pro*C precompiler. (A *function prototype* declares a function and the datatypes of its arguments.) The precompiler generates function prototypes for SQL library routines, so that your C compiler can resolve external references. The CODE option lets you control the prototyping.

Syntax

CODE={ANSI_C|KR_C | CPP}

Default

KR_C

Usage Notes

Can be entered inline or on the command line.

ANSI C standard X3.159–1989 provides for function prototyping. When CODE=ANSI_C, Pro*C/C++ generates full function prototypes, which conform to the ANSI C standard. An example follows:

```
extern void sqlora(long *, void *);
```

The precompiler can also generate other ANSI–approved constructs such as the **const** type qualifier.

When CODE=KR_C (the default), the precompiler comments out the argument lists of generated function prototypes, as shown here:

```
extern void sqlora(/*_ long *, void * _*/);
```

So, specify CODE=KR_C if your C compiler is not compliant with the X3.159 standard.

When CODE=CPP, the precompiler generates C++ compatible code.

COMP_CHARSET

Purpose

Indicates to the Pro*C/C++ Precompiler whether multi–byte character sets are (or are not) supported by the compiler to be used. It is intended for use by developers working in a multi–byte client–side environment (for example, when NLS_LANG is set to a multi–byte character set).

Syntax

COMP_CHARSET={MULTI_BYTE | SINGLE_BYTE}

Default

MULTI_BYTE

Usage Notes

Can be entered only on the command line.

With COMP_CHARSET=MULTI_BYTE (default), Pro*C/C++ generates C code that is to be compiled by a compiler that supports multi–byte NLS character sets.

With COMP_CHARSET=SINGLE_BYTE, Pro*C/C++ generates C code for single–byte compilers that addresses a complication that *may* arise from the ASCII equivalent of a backslash (\) character in the second byte of a double–byte character in a multi–byte string. In this case, the backslash (\) character is “escaped” with another backslash character preceding it.

> **Note:** The need for this feature is common when developing in a Shift–JIS environment with older C compilers.

This option has no effect when NLS_LANG is set to a single–byte character set.

CONFIG

Purpose Specifies the name of a user configuration file.

Syntax CONFIG=<filename>

Default None

Usage Notes Can be entered only on the command line.

This option is the only way you can inform Pro*C/C++ of the name and location of user configuration files.

CPP_SUFFIX

Purpose The CPP_SUFFIX option allows you to specify the filename extension that the precompiler appends to the C++ output file generated when the CODE=CPP option is specified.

Syntax CPP_SUFFIX=<filename extension>

Default System–specific.

Usage Notes Most C compilers expect a default extension of ".c'' for their input files. Different C++ compilers, however, can expect different filename extensions. The CPP_SUFFIX option allows you to specify the filename extension that the precompiler generates. The value of this option is a string, without the quotes or the period. For example, CPP_SUFFIX=cc, or CPP_SUFFIX=C.

DBMS

Purpose

Specifies whether Oracle follows the semantic and syntactic rules of Oracle Version 6, Oracle7, or the native version of Oracle (that is, the version to which the application is connected).

Syntax

DBMS={NATIVE | V6 | V7 | V6_CHAR}

Default

NATIVE

Usage Notes

Can be entered only on the command line, or in a configuration file.

The DBMS option lets you control the version–specific behavior of Oracle. When DBMS=NATIVE (the default), Oracle follows the semantic and syntactic rules of the database version to which the application is connected.

When DBMS=V6, V6_CHAR, or DBMS=V7, Oracle follows the respective rules for Oracle Version 6 or Oracle7. A summary of the differences between DBMS=V6, DBMS=V6_CHAR, and DBMS=V7 follows:

- When DBMS=V6 or V6_CHAR, Oracle treats string literals like variable–length character values. However, when DBMS=V7, Oracle treats string literals like fixed–length character values, and CHAR semantics change slightly to comply with the current SQL standard.
- When DBMS=V6, Oracle treats local CHAR variables in a PL/SQL block like variable–length character values. When DBMS=V6_CHAR, however, Oracle treats the CHAR variables like SQL standard, fixed–length character values.
- When DBMS=V6 or V6_CHAR, Oracle treats the return value of the SQL function USER like a variable–length character value. However, when DBMS=V7, Oracle treats the return value of USER like a SQL standard, fixed–length character value.
- When DBMS=V6 or V6_CHAR, the default Oracle external datatype for variables that have the C type **char** or **char[n]** is CHAR. When DBMS=V7, the default external datatype is CHARZ for **char[n]** and VARCHAR2 for **char**.

- When DBMS=V6 (but not V6_CHAR), if you process a multirow query that calls a SQL group function such as AVG or COUNT, the function is called at OPEN time. When DBMS=V7 or V6_CHAR, however, the function is called at FETCH time. At OPEN time or FETCH time, if the function call fails, Oracle issues an error message immediately. Thus, the DBMS value affects error reporting slightly.
- When DBMS=V6, no error is returned if a SELECT or FETCH statement selects a null, and there is no indicator variable associated with the output host variable. When DBMS=V7 or V6_CHAR, SELECTing or FETCHing a null column or expression into a host variable that has no associated indicator variable causes an error (SQLSTATE is "22002"; SQLCODE is –01405).
- When DBMS={V6 | V6_CHAR}, a DESCRIBE operation of a fixed–length string (in Dynamic SQL Method 4) returns datatype code 1. When DBMS=V7, the DESCRIBE operation returns datatype code 96.
- When DBMS=V6, PCTINCREASE is allowed for rollback segments. When DBMS=V7 or V6_CHAR, PCTINCREASE is not allowed for rollback segments.
- When DBMS=V6, illegal MAXEXTENTS storage parameters are allowed. They are not allowed when DBMS=V7 or V6_CHAR.
- When DBMS=V6, constraints (except NOT NULL) are not enabled. When DBMS=V7 or V6_CHAR, all Oracle7 constraints are enabled.

If you precompile using the DBMS=V6 option, and connect to an Oracle7 database, then a Data Definition Language statement such as

```
CREATE TABLE T1 (COL1 CHAR(10))
```

creates the table using the VARCHAR2 (variable–length) datatype, just as if the CREATE TABLE statement had been

```
CREATE TABLE T1 (COL1 VARCHAR2(10))
```

Situation	**DBMS=V7|V6_CHAR MODE=ANSI**	**DBMS=V7|V6_CHAR MODE=ORACLE**	**DBMS=V6 MODE=ORACLE**
"no data found" warning code	+100	+1403	+1403
fetch nulls without using indicator variables	error –1405	error –1405	no error
fetch truncated values without using indicator variables	no error but SQLWARN(2) is set	no error but SQLWARN(2) is set	error –1406 and SQLWARN(2) is set
cursors closed by COMMIT or ROLLBACK	all explicit	CURRENT OF only	CURRENT OF only (NO TAG)
open an already OPENed cursor	error –2117	no error	no error
close an already CLOSEd cursor	error –2114	no error	no error
SQL group function ignores nulls	no warning	no warning	SQLWARN(3) is set
when SQL group function in multi-row query is called	FETCH time	FETCH time	OPEN time
declare SQLCA structure	optional	required	required (NO TAG)
declare SQLCODE or SQLSTATE status variable	required	optional but Oracle ignores	optional but Oracle ignores (NO TAG)
integrity constraints	enabled	enabled	disabled
PCTINCREASE for rollback segments	not allowed	not allowed	allowed
MAXEXTENTS storage parameters	not allowed	not allowed	allowed

Table 7 – 2 How DBMS and MODE Interact

DEF_SQLCODE

Purpose

Controls whether the Pro*C precompiler generates **#define**'s for SQLCODE.

Syntax

DEF_SQLCODE={NO | YES}

Default

NO

Usage Notes

Can be used only on the command line or in a configuration file.

When DEF_SQLCODE=YES, the precompiler defines SQLCODE in the generated source code as follows:

```
#define SQLCODE sqlca.sqlcode
```

You can then use SQLCODE to check the results of executable SQL statement. The DEF_SQLCODE option is supplied for compliance with standards that require the use of SQLCODE.

In addition, you must also include the SQLCA using one of the following entries in your source code:

```
#include <sqlca.h>
```

or

```
EXEC SQL INCLUDE SQLCA;
```

If the SQLCA is not included, using this option causes a precompile time error.

DEFINE

Purpose

Defines a name that can be used in **#ifdef** and **#ifndef** Pro*C preprocessor directives. The defined name can also be used by the EXEC ORACLE IFDEF and EXEC ORACLE IFNDEF statements.

Syntax

DEFINE=*name*

Default

None

Usage Notes

Can be entered on the command line or inline. You can only use DEFINE to define a name—you cannot define macros with it. For example, the following use of define is not valid:

```
proc my_prog DEFINE=LEN=20
```

Using DEFINE in the correct way, you could do

```
proc my_prog DEFINE=XYZZY
```

And then in *my_prog.pc*, code

```
#ifdef XYZZY
...
#else
...
#endif
```

Or you could just as well code

```
EXEC ORACLE IFDEF XYZZY;
...
EXEC ORACLE ELSE;
...
EXEC ORACLE ENDIF;
```

The following example is *invalid*:

```
#define XYZZY
...
EXEC ORACLE IFDEF XYZZY
...
EXEC ORACLE ENDIF;
```

EXEC ORACLE conditional statements are *valid* only if the macro is defined using EXEC ORACLE DEFINE or the DEFINE option.

If you define a name using DEFINE=, and then conditionally include (or exclude) a code section using the Pro*C preprocessor **#ifdef** (or **#ifndef**) directives, you must also make sure that the name is defined when you run the C compiler. For example, for UNIX *cc*, you must use the –D option to define the name for the C compiler.

ERRORS

Purpose

Specifies whether error messages are sent to the terminal as well as the listing file (YES), or just to the listing file (NO).

Syntax

ERRORS={YES | NO}

Default

YES

Usage Notes

Can be entered only on the command line, or in a configuration file.

FIPS

Purpose

Specifies whether extensions to ANSI SQL are flagged (by the FIPS Flagger). An extension is any SQL element that violates ANSI format or syntax rules, except privilege enforcement rules.

Syntax

FIPS={NONE | SQL89 | SQL2 | YES | NO}

Default

None

Usage Notes

Can be entered inline or on the command line.

When FIPS=YES, the FIPS Flagger is enabled, and warning (not error) messages are issued if you use an Oracle extension to ANSI SQL, or use an ANSI SQL feature in a nonconforming manner. Extensions to ANSI SQL that are flagged at precompile time include the following:

- array interface including the FOR clause
- SQLCA, ORACA, and SQLDA data structures
- dynamic SQL including the DESCRIBE statement
- embedded PL/SQL blocks
- automatic datatype conversion
- DATE, NUMBER, RAW, LONGRAW, VARRAW, ROWID, VARCHAR2, and VARCHAR datatypes
- pointer host variables
- Oracle OPTION statement for specifying runtime options
- IAF statements in user exits
- CONNECT statement
- TYPE and VAR datatype equivalencing statements

- AT <db_name> clause
- DECLARE...DATABASE, ...STATEMENT, and ...TABLE statements
- SQLWARNING condition in WHENEVER statement
- DO *function_name()* and DO **break** actions in WHENEVER statement
- COMMENT and FORCE TRANSACTION clauses in COMMIT statement
- FORCE TRANSACTION and TO SAVEPOINT clauses in ROLLBACK statement
- RELEASE parameter in COMMIT and ROLLBACK statements
- optional colon–prefixing of WHENEVER...GOTO labels, and of host variables in the INTO clause

A sample of Pro*C/C++ output when FIPS=YES is shown below.

```
Pro*C/C++: Release 2.1.1.0.0 – Beta on Thu Sep 22 14:28:43 1994

Copyright (c) Oracle Corporation 1979, 1994.  All rights reserved.

System default option values taken from:
/private2/dve/k72tt/proc/pcscfg.h

Oracle FIPS Flagging Report Version 1.0 – Development

This report lists extensions to ANSI SQL document X3.168–1989
The following extensions were detected:

Violation  Line Number    Description
---------  -----------    -------------------------------------
00707      106            keyword WORK required after ROLLBACK
00709      106            use of RELEASE clause
00710       65            use of dynamic SQL
00717       28  56        use of DO within WHENEVER clause
00724       22  53        invalid datatype

The following non–standard usages were detected:

No extensions were detected.
```

```
The following extensions which will become standard in a future
release of the SQL standard were detected:

Violation  Revision       Line Number    Description
---------  -----------    -----------    -------------------------
–
00704      SQL 2 Draft     30            use of the CONNECT
statement

The following deprecated features were detected:

No extensions were detected.

Found 6 violations of ANSI SQL standard X3.168–1989.

ANSI SQL document comparison cross–reference by line number:

Line Numbers       Document Referenced
------------       -------------------
1 – end            X3.168–1989
The Oracle FIPS Flagger was active for all lines in the source.
```

HOLD_CURSOR

Purpose

Specifies how the cursors for SQL statements and PL/SQL blocks are handled in the cursor cache.

Syntax

HOLD_CURSOR={YES | NO}

Default

NO

Usage Notes

Can be entered inline or on the command line.

You can use HOLD_CURSOR to improve the performance of your program. For more information, see Appendix C.

When a SQL data manipulation statement is executed, its associated cursor is linked to an entry in the cursor cache. The cursor cache entry is in turn linked to an Oracle private SQL area, which stores information needed to process the statement. HOLD_CURSOR controls what happens to the link between the cursor and cursor cache.

When HOLD_CURSOR=NO, after Oracle executes the SQL statement and the cursor is closed, the precompiler marks the link as reusable. The link is reused as soon as the cursor cache entry to which it points is needed for another SQL statement. This frees memory allocated to the private SQL area and releases parse locks.

When HOLD_CURSOR=YES and RELEASE_CURSOR=NO, the link is maintained; the precompiler does not reuse it. This is useful for SQL statements that are executed often because it speeds up subsequent executions. There is no need to reparse the statement or allocate memory for an Oracle private SQL area.

For inline use with implicit cursors, set HOLD_CURSOR before executing the SQL statement. For inline use with explicit cursors, set HOLD_CURSOR before CLOSEing the cursor.

Note that RELEASE_CURSOR=YES overrides HOLD_CURSOR=YES and that HOLD_CURSOR=NO overrides RELEASE_CURSOR=NO. For information showing how these two options interact, see Table C – 1.

INAME

Purpose

Specifies the name of the input file.

Syntax

INAME=<path and filename>

Default

None

Usage Notes

Can be entered only on the command line.

You can omit the filename extension if is *.pc*. If the input filename is the first option on the command line, you can omit the INAME= part of the option. For example:

```
proc sample1 MODE=ansi
```

to precompile the file *sample1.pc*, using ANSI mode. This command is the same as

```
proc INAME=sample1 MODE=ansi
```

INCLUDE

Purpose

Specifies a directory path for files included using the **#include** or EXEC SQL INCLUDE directives.

Syntax

INCLUDE=*pathname* or INCLUDE=(*path_1*,*path_2*,...,*path_n*)

Default

Current directory and paths built into Pro*C

Usage Notes

Can be entered inline or on the command line.

You use INCLUDE to specify a directory path for included files. The precompiler searches directories in the following order:

1. the directory specified in a SYS_INCLUDE precompiler option
2. the current directory
3. the built–in directory for standard header files
4. the directory specified by the INCLUDE option

Because of step 3, you normally do not need to specify a directory path for standard header files such as *sqlca.h* and *sqlda.h*. (On UNIX systems, the precompiler searches for these files in *$ORACLE_HOME/sqllib/public*.)

> **Note:** If you specify a filename without an extension for inclusion, Pro*C assumes an extension of *.h*. So, files to be included should have an extension, even if it is not *.h*.

You must still use INCLUDE to specify a directory path for nonstandard files unless they are stored in the current directory. You can specify more than one path on the command line, as follows:

```
... INCLUDE=<path_1> INCLUDE=<path_2> ...
```

The precompiler searches first in the current directory, then in the directory for standard header files, and finally in the directory named by *path1*, then in the directory named by *path2*.

⚠ **Warning:** The precompiler looks for a file in the current directory first—even if you specify a directory path. So, if the file you want to include resides in another directory, make sure no file with the same name resides in the current directory.

The syntax for specifying a directory path using the INCLUDE option is system specific. Follow the conventions used for your operating system.

LINES

Purpose Specifies whether the Pro*C precompiler adds **#line** preprocessor directives to its output file.

Syntax LINES={YES | NO}

Default NO

Usage Notes Can be entered only on the command line.

The LINES option helps with debugging. When LINES=YES, the Pro*C precompiler adds **#line** preprocessor directives to its output file.

Normally, your C compiler increments its line count after each input line is processed. The **#line** directives force the compiler to reset its input line counter so that lines of precompiler–generated code are not counted. Moreover, when the name of the input file changes, the next **#line** directive specifies the new filename.

The C compiler uses the line numbers and filenames to show the location of errors. Thus, error messages issued by the C compiler always refer to your original source files, not the modified source file.

When LINES=NO (the default), the precompiler adds no **#line** directives to its output file.

> **Note:** On page 3 – 3, it is stated that the Pro*C precompiler does not support the **#line** directive. This means that you cannot directly code **#line** directives in the precompiler source. But you can still use the LINES= option to have the precompiler insert **#line** directives for you.

LNAME

Purpose Specifies the name of the listing file.

Syntax LNAME=<filename>

Default None

Usage Notes Can be entered only on the command line.

The default filename extension for the listing file is *.lis*.

LTYPE

Purpose

Specifies the type of listing file generated.

Syntax

LTYPE={NONE | SHORT | LONG}

Default

LONG

Usage Notes

Can be entered on the command line or in a configuration file.

When a listing file is generated, the LONG format is the default. With LTYPE=LONG specified, all of the source code is listed as it is parsed and messages listed as they are generated. In addition, the Pro*C/C++ currently in effect are listed.

With LTYPE=SHORT specified, only the generated messages are listed—no source code—with line references to the source file to help you locate the code that generated the message condition.

With LTYPE=NONE specified, no list file is produced *unless* the LNAME option explicitly specifies a name for a list file. Under the latter condition, the list file *is* generated with LTYPE=LONG assumed.

MAXLITERAL

Purpose

Specifies the maximum length of string literals generated by the precompiler, so that compiler limits are not exceeded.

Syntax

MAXLITERAL=integer, range is 10 to 1024

Default

1024

Usage Notes

Cannot be entered inline.

The maximum value of MAXLITERAL is compiler dependent. For example, some C compilers cannot handle string literals longer than 512 characters, so you would specify MAXLITERAL=512.

Strings that exceed the length specified by MAXLITERAL are divided during precompilation, then recombined (concatenated) at run time.

MAXOPENCURSORS

Purpose

Specifies the number of concurrently open cursors that the precompiler tries to keep cached.

Syntax

MAXOPENCURSORS=*integer*

Default

10

Usage Notes

Can be entered inline or on the command line.

You can use MAXOPENCURSORS to improve the performance of your program. For more information, see Appendix C.

When precompiling separately, use MAXOPENCURSORS as described in "Guidelines for Precompiling Separately" on page 7 – 40.

MAXOPENCURSORS specifies the *initial* size of the SQLLIB cursor cache. If a new cursor is needed, and there are no free cache entries, Oracle tries to reuse an entry. Its success depends on the values of HOLD_CURSOR and RELEASE_CURSOR, and, for explicit cursors, on the status of the cursor itself. Oracle allocates an additional cache entry if it cannot find one to reuse.

If necessary, Oracle keeps allocating additional cache entries until it runs out of memory or reaches the limit set by OPEN_CURSORS. MAXOPENCURSORS must be lower than OPEN_CURSORS by at least 6 to avoid a "maximum open cursors exceeded" Oracle error.

As your program's need for concurrently open cursors grows, you might want to respecify MAXOPENCURSORS to match the need. A value of 45 to 50 is not uncommon, but remember that each cursor requires another private SQL area in the user process memory space. The default value of 10 is adequate for most programs.

MODE

Purpose

Specifies whether your program observes Oracle practices or complies with the current ANSI/ISO SQL standards.

Syntax

MODE={ANSI | ISO | ORACLE}

Default

ORACLE

Usage Notes

Can be entered only on the command line or in a configuration file.

ISO is a synonym for ANSI.

When MODE=ORACLE (the default), your embedded SQL program observes Oracle practices. When MODE=ANSI, your program complies *fully* with the ANSI SQL standard, and the following changes go into effect:

- Issuing a COMMIT or ROLLBACK closes all explicit cursors.
- You cannot OPEN an already open cursor or CLOSE an already closed cursor. (When MODE=ORACLE, you can reOPEN an open cursor to avoid reparsing.)
- You must declare a either a **long** variable named *SQLCODE* or a **char** SQLSTATE[6] variable (uppercase is required for both variables) that is in scope of every EXEC SQL statement. The same *SQLCODE* or *SQLSTATE* variable need not be used in each case; that is, the variable need not be global.
- Declaring the SQLCA is optional. You need not include the SQLCA.
- The “no data found” Oracle warning code returned to SQLCODE becomes +100 instead of +1403. The message text does not change.

NLS_CHAR

Purpose

Specifies which C host character variables are treated by the precompiler as National Language Support (NLS) multi–byte character variables.

Syntax

NLS_CHAR=*varname* or NLS_CHAR=(*var_1*,*var_2*,...,*var_n*)

Default

None.

Usage Notes

Can be entered only on the command line, or in a configuration file.

This option allows you to specify at precompile time a list of the names of one or more host variables that the precompiler must treat as National Language character variables. You can specify only C **char** variables or Pro*C/C++ VARCHARs using this option.

If a you specify in the option list a variable that is not declared in your program, then the precompiler generates no error.

NLS_LOCAL

Purpose

Determines whether NLS character conversions are performed by the precompiler runtime library, or by the Oracle Server.

Syntax

NLS_LOCAL={NO|YES}

Default

NO

Usage Notes

Can be entered only on the command line, or in a configuration file.

When NLS_LOCAL=YES, the runtime library (SQLLIB) performs blank–padding and blank–stripping for host variables that are National Language Support (NLS) multi–byte types.

When NLS_LOCAL=NO, the Oracle Server performs these actions.

When you use the NLS_CHAR option to specify multi–byte character host variables, you must specify NLS_LOCAL=YES.

ONAME

Purpose

Specifies the name of the output file. The output file is the C code file that the precompiler generates.

Syntax

ONAME=<filename>

Default

INAME with an extension determined by CPP_SUFFIX.

Usage Notes

Can be entered only on the command line. Use this option to specify the name of the output file, where the name differs from that of the input (*.pc*) file. For example, if you issue the command

```
proc iname=my_test
```

the default output filename is *my_test.c*. If you want the output filename to be *my_test_1.c*, issue the command

```
proc iname=my_test oname=my_test_1.c
```

Note that you should add the *.c* extension to files specified using ONAME.

The default extension with the ONAME option is platform–specific, but you can override it using the CODE and CPP_SUFFIX options. When CODE=KR_C or ANSI_C, the extension is *c*. When CODE=CPP, you can use the CPP_SUFFIX option to override the platform–specific default.

☞ **Attention:** Oracle recommends that you not let the output filename default, but rather name it explicitly using ONAME.

ORACA

Purpose

Specifies whether a program can use the Oracle Communications Area (ORACA).

Syntax

ORACA={YES | NO}

Default

NO

Usage Notes

Can be entered inline or on the command line.

When ORACA=YES, you must place either the EXEC SQL INCLUDE ORACA or **#include** *oraca.h* statement in your program.

PARSE

Purpose Specifies the way that the Pro*C precompiler parses the source file.

Syntax PARSE={FULL | PARTIAL | NONE}

Default FULL

Usage Notes To generate C++ compatible code, the PARSE option must be either NONE or PARTIAL. If PARSE=FULL, the C parser runs, and it does not understand C++ constructs in your code, such as classes.

See page 6 – 5 for more information on the PARSE option.

With PARSE=FULL or PARSE=PARTIAL Pro*C / C++ fully supports C preprocessor directives, such as **#define**, **#ifdef**, and so on. However, with PARSE=NONE conditional preprocessing is supported by EXEC ORACLE statements as described in "Conditional Preprocessing" on page 7 – 39.

RELEASE_CURSOR

Purpose

Specifies how the cursors for SQL statements and PL/SQL blocks are handled in the cursor cache.

Syntax

RELEASE_CURSOR={YES | NO}

Default

NO

Usage Notes

Can be entered inline or on the command line.

You can use RELEASE_CURSOR to improve the performance of your program. For more information, see Appendix C.

When a SQL data manipulation statement is executed, its associated cursor is linked to an entry in the cursor cache. The cursor cache entry is in turn linked to an Oracle private SQL area, which stores information needed to process the statement. RELEASE_CURSOR controls what happens to the link between the cursor cache and private SQL area.

When RELEASE_CURSOR=YES, after Oracle executes the SQL statement and the cursor is closed, the precompiler immediately removes the link. This frees memory allocated to the private SQL area and releases parse locks. To make sure that associated resources are freed when you CLOSE a cursor, you must specify RELEASE_CURSOR=YES.

When RELEASE_CURSOR=NO and HOLD_CURSOR=YES, the link is maintained. The precompiler does not reuse the link unless the number of open cursors exceeds the value of MAXOPENCURSORS. This is useful for SQL statements that are executed often because it speeds up subsequent executions. There is no need to reparse the statement or allocate memory for an Oracle private SQL area.

For inline use with implicit cursors, set RELEASE_CURSOR before executing the SQL statement. For inline use with explicit cursors, set RELEASE_CURSOR before CLOSEing the cursor.

Note that RELEASE_CURSOR=YES overrides HOLD_CURSOR=YES and that HOLD_CURSOR=NO overrides RELEASE_CURSOR=NO. For a table showing how these two options interact, see Appendix C.

SELECT_ERROR

Purpose

Specifies whether your program generates an error when a SELECT statement returns more than one row, or more rows than a host array can accommodate.

Syntax

SELECT_ERROR={YES | NO}

Default

YES

Usage Notes

Can be entered inline or on the command line.

When SELECT_ERROR=YES, an error is generated when a single–row SELECT returns too many rows, or when an array SELECT returns more rows than the host array can accommodate. The result of the SELECT is indeterminate.

When SELECT_ERROR=NO, no error is generated when a single–row SELECT returns too many rows, or when an array SELECT returns more rows than the host array can accommodate.

Whether you specify YES or NO, a random row is selected from the table. The only way to ensure a specific ordering of rows is to use the ORDER BY clause in your SELECT statement. When SELECT_ERROR=NO and you use ORDER BY, Oracle returns the first row, or the first *n* rows when you are SELECTing into an array. When SELECT_ERROR=YES, whether or not you use ORDER BY, an error is generated when too many rows are returned.

SQLCHECK

Purpose

Specifies the type and extent of syntactic and semantic checking.

Syntax

SQLCHECK={SEMANTICS | FULL | SYNTAX | LIMITED | NONE}

Default

SYNTAX

Usage Notes

Can be entered inline or the command line.

The Pro*C precompiler can help you debug a program by checking the syntax and semantics of embedded SQL statements and PL/SQL blocks. You control the level of checking by entering the SQLCHECK option inline and/or on the command line. However, the level of checking you specify inline cannot be higher than the level you specify (or accept by default) on the command line. For example, if you specify SQLCHECK=SYNTAX on the command line, you cannot specify SQLCHECK=SEMANTICS inline.

SQLCHECK=SEMANTICS | FULL
The precompiler checks the syntax and semantics of

- data manipulation statements such as INSERT and UPDATE
- PL/SQL blocks
- host variable datatypes

However, only syntactic checking is done on data manipulation statements or PL/SQL blocks that use the AT *db_name* clause. No syntax or semantics checking is performed on DDL statements, such as CREATE and ALTER.

Any errors found are reported at precompile time.

The precompiler gets information needed for a semantic check by using embedded DECLARE TABLE statements, or if you specify the USERID option on the command line, by connecting to Oracle and accessing the data dictionary. You need not connect to Oracle if every table referenced in a data manipulation statement or PL/SQL block is defined in a DECLARE TABLE statement.

If you connect to Oracle, but some needed information cannot be found in the data dictionary, you must use DECLARE TABLE statements to supply the missing information. A DECLARE TABLE definition overrides a data dictionary definition if they conflict.

If you embed PL/SQL blocks in a host program, you *must* specify SQLCHECK=SEMANTICS and the USERID option as well.

SQLCHECK=SYNTAX|LIMITED|NONE
The precompiler checks the syntax of

- declarative SQL statements (such as EXEC SQL WHENEVER...)
- Data Manipulation Language statements
- host variables and host variable datatypes

and any errors found are reported at precompile time.

But no semantic checking is done. DECLARE TABLE statements are ignored, and PL/SQL blocks are not allowed.

Specifying the SYNTAX value generates a useable output (code) file, however semantic errors can still occur at runtime.

SYS_INCLUDE

Purpose

Specifies the location of system header files.

Syntax

SYS_INCLUDE=<pathname>

Default

System–specific.

Usage Notes

Pro*C searches for standard system header files, such as *stdio.h*, in standard locations that are platform specific. For example, on almost all UNIX systems, the file *stdio.h* has the full pathname */usr/include/stdio.h.*

But C++ compilers can have system header files, such as *stdio.h*, that are not in the standard system locations. You can use the SYS_INCLUDE command line option to specify a list of directory paths that Pro*C searches to look for system header files. For example

```
SYS_INCLUDE=(/usr/lang/SC2.0.1/include,/usr/lang/SC2.1.1/include)
```

The search path that you specify using SYS_INCLUDE overrides the default header location

If PARSE=NONE, the value specified in SYS_INCLUDE is irrelevant, since there is no need for Pro*C to include system header files. (You must, of course, still include Pro*C–specific headers, such *sqlca.h*.)

The precompiler searches directories in the following order:

1. the directory specified in the SYS_INCLUDE precompiler option
2. the current directory
3. the built–in directory for standard header files
4. the directory specified by the INCLUDE option

Because of step 3, you normally do not need to specify a directory path for standard header files such as *sqlca.h* and *sqlda.h*. (On UNIX systems, the precompiler searches the *$ORACLE_HOME/sqllib/public* directory for these files.)

UNSAFE_NULL

Purpose

Specifying UNSAFE_NULL=YES prevents generation of ORA–01405 messages when fetching NULLs without using indicator variables.

Syntax

UNSAFE_NULL={YES | NO}

Default

NO

Usage Notes

Cannot be entered inline.

The UNSAFE_NULL=YES is allowed only when MODE=ORACLE and DBMS=V7 or V6_CHAR.

The UNSAFE_NULL option has no effect on host variables in an embedded PL/SQL block. You *must* use indicator variables to avoid ORA–01405 errors.

USERID

Purpose

Specifies an Oracle username and password.

Syntax

USERID=username/password

Default

None

Usage Notes

Can be entered only on the command line.

Do not specify this option when using the automatic connect feature, which accepts your Oracle username prefixed with OPS$. The actual value of the "OPS$" string is set as a parameter in the INIT.ORA file.

When SQLCHECK=SEMANTICS, if you want the precompiler to get needed information by connecting to Oracle and accessing the data dictionary, you must also specify USERID.

THREADS

Purpose When THREADS=YES, the precompiler searches for context declarations.

Syntax THREADS={YES | NO}

Default NO

Usage Notes Cannot be entered inline.

This precompiler option is required for any program that requires multi–threaded support.

With THREADS=YES, the precompiler generates an error if no EXEC SQL USE directive is encountered before the first context is visible and an executable SQL statement is found. For more information, see "Developing Multi–threaded Applications" on page 3 – 99.

VARCHAR

Purpose Instructs the Pro*C precompiler to interpret some structs as VARCHAR host variables.

Syntax VARCHAR={NO | YES}

Default NO

Usage Notes Can be entered only on the command line.

When VARCHAR=YES, a C struct that you code as

```
struct {
    short <len>;
    char  <arr>[n];
} name;
```

is interpreted by the precompiler as a **VARCHAR[n]** host variable.

Obsolete Options

The following precompiler options, which are simply parsed and ignored, are not supported in Pro*C/C++ release 2.2:

- ASACC
- IRECLEN
- LRECLEN
- ORECLEN
- PAGELEN
- TEST
- XREF

With Oracle7 and Pro*C Release 1.5 or later, private SQL areas are automatically resized, host variables are rebound only when necessary, and reentrant code is generated automatically for systems that require it. These advances make the AREASIZE, REBIND, and REENTRANT options obsolete. You no longer have to worry about using these options correctly. In fact, if you specify AREASIZE, REBIND, or REENTRANT, you get the following informational message:

```
PCC–I–02355: Invalid or obsolete option, ignored
```

AREASIZE

With some earlier releases of Pro*C, the AREASIZE option specified the size of the initial private SQL area opened for Oracle cursors. You could respecify AREASIZE for each cursor or set of cursors used by your program.

REBIND

The REBIND option specified how often host variables in SQL statements were bound. You could respecify REBIND for each SQL statement or set of SQL statements in your program.

REENTRANT

The REENTRANT option specified whether reentrant code was generated. (A *reentrant* program or subroutine can be reentered before it has finished executing. Thus, it can be used simultaneously by two or more processes.) On some systems, you had to specify REENTRANT=YES.

Conditional Precompilations

Conditional precompilation includes (or excludes) sections of code in your C program based on certain conditions. For example, you might want to include one section of code when precompiling under UNIX and another when precompiling under VMS. Conditional precompiling lets you write programs that can run in different environments.

Conditional sections of code are marked by statements that define the environment and actions to take. You can code C statements as well as EXEC SQL statements in these sections. The following statements let you exercise conditional control over precompilation:

```
EXEC ORACLE DEFINE symbol;   -- define a symbol
EXEC ORACLE IFDEF symbol;    -- if symbol is defined
EXEC ORACLE IFNDEF symbol;   -- if symbol is not defined
EXEC ORACLE ELSE;            -- otherwise
EXEC ORACLE ENDIF;           -- end this control block
```

All EXEC ORACLE statements must be terminated with a semi–colon.

Defining Symbols

You can define a symbol in two ways. Either include the statement

```
EXEC ORACLE DEFINE symbol;
```

in your host program or define the symbol on the command line using the syntax

```
... INAME=filename ... DEFINE=symbol
```

where *symbol* is not case–sensitive.

⚠ **Warning:** The **#define** preprocesssor directive is not the same as the EXEC ORACLE DEFINE command

Some port–specific symbols are predefined for you when the Pro*C Precompiler is installed on your system. For example, predefined operating system symbols include CMS, MVS, MS–DOS, UNIX, and VMS.

An Example

In the following example, the SELECT statement is precompiled only when the symbol *site2* is defined:

```
EXEC ORACLE IFDEF site2;
    EXEC SQL SELECT DNAME
        INTO :dept_name
        FROM DEPT
        WHERE DEPTNO = :dept_number;
EXEC ORACLE ENDIF;
```

Blocks of conditions can be nested as shown in the following example:

```
EXEC ORACLE IFDEF outer;
    EXEC ORACLE IFDEF inner;
    ...
    EXEC ORACLE ENDIF;
EXEC ORACLE ENDIF;
```

You can “comment out” C or embedded SQL code by placing it between IFDEF and ENDIF and *not* defining the symbol.

Guidelines for Precompiling Separately

The following guidelines will help you avoid some common problems.

Referencing Cursors

Cursor names are SQL identifiers, whose scope is the precompilation unit. Hence, cursor operations cannot span precompilation units (files). That is, you cannot DECLARE a cursor in one file, and OPEN or FETCH from it in another file. So, when doing a separate precompilation, make sure all definitions and references to a given cursor are in one file.

Specifying MAXOPENCURSORS

When you precompile the program module that CONNECTs to Oracle, specify a value for MAXOPENCURSORS that is high enough for any of the program modules. If you use MAXOPENCURSORS for another program module, one that does not do a CONNECT, then that value for MAXOPENCURSORS is ignored. Only the value in effect for the CONNECT is used at run time.

Using a Single SQLCA

If you want to use just one SQLCA, you must declare it as global in one of the program modules and as external in the other modules. Use the **extern** storage class, and the following define in your code:

```
#define SQLCA_STORAGE_CLASS extern
```

which tell the precompiler to look for the SQLCA in another program module. Unless you declare the SQLCA as external, each program module uses its own local SQLCA.

Compiling and Linking

To get an executable program, you must compile the output *.c* source files produced by the precompiler, then link the resulting object modules with modules needed from SQLLIB and system–specific Oracle libraries. If you are mixing precompiler code and OCI calls, be sure to also link in the OCI runtime library (*liboci.a* on UNIX systems).

The linker resolves symbolic references in the object modules. If these references conflict, the link fails. This can happen when you try to link third–party software into a precompiled program. Not all third–party software is compatible with Oracle. So, linking your program *shared* might cause an obscure problem. In some cases, linking *stand–alone* or *two–task* might solve the problem.

Compiling and linking are system dependent. On most platforms, example *makefiles* or batch files are supplied that you can use to precompile, compile, and link a Pro*C application. See your system–specific Oracle documentation.

CHAPTER

8

Defining and Controlling Transactions

This chapter explains how to do transaction processing. You learn the basic techniques that safeguard the consistency of your database, including how to control whether changes to Oracle data are made permanent or undone. The following topics are discussed:

- how transactions guard your database
- how transactions begin and end
- making transactions permanent
- undoing transactions
- setting read–only transactions
- overriding default locking
- fetching across COMMITs
- handling distributed transactions
- guidelines

Some Terms You Should Know

Before delving into the subject of transactions, you should know the terms defined in this section.

The jobs or tasks that Oracle manages are called *sessions*. A *user session* is invoked when you run an application program or a tool such as SQL*Forms, and connect to Oracle.

Oracle allows user sessions to work "simultaneously" and share computer resources. To do this, Oracle must control *concurrence*, the accessing of the same data by many users. Without adequate concurrency controls, there might be a loss of *data integrity*. That is, changes to data or structures might be made in the wrong order.

Oracle uses *locks* (sometimes called *enqueues*) to control concurrent access to data. A lock gives you temporary ownership of a database resource such as a table or row of data. Thus, data cannot be changed by other users until you finish with it.

You need never explicitly lock a resource, because default locking mechanisms protect Oracle data and structures. However, you can request *data locks* on tables or rows when it is to your advantage to override default locking. You can choose from several *modes* of locking such as *row share* and *exclusive*.

A *deadlock* can occur when two or more users try to access the same database object. For example, two users updating the same table might wait if each tries to update a row currently locked by the other. Because each user is waiting for resources held by another user, neither can continue until Oracle breaks the deadlock. Oracle signals an error to the participating transaction that had completed the least amount of work, and the "deadlock detected while waiting for resource" Oracle error code is returned to *sqlcode* in the SQLCA.

When a table is being queried by one user and updated by another at the same time, Oracle generates a *read–consistent* view of the table's data for the query. That is, once a query begins and as it proceeds, the data read by the query does not change. As update activity continues, Oracle takes *snapshots* of the table's data and records changes in a *rollback segment*. Oracle uses information in the rollback segment to build read–consistent query results and to undo changes if necessary.

How Transactions Guard Your Database

Oracle is transaction oriented; that is, it uses transactions to ensure data integrity. A transaction is a series of one or more logically related SQL statements you define to accomplish some task. Oracle treats the series of SQL statements as a unit so that all the changes brought about by the statements are either *committed* (made permanent) or *rolled back* (undone) at the same time. If your application program fails in the middle of a transaction, the database is automatically restored to its former (pre–transaction) state.

The coming sections show you how to define and control transactions. Specifically, you learn how to

- begin and end transactions
- use the COMMIT statement to make transactions permanent
- use the SAVEPOINT statement with the ROLLBACK TO statement to undo parts of transactions
- use the ROLLBACK statement to undo whole transactions
- specify the RELEASE option to free resources and log off the database
- use the SET TRANSACTION statement to set read–only transactions
- use the FOR UPDATE clause or LOCK TABLE statement to override default locking

For details about the SQL statements discussed in this chapter, see the *Oracle7 Server SQL Reference.*

How to Begin and End Transactions

You begin a transaction with the first executable SQL statement (other than CONNECT) in your program. When one transaction ends, the next executable SQL statement automatically begins another transaction. Thus, every executable statement is part of a transaction. Because they cannot be rolled back and need not be committed, declarative SQL statements are not considered part of a transaction.

You end a transaction in one of the following ways:

- Code a COMMIT or ROLLBACK statement, with or without the RELEASE option. This *explicitly* makes permanent or undoes changes to the database.
- Code a data definition statement (ALTER, CREATE, or GRANT, for example), which issues an automatic COMMIT before *and* after executing. This *implicitly* makes permanent changes to the database.

A transaction also ends when there is a system failure or your user session stops unexpectedly because of software problems, hardware problems, or a forced interrupt. Oracle rolls back the transaction.

If your program fails in the middle of a transaction, Oracle detects the error and rolls back the transaction. If your operating system fails, Oracle restores the database to its former (pre–transaction) state.

Using the COMMIT Statement

If you do not subdivide your program with the COMMIT or ROLLBACK statement, Oracle treats the whole program as a single transaction (unless the program contains data definition statements, which issue automatic COMMITS).

You use the COMMIT statement to make changes to the database permanent. Until changes are COMMITted, other users cannot access the changed data; they see it as it was before your transaction began. Specifically, the COMMIT statement

- makes permanent all changes made to the database during the current transaction
- makes these changes visible to other users
- erases all savepoints (see the next section)
- releases all row and table locks, but not parse locks

- closes cursors referenced in a CURRENT OF clause or, when MODE=ANSI, closes *all* explicit cursors for the connection specified in the COMMIT statement
- ends the transaction

The COMMIT statement has no effect on the values of host variables or on the flow of control in your program.

When MODE=ORACLE, explicit cursors that are not referenced in a CURRENT OF clause remain open across COMMITs. This can boost performance. For an example, see "Fetching Across COMMITs" on page 8 – 12.

Because they are part of normal processing, COMMIT statements should be placed inline, on the main path through your program. Before your program terminates, it must explicitly COMMIT pending changes. Otherwise, Oracle rolls them back. In the following example, you commit your transaction and disconnect from Oracle:

```
EXEC SQL COMMIT WORK RELEASE;
```

The optional keyword WORK provides ANSI compatibility. The RELEASE option frees all Oracle resources (locks and cursors) held by your program and logs off the database.

You need not follow a data definition statement with a COMMIT statement because data definition statements issue an automatic COMMIT before *and* after executing. So, whether they succeed or fail, the prior transaction is committed.

Using the SAVEPOINT Statement

You use the SAVEPOINT statement to mark and name the current point in the processing of a transaction. Each marked point is called a *savepoint*. For example, the following statement marks a savepoint named *start_delete*:

```
EXEC SQL SAVEPOINT start_delete;
```

Savepoints let you divide long transactions, giving you more control over complex procedures. For example, if a transaction performs several functions, you can mark a savepoint before each function. Then, if a function fails, you can easily restore the Oracle data to its former state, recover, then re–execute the function.

To undo part of a transaction, you use savepoints with the ROLLBACK statement and its TO SAVEPOINT clause. In the following example, you access the table MAIL_LIST to insert new listings, update old listings, and delete (a few) inactive listings. After the delete, you check the third element of *sqlerrd* in the SQLCA for the number of rows deleted. If the number is unexpectedly large, you roll back to the savepoint *start_delete*, undoing just the delete.

```
...
for (;;)
{
     printf(”Customer number? ”);
    gets(temp);
    cust_number = atoi(temp);
     printf(”Customer name? ”);
     gets(cust_name);
     EXEC SQL INSERT INTO mail_list (custno, cname, stat)
         VALUES (:cust_number, :cust_name, ’ACTIVE’);
...
}

for (;;)
{
    printf(”Customer number? ”);
    gets(temp);
    cust_number = atoi(temp);
     printf(”New status? ”);
     gets(new_status);
     EXEC SQL UPDATE mail_list
         SET stat = :new_status
         WHERE custno = :cust_number;
}
/* mark savepoint */
EXEC SQL SAVEPOINT start_delete;

EXEC SQL DELETE FROM mail_list
     WHERE stat = ’INACTIVE’;
if (sqlca.sqlerrd[2] < 25)  /* check number of rows deleted */
     printf(”Number of rows deleted is  %d\n”, sqlca.sqlerrd[2]);
else
{
     printf(”Undoing deletion of %d rows\n”, sqlca.sqlerrd[2]);
     EXEC SQL WHENEVER SQLERROR GOTO sql_error;
     EXEC SQL ROLLBACK TO SAVEPOINT start_delete;
}

EXEC SQL WHENEVER SQLERROR CONTINUE;
EXEC SQL COMMIT WORK RELEASE;
exit(0);
```

```
sql_error:
EXEC SQL WHENEVER SQLERROR CONTINUE;
EXEC SQL ROLLBACK WORK RELEASE;
printf(”Processing error\n”);
exit(1);
```

Rolling back to a savepoint erases any savepoints marked after that savepoint. The savepoint to which you roll back, however, is not erased. For example, if you mark five savepoints, then roll back to the third, only the fourth and fifth are erased.

If you give two savepoints the same name, the earlier savepoint is erased. A COMMIT or ROLLBACK statement erases all savepoints.

By default, the number of active savepoints per user session is limited to 5. An *active* savepoint is one that you marked since the last commit or rollback. Your Database Administrator (DBA) can raise the limit (up to 255) by increasing the value of the Oracle initialization parameter SAVEPOINTS.

Using the ROLLBACK Statement

You use the ROLLBACK statement to undo pending changes made to the database. For example, if you make a mistake, such as deleting the wrong row from a table, you can use ROLLBACK to restore the original data. The TO SAVEPOINT clause lets you roll back to an intermediate statement in the current transaction, so you do not have to undo all your changes.

If you start a transaction that you cannot finish (a SQL statement might not execute successfully, for example), ROLLBACK lets you return to the starting point, so that the database is not left in an inconsistent state. Specifically, the ROLLBACK statement

- undoes all changes made to the database during the current transaction
- erases all savepoints
- ends the transaction
- releases all row and table locks, but not parse locks
- closes cursors referenced in a CURRENT OF clause or, when MODE=ANSI, closes *all* explicit cursors

The ROLLBACK statement has no effect on the values of host variables or on the flow of control in your program.

When MODE=ORACLE, explicit cursors not referenced in a CURRENT OF clause remain open across ROLLBACKs.

Specifically, the ROLLBACK TO SAVEPOINT statement

- undoes changes made to the database since the specified savepoint was marked
- erases all savepoints marked after the specified savepoint
- releases all row and table locks acquired since the specified savepoint was marked

Note that you cannot specify the RELEASE option in a ROLLBACK TO SAVEPOINT statement.

Because they are part of exception processing, ROLLBACK statements should be placed in error handling routines, off the main path through your program. In the following example, you roll back your transaction and disconnect from Oracle:

```
EXEC SQL ROLLBACK WORK RELEASE;
```

The optional keyword WORK provides ANSI compatibility. The RELEASE option frees all resources held by your program and disconnects from the database.

If a WHENEVER SQLERROR GOTO statement branches to an error handling routine that includes a ROLLBACK statement, your program might enter an infinite loop if the ROLLBACK fails with an error. You can avoid this by coding WHENEVER SQLERROR CONTINUE before the ROLLBACK statement, as shown in the following example:

```
EXEC SQL WHENEVER SQLERROR GOTO sql_error;

for (;;)
{
     printf(”Employee number? ”);
    gets(temp);
    emp_number = atoi(temp);
     printf(”Employee name? ”);
     gets(emp_name);
     EXEC SQL INSERT INTO emp (empno, ename)
         VALUES (:emp_number, :emp_name);
...
}
...
sql_error:
EXEC SQL WHENEVER SQLERROR CONTINUE;
EXEC SQL ROLLBACK WORK RELEASE;
printf(”Processing error\n”);
exit(1);
```

Oracle automatically rolls back transactions if your program terminates abnormally. Refer to the section "Using the Release Option" below.

Statement–Level Rollbacks

Before executing any SQL statement, Oracle marks an implicit savepoint (not available to you). Then, if the statement fails, Oracle automatically rolls it back and returns the applicable error code to *sqlcode* in the SQLCA. For example, if an INSERT statement causes an error by trying to insert a duplicate value in a unique index, the statement is rolled back.

Oracle can also roll back single SQL statements to break deadlocks. Oracle signals an error to one of the participating transactions and rolls back the current statement in that transaction.

Only work started by the failed SQL statement is lost; work done before that statement in the current transaction is saved. Thus, if a data definition statement fails, the automatic commit that precedes it is not undone.

Before executing a SQL statement, Oracle must parse it, that is, examine it to make sure it follows syntax rules and refers to valid database objects. Errors detected while executing a SQL statement cause a rollback, but errors detected while parsing the statement do not.

Using the RELEASE Option

Oracle automatically rolls back changes if your program terminates abnormally. Abnormal termination occurs when your program does not explicitly commit or roll back work and disconnect from Oracle using the RELEASE option. Normal termination occurs when your program runs its course, closes open cursors, explicitly commits or rolls back work, disconnects from Oracle, and returns control to the user.

Your program will exit gracefully if the last SQL statement it executes is either

```
EXEC SQL COMMIT RELEASE;
```

or

```
EXEC SQL ROLLBACK RELEASE;
```

Otherwise, locks and cursors acquired by your user session are held after program termination until Oracle recognizes that the user session is no longer active. This might cause other users in a multiuser environment to wait longer than necessary for the locked resources.

Using the SET TRANSACTION Statement

You use the SET TRANSACTION statement to begin a read–only transaction. Because they allow "repeatable reads," read–only transactions are useful for running multiple queries against one or more tables while other users update the same tables. An example of the SET TRANSACTION statement follows:

```
EXEC SQL SET TRANSACTION READ ONLY;
```

The SET TRANSACTION statement must be the first SQL statement in a read–only transaction and can appear only once in a transaction. The READ ONLY parameter is required. Its use does not affect other transactions.

Only the SELECT, COMMIT, and ROLLBACK statements are allowed in a read–only transaction. For example, including an INSERT, DELETE, or SELECT FOR UPDATE OF statement causes an error.

During a read–only transaction, all queries refer to the same snapshot of the database, providing a multitable, multiquery, read–consistent view. Other users can continue to query or update data as usual.

A COMMIT, ROLLBACK, or data definition statement ends a read–only transaction. (Recall that data definition statements issue an implicit COMMIT.)

In the following example, as a store manager, you check sales activity for the day, the past week, and the past month by using a read–only transaction to generate a summary report. The report is unaffected by other users updating the database during the transaction.

```
EXEC SQL SET TRANSACTION READ ONLY;
EXEC SQL SELECT sum(saleamt) INTO :daily FROM sales
     WHERE saledate = SYSDATE;
EXEC SQL SELECT sum(saleamt) INTO :weekly FROM sales
     WHERE saledate > SYSDATE – 7;
EXEC SQL SELECT sum(saleamt) INTO :monthly FROM sales
     WHERE saledate > SYSDATE – 30;
EXEC SQL COMMIT WORK;
     /* simply ends the transaction since there are no changes
        to make permanent */
/* format and print report */
```

Overriding Default Locking

By default, Oracle implicitly (automatically) locks many data structures for you. However, you can request specific data locks on rows or tables when it is to your advantage to override default locking. Explicit locking lets you share or deny access to a table for the duration of a transaction or ensure multitable and multiquery read consistency.

With the SELECT FOR UPDATE OF statement, you can explicitly lock specific rows of a table to make sure they do not change before an UPDATE or DELETE is executed. However, Oracle automatically obtains row–level locks at UPDATE or DELETE time. So, use the FOR UPDATE OF clause only if you want to lock the rows *before* the UPDATE or DELETE.

You can explicitly lock entire tables using the LOCK TABLE statement.

Using FOR UPDATE OF

When you DECLARE a cursor that is referenced in the CURRENT OF clause of an UPDATE or DELETE statement, you use the FOR UPDATE OF clause to acquire exclusive row locks. SELECT FOR UPDATE OF identifies the rows that will be updated or deleted, then locks each row in the active set. This is useful, for example, when you want to base an update on the existing values in a row. You must make sure the row is not changed by another user before your update.

The FOR UPDATE OF clause is optional. For example, instead of coding

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
    SELECT ename, job, sal FROM emp WHERE deptno = 20
        FOR UPDATE OF sal;
```

you can drop the FOR UPDATE OF clause and simply code

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
    SELECT ename, job, sal FROM emp WHERE deptno = 20;
```

The CURRENT OF clause signals the precompiler to add a FOR UPDATE clause if necessary. You use the CURRENT OF clause to refer to the latest row FETCHed from a cursor. For an example, see page 4 – 17, “Using the CURRENT of Clause”.

Restrictions

If you use the FOR UPDATE OF clause, you cannot reference multiple tables.

An explicit FOR UPDATE OF or an implicit FOR UPDATE acquires exclusive row locks. All rows are locked at the OPEN, not as they are FETCHed. Row locks are released when you COMMIT or ROLLBACK (except when you ROLLBACK to a savepoint). Therefore, you cannot FETCH from a FOR UPDATE cursor after a COMMIT.

Using LOCK TABLE

You use the LOCK TABLE statement to lock one or more tables in a specified lock mode. For example, the statement below locks the EMP table in *row share* mode. Row share locks allow concurrent access to a table; they prevent other users from locking the entire table for exclusive use.

```
EXEC SQL LOCK TABLE EMP IN ROW SHARE MODE NOWAIT;
```

The lock mode determines what other locks can be placed on the table. For example, many users can acquire row share locks on a table at the same time, but only one user at a time can acquire an *exclusive* lock. While one user has an exclusive lock on a table, no other users can INSERT, UPDATE, or DELETE rows in that table.

For more information about lock modes, see the *Oracle7 Server Concepts*.

The optional keyword NOWAIT tells Oracle not to wait for a table if it has been locked by another user. Control is immediately returned to your program, so it can do other work before trying again to acquire the lock. (You can check *sqlcode* in the SQLCA to see if the LOCK TABLE failed.) If you omit NOWAIT, Oracle waits until the table is available; the wait has no set limit.

A table lock never keeps other users from querying a table, and a query never acquires a table lock. So, a query never blocks another query or an update, and an update never blocks a query. Only if two different transactions try to update the same row will one transaction wait for the other to complete.

Table locks are released when your transaction issues a COMMIT or ROLLBACK.

Fetching Across COMMITs

If you want to intermix COMMITs and FETCHes, do not use the CURRENT OF clause. Instead, SELECT the ROWID of each row, then use that value to identify the current row during the update or delete. An example follows:

```
...
EXEC SQL DECLARE emp_cursor CURSOR FOR
     SELECT ename, sal, ROWID FROM emp WHERE job = 'CLERK';
...
EXEC SQL OPEN emp_cursor;
EXEC SQL WHENEVER NOT FOUND GOTO ...
for (;;)
```

```
{
     EXEC SQL FETCH emp_cursor INTO :emp_name, :salary, :row_id;
     ...
     EXEC SQL UPDATE emp SET sal = :new_salary
         WHERE ROWID = :row_id;
     EXEC SQL COMMIT;
...
}
```

Note, however, that the FETCHed rows are *not* locked. So, you might get inconsistent results if another user modifies a row after you read it but before you update or delete it.

Handling Distributed Transactions

A *distributed database* is a single logical database comprising multiple physical databases at different nodes. A *distributed statement* is any SQL statement that accesses a remote node using a database link. A *distributed transaction* includes at least one distributed statement that updates data at multiple nodes of a distributed database. If the update affects only one node, the transaction is non–distributed.

When you issue a COMMIT, changes to each database affected by the distributed transaction are made permanent. If instead you issue a ROLLBACK, all the changes are undone. However, if a network or machine fails during the commit or rollback, the state of the distributed transaction might be unknown or *in doubt*. In such cases, if you have FORCE TRANSACTION system privileges, you can manually commit or roll back the transaction at your local database by using the FORCE clause. The transaction must be identified by a quoted literal containing the transaction ID, which can be found in the data dictionary view DBA_2PC_PENDING. Some examples follow:

```
EXEC SQL COMMIT FORCE ’22.31.83’;
...
EXEC SQL ROLLBACK FORCE ’25.33.86’;
```

FORCE commits or rolls back only the specified transaction and does not affect your current transaction. Note that you cannot manually roll back in–doubt transactions to a savepoint.

The COMMENT clause in the COMMIT statement lets you specify a comment to be associated with a distributed transaction. If ever the transaction is in doubt, Oracle stores the text specified by COMMENT in the data dictionary view DBA_2PC_PENDING along with the transaction ID. The text must be a quoted literal ≤ 50 characters in length. An example follows:

```
EXEC SQL COMMIT COMMENT 'In–doubt trans; notify Order Entry';
```

For more information about distributed transactions, see the *Oracle7 Server Concepts*.

Guidelines

The following guidelines will help you avoid some common problems.

Designing Applications

When designing your application, group logically related actions together in one transaction. A well–designed transaction includes all the steps necessary to accomplish a given task—no more and no less.

Data in the tables you reference must be left in a consistent state. So, the SQL statements in a transaction should change the data in a consistent way. For example, a transfer of funds between two bank accounts should include a debit to one account and a credit to another. Both updates should either succeed or fail together. An unrelated update, such as a new deposit to one account, should not be included in the transaction.

Obtaining Locks

If your application programs include SQL locking statements, make sure the Oracle users requesting locks have the privileges needed to obtain the locks. Your DBA can lock any table. Other users can lock tables they own or tables for which they have a privilege, such as ALTER, SELECT, INSERT, UPDATE, or DELETE.

Using PL/SQL

If a PL/SQL block is part of a transaction, COMMITs and ROLLBACKs inside the block affect the whole transaction. In the following example, the ROLLBACK undoes changes made by the UPDATE *and* the INSERT:

```
EXEC SQL INSERT INTO EMP ...
EXEC SQL EXECUTE
     BEGIN
         UPDATE emp ...
         ...
     EXCEPTION
         WHEN DUP_VAL_ON_INDEX THEN
             ROLLBACK;
         ...
     END;
END–EXEC;
...
```

CHAPTER

9

Handling Runtime Errors

An application program must anticipate runtime errors and attempt to recover from them. This chapter provides an in–depth discussion of error reporting and recovery. You learn how to handle errors and status changes using the SQLSTATE status variable, as well as the SQL Communications Area (SQLCA) and the WHENEVER statement. You also learn how to diagnose problems using the Oracle Communications Area (ORACA). The following topics are discussed:

- the need for error handling
- error handling alternatives
- using SQLSTATE
- declaring SQLCODE
- using the SQLCA
- using the WHENEVER statement
- using the ORACA

The Need for Error Handling

A significant part of every application program must be devoted to error handling. The main reason for error handling is that it allows your program to continue operating in the presence of errors. Errors arise from design faults, coding mistakes, hardware failures, invalid user input, and many other sources.

You cannot anticipate all possible errors, but you can plan to handle certain kinds of errors meaningful to your program. For the Pro*C Precompiler, error handling means detecting and recovering from SQL statement execution errors.

You can also prepare to handle warnings such as “value truncated” and status changes such as “end of data.”

It is especially important to check for error and warning conditions after every SQL data manipulation statement, because an INSERT, UPDATE, or DELETE statement might fail before processing all eligible rows in a table.

Error Handling Alternatives

There are several alternatives that you can use to detect errors and status changes in the application. This chapter describes these alternatives, however, no specific recommendations are made about what method you should use. The method is, after all, dictated by the design of the application program or tool that you are building.

Status Variables

You can declare a separate status variable, SQLSTATE or SQLCODE, examine its value after each executable SQL statement, and take appropriate action. The action might be calling an error–reporting function, then exiting the program if the error is unrecoverable. Or, you might be able to adjust data, or control variables, and retry the action. See the sections “The SQLSTATE Status Variable” on page 9 – 4 and the “Declaring SQLCODE” on page 9 – 13 in this chapter for complete information about these status variables.

The SQL Communications Area

Another alternative that you can use is to include the SQL Communications Area structure (*sqlca*) in your program. This structure contains components that are filled in at runtime after the SQL statement is processed by Oracle.

> **Note:** In this guide, the *sqlca* structure is commonly referred to using the acronym for *SQL Communications Area* (SQLCA). When this guide refers to a specific component in the C **struct**, the structure name (*sqlca*) is used.

The SQLCA is defined in the header file *sqlca.h*, which you include in your program using either of the following statements:

- EXEC SQL INCLUDE SQLCA;
- #include <sqlca.h>

Oracle updates the SQLCA after every *executable* SQL statement. (SQLCA values are unchanged after a declarative statement.) By checking Oracle return codes stored in the SQLCA, your program can determine the outcome of a SQL statement. This can be done in the following two ways:

- implicit checking with the WHENEVER statement
- explicit checking of SQLCA components

You can use WHENEVER statements, code explicit checks on SQLCA components, or do both.

The most frequently–used components in the SQLCA are the status variable (*sqlca.sqlcode*), and the text associated with the error code (*sqlca.sqlerrm.sqlerrmc*). Other components contain warning flags and miscellaneous information about the processing of the SQL statement. For complete information about the SQLCA structure, see the "Using the SQL Communications Area" section on page 9 – 15.

> **Note:** SQLCODE (upper case) always refers to a separate status variable, not a component of the SQLCA. SQLCODE is declared as a **long** integer. When referring to the component of the SQLCA named *sqlcode*, the fully–qualified name *sqlca.sqlcode* is always used.

When more information is needed about runtime errors than the SQLCA provides, you can use the ORACA. The ORACA is a C **struct** that handles Oracle communication. It contains cursor statistics, information about the current SQL statement, option settings, and system statistics. See the "Using the Oracle Communications Area" section on page 9 – 32 for complete information about the ORACA.

The SQLSTATE Status Variable

The precompiler command line option MODE governs ANSI/ISO compliance. When MODE=ANSI, declaring the SQLCA data structure is optional. However, you must declare a separate status variable named SQLCODE. SQL92 specifies a similar status variable named SQLSTATE, which you can use with or without SQLCODE.

After executing a SQL statement, the Oracle Server returns a status code to the SQLSTATE variable currently in scope. The status code indicates whether the SQL statement executed successfully or raised an exception (error or warning condition). To promote *interoperability* (the ability of systems to exchange information easily), SQL92 predefines all the common SQL exceptions.

Unlike SQLCODE, which stores only error codes, SQLSTATE stores error and warning codes. Furthermore, the SQLSTATE reporting mechanism uses a standardized coding scheme. Thus, SQLSTATE is the preferred status variable. Under SQL92, SQLCODE is a “deprecated feature” retained only for compatibility with SQL89 and likely to be removed from future versions of the standard.

Declaring SQLSTATE

When MODE=ANSI, you must declare SQLSTATE or SQLCODE. Declaring the SQLCA is optional. When MODE=ORACLE, if you declare SQLSTATE, it is not used.

Unlike SQLCODE, which stores signed integers and can be declared outside the Declare Section, SQLSTATE stores 5–character null–terminated strings and must be declared inside the Declare Section. You declare SQLSTATE as

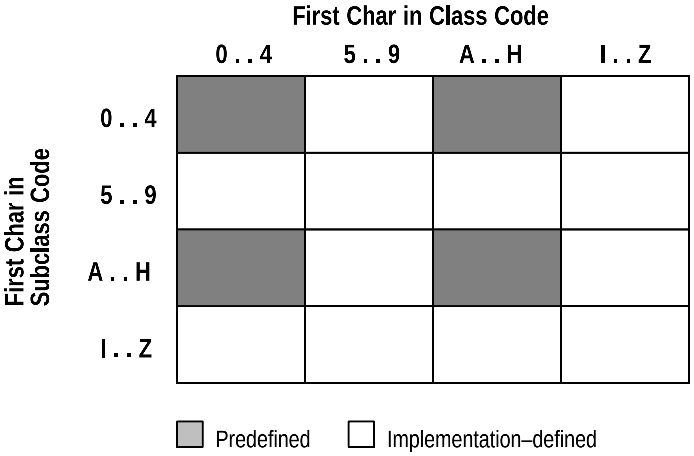
```
char  SQLSTATE[6];  /* Upper case is required. */
```

Note: SQLSTATE must be declared with a dimension of *exactly* 6 characters.

SQLSTATE Values

SQLSTATE status codes consist of a 2–character *class code* followed by a 3–character *subclass code*. Aside from class code 00 (“successful completion”), the class code denotes a category of exceptions. And, aside from subclass code 000 (“not applicable”), the subclass code denotes a specific exception within that category. For example, the SQLSTATE value ‘22012’ consists of class code 22 (“data exception”) and subclass code 012 (“division by zero”).

Each of the five characters in a SQLSTATE value is a digit (0..9) or an uppercase Latin letter (A..Z). Class codes that begin with a digit in the range 0..4 or a letter in the range A..H are reserved for predefined conditions (those defined in SQL92). All other class codes are reserved for implementation–defined conditions. Within predefined classes, subclass codes that begin with a digit in the range 0..4 or a letter in the range A..H are reserved for predefined subconditions. All other subclass codes are reserved for implementation–defined subconditions. Figure 9 – 1 shows the coding scheme.



First Char in Class Code

First Char in Subclass Code

	0..4	5..9	A..H	I..Z
0..4	▨	□	▨	□
5..9	□	□	□	□
A..H	▨	□	▨	□
I..Z	□	□	□	□

▨ Predefined □ Implementation–defined

Figure 9 – 1 SQLSTATE Coding Scheme

Table 9 – 1 shows the classes predefined by SQL92.

Class	Condition
00	success completion
01	warning
02	no data
07	dynamic SQL error

Table 9 – 1 Predefined Classes

Class	Condition
08	connection exception
0A	feature not supported
21	cardinality violation
22	data exception
23	integrity constraint violation
24	invalid cursor state
25	invalid transaction state
26	invalid SQL statement name
27	triggered data change violation
28	invalid authorization specification
2A	direct SQL syntax error or access rule violation
2B	dependent privilege descriptors still exist
2C	invalid character set name
2D	invalid transaction termination
2E	invalid connection name
33	invalid SQL descriptor name
34	invalid cursor name
35	invalid condition number
37	dynamic SQL syntax error or access rule violation
3C	ambiguous cursor name
3D	invalid catalog name
3F	invalid schema name
40	transaction rollback
42	syntax error or access rule violation
44	with check option violation
HZ	remote database access

Table 9 – 1 Predefined Classes

Note: The class code HZ is reserved for conditions defined in International Standard ISO / IEC DIS 9579–2, *Remote Database Access.*

Table 9 – 2 shows how SQLSTATE status codes and conditions are mapped to Oracle errors. Status codes in the range 60000 .. 99999 are implementation–defined.

Code	Condition	OracleError(s)
00000	successful completion	ORA–00000
01000	warning	
01001	cursor operation conflict	
01002	disconnect error	
01003	null value eliminated in set function	
01004	string data–right truncation	
01005	insufficient item descriptor areas	
01006	privilege not revoked	
01007	privilege not granted	
01008	implicit zero–bit padding	
01009	search condition too long for info schema	
0100A	query expression too long for info schema	
02000	no data	ORA–01095 ORA–01403
07000	dynamic SQL error	
07001	using clause does not match parameter specs	
07002	using clause does not match target specs	
07003	cursor specification cannot be executed	
07004	using clause required for dynamic parameters	
07005	prepared statement not a cursor specification	
07006	restricted datatype attribute violation	
07007	using clause required for result components invalid descriptor count	
07008	invalid descriptor count	SQL–02126
07009	invalid descriptor index	
08000	connection exception	
08001	SQL–client unable to establish SQL–connection	
08002	connection name is use	
08003	connection does not exist	SQL–02121
08004	SQL–server rejected SQL–connection	

Table 9 – 2 SQLSTATE Status Codes

Code	Condition	OracleError(s)
08006	connection failure	
08007	transaction resolution unknown	
0A000	feature not supported	ORA–03000..03099
0A001	multiple server transactions	
21000	cardinality violation	ORA–01427 SQL–02112
22000	data exception	
22001	string data – right truncation	ORA–01406
22002	null value–no indicator parameter	SQL–02124
22003	numeric value out of range	ORA–01426
22005	error in assignment	
22007	invalid datetime format	
22008	datetime field overflow	ORA–01800..01899
22009	invalid time zone displacement value	
22011	substring error	
22012	division by zero	ORA–01476
22015	interval field overflow	
22018	invalid character value for cast	
22019	invalid escape character	ORA–00911
22021	character not in repertoire	
22022	indicator overflow	ORA–01411
22023	invalid parameter value	ORA–01025 ORA–04000..04019
22024	unterminated C string	ORA–01479 ORA–01480
22025	invalid escape sequence	ORA–01424 ORA–01425
22026	string data–length mismatch	ORA–01401
22027	trim error	
23000	integrity constraint violation	ORA–02290..02299
24000	invalid cursor state	ORA–001002 ORA–001003 SQL–02114 SQL–02117
25000	invalid transaction state	SQL–02118

Table 9 – 2 SQLSTATE Status Codes

Code	Condition	OracleError(s)
26000	invalid SQL statement name	
27000	triggered data change violation	
28000	invalid authorization specification	
2A000	direct SQL syntax error or access rule violation	
2B000	dependent privilege descriptors still exist	
2C000	invalid character set name	
2D000	invalid transaction termination	
2E000	invalid connection name	
33000	invalid SQL descriptor name	
34000	invalid cursor name	
35000	invalid condition number	
37000	dynamic SQL syntax error or access rule violation	
3C000	ambiguous cursor name	
3D000	invalid catalog name	
3F000	invalid schema name	
40000	transaction rollback	ORA–02091 ORA–02092
40001	serialization failure	
40002	integrity constraint violation	
40003	statement completion unknown	
42000	syntax error or access rule violation	ORA–00022 ORA–00251 ORA–00900..00999 ORA–01031 ORA–01490..01493 ORA–01700..01799 ORA–01900..02099 ORA–02140..02289 ORA–02420..02424 ORA–02450..02499 ORA–03276..03299 ORA–04040..04059 ORA–04070..04099
44000	with check option violation	ORA–01402

Table 9 – 2 SQLSTATE Status Codes

Code	Condition	OracleError(s)
60000	system error	ORA–00370..00429 ORA–00600..00899 ORA–06430..06449 ORA–07200..07999 ORA–09700..09999
61000	multi–threaded server and detached process errors	ORA–00018..00035 ORA–00050..00068 ORA–02376..02399 ORA–04020..04039
62000	multi–threaded server and detached process errors	ORA–00100..00120 ORA–00440..00569
63000	Oracle*XA and two–task interface errors	ORA–00150..00159 ORA–02700..02899 ORA–03100..03199 ORA–06200..06249 SQL–02128
64000	control file, database file, and redo file errors; archival and media recovery errors	ORA–00200..00369 ORA–01100..01250
65000	PL/SQL errors	ORA–06500..06599
66000	SQL*Net driver errors	ORA–06000..06149 ORA–06250..06429 ORA–06600..06999 ORA–12100..12299 ORA–12500..12599
67000	licensing errors	ORA–00430..00439
69000	SQL*Connect errors	ORA–00570..00599 ORA–07000..07199
72000	SQL execute phase errors	ORA–00001 ORA–01000..01099 ORA–01400..01489 ORA–01495..01499 ORA–01500..01699 ORA–02400..02419 ORA–02425..02449 ORA–04060..04069 ORA–08000..08190 ORA–12000..12019 ORA–12300..12499 ORA–12700..21999
82100	out of memory (could not allocate)	SQL–02100
82101	inconsistent cursor cache (UCE/CUC mismatch)	SQL–02101
82102	inconsistent cursor cache (no CUC entry for UCE)	SQL–02102

Table 9 – 2 SQLSTATE Status Codes

Code	Condition	OracleError(s)
82103	inconsistent cursor cache (out–or–range CUC ref)	SQL–02103
82104	inconsistent cursor cache (no CUC available)	SQL–02104
82105	inconsistent cursor cache (no CUC entry in cache)	SQL–02105
82106	inconsistent cursor cache (invalid cursor number)	SQL–02106
82107	program too old for runtime library; re–precompile	SQL–02107
82108	invalid descriptor passed to runtime library	SQL–02108
82109	inconsistent host cache (out–or–range SIT ref)	SQL–02109
82110	inconsistent host cache (invalid SQL type)	SQL–02110
82111	heap consistency error	SQL–02111
82113	code generation internal consistency failed	SQL–02115
82114	reentrant code generator gave invalid context	SQL–02116
82117	invalid OPEN or PREPARE for this connection	SQL–02122
82118	application context not found	SQL–02123
82119	unable to obtain error message text	SQL–02125
82120	Precompiler/SQLLIB version mismatch	SQL–02127
82121	NCHAR error; fetched number of bytes is odd	SQL–02129
82122	EXEC TOOLS interface not available	SQL–02130
82123	runtime context in use	SQL–02131
82124	unable to allocate runtime context	SQL–02132
82125	unable to initialize process for use with threads	SQL–02133
82126	invalid runtime context	SQL–02134
HZ000	remote database access	

Table 9 – 2 SQLSTATE Status Codes

Using SQLSTATE

The following rules apply to using SQLSTATE with SQLCODE or the SQLCA when you precompile with the option setting MODE=ANSI. SQLSTATE must be declared inside a Declare Section; otherwise, it is ignored.

If you declare SQLSTATE

- Declaring SQLCODE is optional. If you declare SQLCODE inside the Declare Section, the Oracle Server returns status codes to SQLSTATE and SQLCODE after every SQL operation. However, if you declare SQLCODE outside the Declare Section, Oracle returns a status code only to SQLSTATE.
- Declaring the SQLCA is optional. If you declare the SQLCA, Oracle returns status codes to SQLSTATE and the SQLCA. In this case, to avoid compilation errors, do *not* declare SQLCODE.

If you do *not* declare SQLSTATE

- You must declare SQLCODE inside or outside the Declare Section. The Oracle Server returns a status code to SQLCODE after every SQL operation.
- Declaring the SQLCA is optional. If you declare the SQLCA, Oracle returns status codes to SQLCODE and the SQLCA.

You can learn the outcome of the most recent executable SQL statement by checking SQLSTATE explicitly with your own code or implicitly with the WHENEVER SQLERROR statement. Check SQLSTATE only after executable SQL statements and PL/SQL statements.

Declaring SQLCODE

When MODE=ANSI, and you have not declared a SQLSTATE status variable, you must declare a **long** integer variable named SQLCODE inside or outside the Declare Section. An example follows:

```
/* declare host variables */
EXEC SQL BEGIN DECLARE SECTION;
int  emp_number, dept_number;
char emp_name[20];
EXEC SQL END DECLARE SECTION;

/* declare status variable––must be upper case */
long SQLCODE;
```

When MODE=ORACLE, if you declare SQLCODE, it is not used.

You can declare more than one SQLCODE. Access to a local SQLCODE is limited by its scope within your program.

After every SQL operation, Oracle returns a status code to the SQLCODE currently in scope. So, your program can learn the outcome of the most recent SQL operation by checking SQLCODE explicitly, or implicitly with the WHENEVER statement.

When you declare SQLCODE instead of the SQLCA in a particular compilation unit, the precompiler allocates an internal SQLCA for that unit. Your host program cannot access the internal SQLCA. If you declare the SQLCA *and* SQLCODE, Oracle returns the same status code to both after every SQL operation.

Key Components of Error Reporting Using the SQLCA

Error reporting depends on variables in the SQLCA. This section highlights the key components of error reporting. The next section takes a close look at the SQLCA.

Status Codes

Every executable SQL statement returns a status code to the SQLCA variable *sqlcode,* which you can check implicitly with the WHENEVER statement or explicitly with your own code.

A zero status code means that Oracle executed the statement without detecting an error or exception. A positive status code means that Oracle executed the statement but detected an exception. A negative status code means that Oracle did not execute the SQL statement because of an error.

Warning Flags

Warning flags are returned in the SQLCA variables *sqlwarn[0]* through *sqlwarn[7],* which you can check implicitly or explicitly. These warning flags are useful for runtime conditions not considered errors by Oracle. For example, when DBMS=V6, if an indicator variable is available, Oracle signals a warning after assigning a truncated column value to a host variable. (If no indicator variable is available, Oracle issues an error message.)

Rows–Processed Count

The number of rows processed by the most recently executed SQL statement is returned in the SQLCA variable *sqlca.sqlerrd[2],* which you can check explicitly.

Strictly speaking, this variable is not for error reporting, but it can help you avoid mistakes. For example, suppose you expect to delete about ten rows from a table. After the deletion, you check *sqlca.sqlerrd[2]* and find that 75 rows were processed. To be safe, you might want to roll back the deletion and examine your WHERE–clause search condition.

Parse Error Offset

Before executing a SQL statement, Oracle must *parse* it, that is, examine it to make sure it follows syntax rules and refers to valid database objects. If Oracle finds an error, an offset is stored in the SQLCA variable *sqlca.sqlerrd[4],* which you can check explicitly. The offset specifies the character position in the SQL statement at which the parse error begins. As in a normal C string, the first character occupies position zero. For example, if the offset is 9, the parse error begins at the 10*th* character.

By default, static SQL statements are checked for syntactic errors at precompile time. So, *sqlca.sqlerrd[4]* is most useful for debugging dynamic SQL statements, which your program accepts or builds at run time.

Parse errors arise from missing, misplaced, or misspelled keywords, invalid options, nonexistent tables, and the like. For example, the dynamic SQL statement

```
"UPDATE emp SET jib = :job_title WHERE empno = :emp_number"
```

causes the parse error

```
ORA–00904: invalid column name
```

because the column name JOB is misspelled. The value of *sqlca.sqlerrd[4]* is 15 because the erroneous column name JIB begins at the 16*th* character.

If your SQL statement does not cause a parse error, Oracle sets *sqlca.sqlerrd[4]* to zero. Oracle also sets *sqlca.sqlerrd[4]* to zero if a parse error begins at the first character (which occupies position zero). So, check *sqlca.sqlerrd[4]* only if *sqlca.sqlcode* is negative, which means that an error has occurred.

Error Message Text

The error code and message for Oracle errors are available in the SQLCA variable SQLERRMC. At most, the first 70 characters of text are stored. To get the full text of messages longer than 70 characters, you use the *sqlglm()* function. See the section "Getting the Full Text of Error Messages" on page 9 – 22.

Using the SQL Communications Area (SQLCA)

The SQLCA is a data structure. Its components contain error, warning, and status information updated by Oracle whenever a SQL statement is executed. Thus, the SQLCA always reflects the outcome of the most recent SQL operation. To determine the outcome, you can check variables in the SQLCA.

Your program can have more than one SQLCA. For example, it might have one global SQLCA and several local ones. Access to a local SQLCA is limited by its scope within the program. Oracle returns information only to the SQLCA that is in scope.

> **Note:** When your application uses SQL*Net to access a combination of local and remote databases concurrently, all the databases write to one SQLCA. There is *not* a different SQLCA for each database. For more information, see the section "Concurrent Connections" on page 3 – 88.

Declaring the SQLCA

When MODE=ORACLE, declaring the SQLCA is required. To declare the SQLCA, you should copy it into your program with the INCLUDE or **#include** statement, as follows:

```
EXEC SQL INCLUDE SQLCA;
```

or

```
#include <sqlca.h>
```

If you use a Declare Section, the SQLCA must be declared *outside* the Declare Section. Not declaring the SQLCA results in compile–time errors.

When you precompile your program, the INCLUDE SQLCA statement is replaced by several variable declarations that allow Oracle to communicate with the program.

When MODE=ANSI, declaring the SQLCA is optional. But in this case you must declare a SQLCODE or SQLSTATE status variable. The type of SQLCODE (upper case is required) is **long**. If you declare SQLCODE or SQLSTATE instead of the SQLCA in a particular compilation unit, the precompiler allocates an internal SQLCA for that unit. Your Pro*C program cannot access the internal SQLCA. If you declare the SQLCA *and* SQLCODE, Oracle returns the same status code to both after every SQL operation.

> **Note:** Declaring the SQLCA is optional when MODE=ANSI, but you cannot use the WHENEVER SQLWARNING statement without the SQLCA. So, if you want to use the WHENEVER SQLWARNING statement, you must declare the SQLCA.

> **Note:** This Guide uses SQLCODE when referring to the SQLCODE status variable, and *sqlca.sqlcode* when explicitly referring to the component of the SQLCA structure.

What’s in the SQLCA?

The SQLCA contains the following runtime information about the outcome of SQL statements:

- Oracle error codes
- warning flags
- event information
- rows–processed count
- diagnostics

The *sqlca.h* header file is:

```
/*
NAME
  SQLCA : SQL Communications Area.
FUNCTION
  Contains no code. Oracle fills in the SQLCA with status info
  during the execution of a SQL stmt.
NOTES
  **************************************************************
  ***                                                        ***
  *** This file is SOSD.  Porters must change the data types ***
  *** appropriately on their platform.  See notes/pcport.doc ***
  *** for more information.                                  ***
  ***                                                        ***
  **************************************************************

  If the symbol SQLCA_STORAGE_CLASS is defined, then the SQLCA
  will be defined to have this storage class. For example:

    #define SQLCA_STORAGE_CLASS extern

  will define the SQLCA as an extern.

  If the symbol SQLCA_INIT is defined, then the SQLCA will be
  statically initialized. Although this is not necessary in order
  to use the SQLCA, it is a good programing practice not to have
  unitialized variables. However, some C compilers/OS’s don’t
  allow automatic variables to be initialized in this manner.
  Therefore, if you are INCLUDE’ing the SQLCA in a place where it
  would be an automatic AND your C compiler/OS doesn’t allow this
  style of initialization, then SQLCA_INIT should be left
  undefined –– all others can define SQLCA_INIT if they wish.

  If the symbol SQLCA_NONE is defined, then the SQLCA
  variable will not be defined at all.  The symbol SQLCA_NONE
  should not be defined in source modules that have embedded SQL.
  However, source modules that have no embedded SQL, but need to
  manipulate a sqlca struct passed in as a parameter, can set the
  SQLCA_NONE symbol to avoid creation of an extraneous sqlca
  variable.
*/
#ifndef SQLCA
#define SQLCA 1
struct   sqlca
         {
         /* ub1 */ char    sqlcaid[8];
         /* b4  */ long    sqlabc;
         /* b4  */ long    sqlcode;
         struct
           {
           /* ub2 */ unsigned short sqlerrml;
           /* ub1 */ char           sqlerrmc[70];
           } sqlerrm;
         /* ub1 */ char    sqlerrp[8];
         /* b4  */ long    sqlerrd[6];
         /* ub1 */ char    sqlwarn[8];
         /* ub1 */ char    sqlext[8];
         };
#ifndef SQLCA_NONE
#ifdef   SQLCA_STORAGE_CLASS
SQLCA_STORAGE_CLASS struct sqlca sqlca
#else
         struct sqlca sqlca
#endif
#ifdef  SQLCA_INIT
         = {
         {'S', 'Q', 'L', 'C', 'A', ' ', ' ', ' '},
         sizeof(struct sqlca),
         0,
         { 0, {0}},
         {'N', 'O', 'T', ' ', 'S', 'E', 'T', ' '},
         {0, 0, 0, 0, 0, 0},
         {0, 0, 0, 0, 0, 0, 0, 0},
         {0, 0, 0, 0, 0, 0, 0, 0}
         }
#endif
         ;
#endif
#endif
```

Structure of the SQLCA

This section describes the structure of the SQLCA, its components, and the values they can store.

sqlcaid

This string component is initialized to “SQLCA” to identify the SQL Communications Area.

sqlcabc

This integer component holds the length, in bytes, of the SQLCA structure.

sqlcode

This integer component holds the status code of the most recently executed SQL statement. The status code, which indicates the outcome of the SQL operation, can be any of the following numbers:

0

Means that Oracle executed the statement without detecting an error or exception.

>0

Means that Oracle executed the statement but detected an exception. This occurs when Oracle cannot find a row that meets your WHERE–clause search condition or when a SELECT INTO or FETCH returns no rows.

When MODE=ANSI, +100 is returned to *sqlcode* after an INSERT of no rows. This can happen when a subquery returns no rows to process.

<0

Means that Oracle did not execute the statement because of a database, system, network, or application error. Such errors can be fatal. When they occur, the current transaction should, in most cases, be rolled back.

Negative return codes correspond to error codes listed in the *Oracle7 Server Messages* manual.

sqlerrm

This embedded struct contains the following two components:

sqlerrml

This integer component holds the length of the message text stored in *sqlerrmc.*

sqlerrmc

This string component holds the message text corresponding to the error code stored in *sqlcode.* The string is *not* null terminated. Use the *sqlerrml* component to determine the length.

This component can store up to 70 characters. To get the full text of messages longer than 70 characters, you must use the *sqlglm* function (discussed later).

Make sure *sqlcode* is negative *before* you reference *sqlerrmc*. If you reference *sqlerrmc* when *sqlcode* is zero, you get the message text associated with a prior SQL statement.

sqlerrp

This string component is reserved for future use.

sqlerrd

This array of binary integers has six elements. Descriptions of the components in *sqlerrd* follow:

sqlerrd[0] This component is reserved for future use.

sqlerrd[1] This component is reserved for future use.

sqlerrd[2] This component holds the number of rows processed by the most recently executed SQL statement. However, if the SQL statement failed, the value of *sqlca.sqlerrd[2]* is undefined, with one exception. If the error occurred during an array operation, processing stops at the row that caused the error, so *sqlca.sqlerrd[2]* gives the number of rows processed successfully.

The rows–processed count is zeroed after an OPEN statement and incremented after a FETCH statement. For the EXECUTE, INSERT, UPDATE, DELETE, and SELECT INTO statements, the count reflects the number of rows processed successfully. The count does *not* include rows processed by an UPDATE or DELETE CASCADE. For example, if 20 rows are deleted because they meet WHERE–clause criteria, and 5 more rows are deleted because they now (after the primary delete) violate column constraints, the count is 20 not 25.

sqlerrd[3] This component is reserved for future use.

sqlerrd[4] This component holds an offset that specifies the character position at which a parse error begins in the most recently executed SQL statement. The first character occupies position zero.

sqlerrd[5] This component is reserved for future use.

sqlwarn

This array of single characters has eight elements. They are used as warning flags. Oracle sets a flag by assigning it a “W” (for warning) character value.

The flags warn of exceptional conditions. For example, a warning flag is set when Oracle assigns a truncated column value to an output host variable.

Descriptions of the components in *sqlwarn* follow:

sqlwarn[0]

This flag is set if another warning flag is set.

sqlwarn[1]

This flag is set if a truncated column value was assigned to an output host variable. This applies only to character data. Oracle truncates certain numeric data without setting a warning or returning a negative *sqlcode.*

To find out if a column value was truncated and by how much, check the indicator variable associated with the output host variable. The (positive) integer returned by an indicator variable is the original length of the column value. You can increase the length of the host variable accordingly.

sqlwarn[2]

This flag is set if a NULL column is not used in the result of a SQL group function, such as AVG() or SUM().

sqlwarn[3]

This flag is set if the number of columns in a query select list does not equal the number of host variables in the INTO clause of the SELECT or FETCH statement. The number of items returned is the lesser of the two.

sqlwarn[4]

This flag is set if every row in a table was processed by an UPDATE or DELETE statement without a WHERE clause. An update or deletion is called *unconditional* if no search condition restricts the number of rows processed. Such updates and deletions are unusual, so Oracle sets this warning flag. That way, you can roll back the transaction if necessary.

sqlwarn[5] This flag is set when an EXEC SQL CREATE {PROCEDURE | FUNCTION | PACKAGE | PACKAGE BODY} statement fails because of a PL/SQL compilation error.

sqlwarn[6] This flag is no longer in use.

sqlwarn[7] This flag is no longer in use.

sqlext This string component is reserved for future use.

PL/SQL Considerations

When the precompiler application executes an embedded PL/SQL block, not all components of the SQLCA are set. For example, if the block fetches several rows, the rows–processed count (*sqlerrd[2]*) is set to only 1. You should depend only on the *sqlcode* and *sqlerrm* components of the SQLCA after execution of a PL/SQL block.

Getting the Full Text of Error Messages

The SQLCA can accommodate error messages up to 70 characters long. To get the full text of longer (or nested) error messages, you need the *sqlglm* function. The syntax of the *sqlglm()* is

```
void sqlglm(char   *message_buffer,
            size_t *buffer_size,
            size_t *message_length);
```

where:

message_buffer Is the text buffer in which you want Oracle to store the error message (Oracle blank–pads to the end of this buffer).

buffer_size Is a scalar variable that specifies the maximum size of the buffer in bytes.

message_length Is a scalar variable in which Oracle stores the actual length of the error message.

Note: The types of the last two arguments for the *sqlglm()* function are shown here generically as **size_t** pointers. However on your platform they might have a different type. For example, on many UNIX workstation ports, they are **unsigned int *.**

You should check the file *sqlcpr.h*, which is in the standard include directory on your system, to determine the datatype of these parameters.

The maximum length of an Oracle error message is 512 characters including the error code, nested messages, and message inserts such as table and column names. The maximum length of an error message returned by *sqlglm* depends on the value you specify for *buffer_size*.

The following example calls *sqlglm* to get an error message of up to 200 characters in length:

```
EXEC SQL WHENEVER SQLERROR DO sql_error();
...
/* other statements */
...
sql_error()
{
    char msg[200];
    size_t buf_len, msg_len;

    buf_len = sizeof (msg);
    sqlglm(msg, &buf_len, &msg_len);   /* note use of pointers */
    printf(”%.*s\n\n”, msg_len, msg);
    exit(1);
}
```

Notice that *sqlglm* is called only when a SQL error has occurred. Always make sure SQLCODE (or *sqlca.sqlcode*) is non–zero *before* calling *sqlglm*. If you call *sqlglm* when SQLCODE is zero, you get the message text associated with a prior SQL statement.

Using the WHENEVER Statement

By default, precompiled programs ignore Oracle error and warning conditions and continue processing if possible. To do automatic condition checking and error handling, you need the WHENEVER statement.

With the WHENEVER statement you can specify actions to be taken when Oracle detects an error, warning condition, or “not found” condition. These actions include continuing with the next statement, calling a routine, branching to a labeled statement, or stopping.

You code the WHENEVER statement using the following syntax:

```
EXEC SQL WHENEVER <condition> <action>;
```

Conditions

You can have Oracle automatically check the SQLCA for any of the following conditions.

SQLWARNING

sqlwarn[0] is set because Oracle returned a warning (one of the warning flags, *sqlwarn[1]* through *sqlwarn[7]*, is also set) or SQLCODE has a positive value other than +1403. For example, *sqlwarn[0]* is set when Oracle assigns a truncated column value to an output host variable.

Declaring the SQLCA is optional when MODE=ANSI. To use WHENEVER SQLWARNING, however, you *must* declare the SQLCA.

SQLERROR

SQLCODE has a negative value because Oracle returned an error.

NOT FOUND

SQLCODE has a value of +1403 (+100 when MODE=ANSI) because Oracle could not find a row that meets your WHERE–clause search condition, or a SELECT INTO or FETCH returned no rows.

When MODE=ANSI, +100 is returned to SQLCODE after an INSERT of no rows.

Actions

When Oracle detects one of the preceding *conditions*, you can have your program take any of the following actions.

CONTINUE

Your program continues to run with the next statement if possible. This is the default action, equivalent to not using the WHENEVER statement. You can use it to turn off condition checking.

DO

Your program transfers control to an error handling function in the program. When the end of the routine is reached, control transfers to the statement that follows the failed SQL statement.

The usual rules for entering and exiting a function apply. You can pass parameters to the error handler invoked by an EXEC SQL WHENEVER ... DO ... statement, and the function can return a value.

GOTO label_name

Your program branches to a labeled statement.

STOP

Your program stops running and uncommitted work is rolled back.

STOP in effect just generates an *exit()* call whenever the condition occurs. Be careful. The STOP action displays no messages before disconnecting from Oracle.

Some Examples

If you want your program to

- go to *close_cursor* if a “no data found” condition occurs,
- continue with the next statement if a warning occurs, and
- go to *error_handler* if an error occurs

simply code the following WHENEVER statements before the first executable SQL statement:

```
EXEC SQL WHENEVER NOT FOUND GOTO close_cursor;
EXEC SQL WHENEVER SQLWARNING CONTINUE;
EXEC SQL WHENEVER SQLERROR GOTO error_handler;
```

In the following example, you use WHENEVER...DO statements to handle specific errors:

```
...
EXEC SQL WHENEVER SQLERROR DO handle_insert_error(”INSERT error”);
EXEC SQL INSERT INTO emp (empno, ename, deptno)
     VALUES (:emp_number, :emp_name, :dept_number);
EXEC SQL WHENEVER SQLERROR DO handle_delete_error(”DELETE error”);
EXEC SQL DELETE FROM dept WHERE deptno = :dept_number;
...
handle_insert_error(char *stmt)
{   switch(sqlca.sqlcode)
    {
    case –1:
    /* duplicate key value */
        ...
        break;
    case –1401:
    /* value too large */
        ...
        break;
    default:
    /* do something here too */
        ...
        break;
    }
}
```

```
handle_delete_error(char *stmt)
{
    printf(”%s\n\n”, stmt);
    if (sqlca.sqlerrd[2] == 0)
    {
        /* no rows deleted */
        ...
    }
    else
    {   ...
    }
    ...
}
```

Notice how the procedures check variables in the SQLCA to determine a course of action.

Scope of WHENEVER

Because WHENEVER is a declarative statement, its scope is positional, not logical. That is, it tests all executable SQL statements that *physically* follow it in the source file, not in the flow of program logic. So, code the WHENEVER statement before the first executable SQL statement you want to test.

A WHENEVER statement stays in effect until superseded by another WHENEVER statement checking for the same condition.

In the example below, the first WHENEVER SQLERROR statement is superseded by a second, and so applies only to the CONNECT statement. The second WHENEVER SQLERROR statement applies to both the UPDATE and DROP statements, despite the flow of control from *step1* to *step3*.

```
step1:
     EXEC SQL WHENEVER SQLERROR STOP;
     EXEC SQL CONNECT :username IDENTIFIED BY :password;
     ...
     goto step3;
step2:
     EXEC SQL WHENEVER SQLERROR CONTINUE;
     EXEC SQL UPDATE emp SET sal = sal * 1.10;
     ...
step3:
     EXEC SQL DROP INDEX emp_index;
     ...
```

Guidelines

The following guidelines will help you avoid some common pitfalls.

Placing the Statements

In general, code a WHENEVER statement before the first executable SQL statement in your program. This ensures that all ensuing errors are trapped because WHENEVER statements stay in effect to the end of a file.

Handling End–of–Data Conditions

Your program should be prepared to handle an end–of–data condition when using a cursor to fetch rows. If a FETCH returns no data, the program should exit the fetch loop, as follows:

```
EXEC SQL WHENEVER NOT FOUND DO break;
for (;;)
{
    EXEC SQL FETCH...
}
EXEC SQL CLOSE my_cursor;
...
```

Avoiding Infinite Loops

If a WHENEVER SQLERROR GOTO statement branches to an error handling routine that includes an executable SQL statement, your program might enter an infinite loop if the SQL statement fails with an error. You can avoid this by coding WHENEVER SQLERROR CONTINUE before the SQL statement, as shown in the following example:

```
EXEC SQL WHENEVER SQLERROR GOTO sql_error;
...
sql_error:
     EXEC SQL WHENEVER SQLERROR CONTINUE;
     EXEC SQL ROLLBACK WORK RELEASE;
     ...
```

Without the WHENEVER SQLERROR CONTINUE statement, a ROLLBACK error would invoke the routine again, starting an infinite loop.

Careless use of WHENEVER can cause problems. For example, the following code enters an infinite loop if the DELETE statement sets NOT FOUND because no rows meet the search condition:

```
/* improper use of WHENEVER */
...
EXEC SQL WHENEVER NOT FOUND GOTO no_more;
for (;;)
{
     EXEC SQL FETCH emp_cursor INTO :emp_name, :salary;
     ...
}
```

```
no_more:
    EXEC SQL DELETE FROM emp WHERE empno = :emp_number;
     ...
```

The next example handles the NOT FOUND condition properly by resetting the GOTO target:

```
/* proper use of WHENEVER */
...
EXEC SQL WHENEVER NOT FOUND GOTO no_more;
for (;;)
{
    EXEC SQL FETCH emp_cursor INTO :emp_name, :salary;
    ...
}
no_more:
    EXEC SQL WHENEVER NOT FOUND GOTO no_match;
    EXEC SQL DELETE FROM emp WHERE empno = :emp_number;
    ...
no_match:
    ...
```

Maintaining Addressability

Make sure all SQL statements governed by a WHENEVER GOTO statement can branch to the GOTO label. The following code results in a compile–time error because *labelA* in *func1* is not within the scope of the INSERT statement in *func2*:

```
func1()
{

    EXEC SQL WHENEVER SQLERROR GOTO labelA;
    EXEC SQL DELETE FROM emp WHERE deptno = :dept_number;
    ...
labelA:
...
}
func2()
{

    EXEC SQL INSERT INTO emp (job) VALUES (:job_title);
    ...
}
```

The label to which a WHENEVER GOTO statement branches must be in the same precompilation file as the statement.

Returning after an Error

If your program must return after handling an error, use the DO *routine_call* action. Alternatively, you can test the value of *sqlcode*, as shown in the following example:

```
...
EXEC SQL UPDATE emp SET sal = sal * 1.10;
if (sqlca.sqlcode < 0)
{  /* handle error  */

EXEC SQL DROP INDEX emp_index;
```

Just make sure no WHENEVER GOTO or WHENEVER STOP statement is active.

Obtaining the Text of SQL Statements

In many precompiler applications it is convenient to know the text of the statement being processed, its length, and the SQL command (such as INSERT or SELECT) that it contains. This is especially true for applications that use dynamic SQL.

The *sqlgls()* function—part of the SQLLIB runtime library—returns the following information:

- the text of the most recently parsed SQL statement
- the effective length of the statement
- a function code for the SQL command used in the statement

You can call *sqlgls()* after issuing a static SQL statement. For dynamic SQL Method 1, call *sqlgls()* after the SQL statement is executed. For dynamic SQL Methods 2, 3, and 4, you can call *sqlgls()* as soon as the statement has been PREPAREd.

The syntax for *sqlgls()* is

```
int sqlgls(char *sqlstm, size_t *stmlen, size_t *sqlfc);
```

The *sqlstm* parameter is a character buffer that holds the returned text of the SQL statement. Your program must statically declare the buffer or dynamically allocate memory for the buffer.

The *stmlen* parameter is a long integer. Before calling *sqlgls()*, set this parameter to the actual size, in bytes, of the *sqlstm* buffer. When *sqlgls()* returns, the *sqlstm* buffer contains the SQL statement text, blank padded to the length of the buffer. The *stmlen* parameter returns the actual number of bytes in the returned statement text, not counting blank padding.

The *sqlfc* parameter is a long integer that returns the SQL function code for the SQL command in the statement. Table 9 – 3 shows the SQL function codes for the commands.

Code	SQL Function	Code	SQL Function	Code	SQL Function
01	CREATE TABLE	26	ALTER TABLE	51	DROP TABLESPACE
02	SET ROLE	27	EXPLAIN	52	ALTER SESSION
03	INSERT	28	GRANT	53	ALTER USER
04	SELECT	29	REVOKE	54	COMMIT
05	UPDATE	30	CREATE SYNONYM	55	ROLLBACK
06	DROP ROLE	31	DROP SYNONYM	56	SAVEPOINT
07	DROP VIEW	32	ALTER SYSTEM SWITCH LOG	57	CREATE CONTROL FILE
08	DROP TABLE	33	SET TRANSACTION	58	ALTER TRACING
09	DELETE	34	PL/SQL EXECUTE	59	CREATE TRIGGER
10	CREATE VIEW	35	LOCK TABLE	60	ALTER TRIGGER
11	DROP USER	36	(NOT USED)	61	DROP TRIGGER
12	CREATE ROLE	37	RENAME	62	ANALYZE TABLE
13	CREATE SEQUENCE	38	COMMENT	63	ANALYZE INDEX
14	ALTER SEQUENCE	39	AUDIT	64	ANALYZE CLUSTER
15	(NOT USED)	40	NOAUDIT	65	CREATE PROFILE
16	DROP SEQUENCE	41	ALTER INDEX	66	DROP PROFILE
17	CREATE SCHEMA	42	CREATE EXTERNAL DATABASE	67	ALTER PROFILE
18	CREATE CLUSTER	43	DROP EXTERNAL DATABASE	68	DROP PROCEDURE

Table 9 – 3 SQL Codes

Code	SQL Function	Code	SQL Function	Code	SQL Function
19	CREATE USER	44	CREATE DATABASE	69	(NOT USED)
20	CREATE INDEX	45	ALTER DATABASE	70	ALTER RESOURCE COST
21	DROP INDEX	46	CREATE ROLLBACK SEGMENT	71	CREATE SNAPSHOT LOG
22	DROP CLUSTER	47	ALTER ROLLBACK SEGMENT	72	ALTER SNAPSHOT LOG
23	VALIDATE INDEX	48	DROP ROLLBACK SEGMENT	73	DROP SNAPSHOT LOG
24	CREATE PROCEDURE	49	CREATE TABLESPACE	74	CREATE SNAPSHOT
25	ALTER PROCEDURE	50	ALTER TABLESPACE	75	ALTER SNAPSHOT
				76	DROP SNAPSHOT

Table 9 – 3 SQL Codes

The *sqlgls()* function returns an **int**. The return value is zero (FALSE) if an error occurred, or is one (TRUE) if there was no error. The length parameter (*stmlen*) returns a zero if an error occurred. Possible error conditions are:

- no SQL statement has been parsed
- you passed an invalid parameter (for example, a negative length parameter)
- an internal exception occurred in SQLLIB

Restrictions

sqlgls() does not return the text for statements that contain the following commands:

- CONNECT
- COMMIT
- ROLLBACK
- RELEASE
- FETCH

There are no SQL function codes for these commands.

Sample Program

The sample program *sqlvcp.pc*, which is listed in Chapter 3, demonstrates how you can use the *sqlgls()* function. This program is also available on–line, in your *demo* directory.

Using the Oracle Communications Area (ORACA)

The SQLCA handles standard SQL communications; the ORACA handles Oracle communications. When you need more information about runtime errors and status changes than the SQLCA provides, use the ORACA. It contains an extended set of diagnostic tools. However, use of the ORACA is optional because it adds to runtime overhead.

Besides helping you to diagnose problems, the ORACA lets you monitor your program's use of Oracle resources such as the SQL Statement Executor and the cursor cache.

Your program can have more than one ORACA. For example, it might have one global ORACA and several local ones. Access to a local ORACA is limited by its scope within the program. Oracle returns information only to the ORACA that is in scope.

Declaring the ORACA

To declare the ORACA, copy it into your program with the INCLUDE statement or the **#include** preprocessor directive, as follows:

```
EXEC SQL INCLUDE ORACA;
```

or

```
#include <oraca.h>
```

If your ORACA must be of the **extern** storage class, define ORACA_STORAGE_CLASS in your program as follows:

```
#define ORACA_STORAGE_CLASS extern
```

If the program uses a Declare Section, the ORACA must be defined *outside* it.

Enabling the ORACA

To enable the ORACA, you must specify the ORACA option, either on the command line with

```
ORACA=YES
```

or inline with

```
EXEC ORACLE OPTION (ORACA=YES);
```

Then, you must choose appropriate runtime options by setting flags in the ORACA.

What’s in the ORACA?

The ORACA contains option settings, system statistics, and extended diagnostics such as

- SQL statement text (you can specify when to save the text)
- the name of the file in which an error occurred (useful when using subroutines)
- location of the error in a file
- cursor cache errors and statistics

A partial listing of *oraca.h* is

```
/*
NAME
  ORACA : Oracle Communications Area.

  If the symbol ORACA_NONE is defined, then there will be no ORACA
  *variable*, although there will still be a struct defined.  This
  macro should not normally be defined in application code.

  If the symbol ORACA_INIT is defined, then the ORACA will be
  statically initialized. Although this is not necessary in order
  to use the ORACA, it is a good pgming practice not to have
  unitialized variables. However, some C compilers/OS’s don’t
  allow automatic variables to be init’d in this manner.
Therefore,
  if you are INCLUDE’ing the ORACA in a place where it would be
  an automatic AND your C compiler/OS doesn’t allow this style
  of initialization, then ORACA_INIT should be left undefined ––
  all others can define ORACA_INIT if they wish.
*/

#ifndef  ORACA
#define  ORACA    1

struct  oraca
{
    char oracaid[8];   /* Reserved                 */
    long oracabc;      /* Reserved                 */

/*   Flags which are setable by User. */

   long  oracchf;      /* <> 0 if ”check cur cache consistncy”*/
   long  oradbgf;      /* <> 0 if ”do DEBUG mode checking”    */
   long  orahchf;      /* <> 0 if ”do Heap consistency check” */
   long  orastxtf;     /* SQL stmt text flag           */
#define  ORASTFNON 0   /* = don’t save text of SQL stmt       */
#define  ORASTFERR 1   /* = only save on SQLERROR         */
#define  ORASTFWRN 2   /* = only save on SQLWARNING/SQLERROR  */
#define  ORASTFANY 3      /* = always save             */
     struct
       {
  unsigned short orastxtl;
  char  orastxtc[70];
       } orastxt;         /* text of last SQL stmt         */
     struct
       {
  unsigned short orasfnml;
  char   orasfnmc[70];
       } orasfnm;        /* name of file containing SQL stmt    */
  long   oraslnr;        /* line nr–within–file of SQL stmt     */
  long   orahoc;         /* highest max open OraCurs requested  */
  long   oramoc;         /* max open OraCursors required       */
  long   oracoc;         /* current OraCursors open        */
  long   oranor;         /* nr of OraCursor re–assignments      */
  long   oranpr;         /* nr of parses               */
  long   oranex;         /* nr of executes             */
     };

#ifndef ORACA_NONE

#ifdef ORACA_STORAGE_CLASS
ORACA_STORAGE_CLASS struct oraca oraca
#else
struct oraca oraca
#endif
#ifdef ORACA_INIT
     =
     {
     {’O’,’R’,’A’,’C’,’A’,’ ’,’ ’,’ ’},
     sizeof(struct oraca),
     0,0,0,0,
     {0,{0}},
     {0,{0}},
     0,
     0,0,0,0,0,0
     }
#endif
     ;

#endif

#endif
/* end oraca.h */
```

Choosing Runtime Options

The ORACA includes several option flags. Setting these flags by assigning them non–zero values allows you to

- save the text of SQL statements
- enable DEBUG operations
- check cursor cache consistency (the *cursor cache* is a continuously updated area of memory used for cursor management)
- check heap consistency (the *heap* is an area of memory reserved for dynamic variables)
- gather cursor statistics

The descriptions below will help you choose the options you need.

Structure of the ORACA

This section describes the structure of the ORACA, its components, and the values they can store.

oracaid

This string component is initialized to "ORACA" to identify the Oracle Communications Area.

oracabc

This integer component holds the length, in bytes, of the ORACA data structure.

oracchf

If the master DEBUG flag (*oradbgf)* is set, this flag enables the gathering of cursor cache statistics and lets you check the cursor cache for consistency before every cursor operation.

The Oracle runtime library does the consistency checking and might issue error messages, which are listed in the *Oracle7 Server Messages*. manual. They are returned to the SQLCA just like Oracle error messages.

This flag has the following settings:

Disable cache consistency checking (the default).

Enable cache consistency checking.

oradbgf

This master flag lets you choose all the DEBUG options. It has the following settings:

Disable all DEBUG operations (the default).

Enable all DEBUG operations.

orahchf

If the master DEBUG flag (*oradbgf*) is set, this flag tells the Oracle runtime library to check the heap for consistency every time the precompiler dynamically allocates or frees memory. This is useful for detecting program bugs that upset memory.

This flag must be set before the CONNECT command is issued and, once set, cannot be cleared; subsequent change requests are ignored. It has the following settings:

Disable heap consistency checking (the default).

Enable heap consistency checking.

orastxtf

This flag lets you specify when the text of the current SQL statement is saved. It has the following settings:

- Never save the SQL statement text (the default).
- Save the SQL statement text on SQLERROR only.
- Save the SQL statement text on SQLERROR or SQLWARNING.
- Always save the SQL statement text.

The SQL statement text is saved in the ORACA embedded struct named *orastxt*.

Diagnostics

The ORACA provides an enhanced set of diagnostics; the following variables help you to locate errors quickly.

orastxt

This embedded struct helps you find faulty SQL statements. It lets you save the text of the last SQL statement parsed by Oracle. It contains the following two components:

orastxtl This integer component holds the length of the current SQL statement.

orastxtc This string component holds the text of the current SQL statement. At most, the first 70 characters of text are saved. The string is *not* null terminated. Use the *oratxtl* length component when printing the string.

Statements parsed by the precompiler, such as CONNECT, FETCH, and COMMIT, are *not* saved in the ORACA.

orasfnm

This embedded struct identifies the file containing the current SQL statement and so helps you find errors when multiple files are precompiled for one application. It contains the following two components:

orasfnml This integer component holds the length of the filename stored in *orasfnmc*.

orasfnmc This string component holds the filename. At most, the first 70 characters are stored.

oraslnr This integer component identifies the line at (or near) which the current SQL statement can be found.

Cursor Cache Statistics

If the master DEBUG flag (*oradbgf*) and the cursor cache flag (*oracchf*) are set, the variables below let you gather cursor cache statistics. They are automatically set by every COMMIT or ROLLBACK command your program issues.

Internally, there is a set of these variables for each CONNECTed database. The current values in the ORACA pertain to the database against which the last COMMIT or ROLLBACK was executed.

orahoc This integer component records the highest value to which MAXOPENCURSORS was set during program execution.

oramoc This integer component records the maximum number of open Oracle cursors required by your program. This number can be higher than *orahoc* if MAXOPENCURSORS was set too low, which forced the precompiler to extend the cursor cache.

oracoc This integer component records the current number of open Oracle cursors required by your program.

oranor This integer component records the number of cursor cache reassignments required by your program. This number shows the degree of “thrashing” in the cursor cache and should be kept as low as possible.

oranpr This integer component records the number of SQL statement parses required by your program.

oranex This integer component records the number of SQL statement executions required by your program. The ratio of this number to the *oranpr* number should be kept as high as possible. In other words, avoid unnecessary reparsing. For help, see Appendix C.

An ORACA Example

The following program prompts for a department number, inserts the name and salary of each employee in that department into one of two tables, then displays diagnostic information from the ORACA. This program is available online in the *demo* directory, as *oraca.pc*.

```
/* oraca.pc
 * This sample program demonstrates how to
 * use the ORACA to determine various performance
 * parameters at runtime.
 */
#include <stdio.h>
#include <string.h>
#include <sqlca.h>
#include <oraca.h>

EXEC SQL BEGIN DECLARE SECTION;
char *userid = ”SCOTT/TIGER”;
char  emp_name[21];
int   dept_number;
float salary;
char SQLSTATE[6];
EXEC SQL END DECLARE SECTION;

void sql_error();

main()
{
    char temp_buf[32];

    EXEC SQL WHENEVER SQLERROR DO sql_error(”Oracle error”);
    EXEC SQL CONNECT :userid;

    EXEC ORACLE OPTION (ORACA=YES);

    oraca.oradbgf  = 1;             /* enable debug operations */
    oraca.oracchf  = 1;      /* gather cursor cache statistics */
    oraca.orastxtf = 3;       /* always save the SQL statement */

    printf(”Enter department number: ”);
    gets(temp_buf);
    dept_number = atoi(temp_buf);


    EXEC SQL DECLARE emp_cursor CURSOR FOR
      SELECT ename, sal + NVL(comm,0) AS sal_comm
        FROM emp
        WHERE deptno = :dept_number
        ORDER BY sal_comm DESC;
    EXEC SQL OPEN emp_cursor;
    EXEC SQL WHENEVER NOT FOUND DO sql_error(”End of data”);

    for (;;)
    {
        EXEC SQL FETCH emp_cursor INTO :emp_name, :salary;
        printf(”%.10s\n”, emp_name);
        if (salary < 2500)
            EXEC SQL INSERT INTO pay1 VALUES (:emp_name, :salary);
       else
            EXEC SQL INSERT INTO pay2 VALUES (:emp_name, :salary);
   }
}

void
sql_error(errmsg)
char *errmsg;
{
    char buf[6];

    strcpy(buf, SQLSTATE);
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL COMMIT WORK RELEASE;

    if (strncmp(errmsg, ”Oracle error”, 12) == 0)
        printf(”\n%s, sqlstate is %s\n\n”, errmsg, buf);
    else
        printf(”\n%s\n\n”, errmsg);

    printf(”Last SQL statement: %.*s\n”,
    oraca.orastxt.orastxtl, oraca.orastxt.orastxtc);
    printf(”\nAt or near line number %d\n”, oraca.oraslnr);
    printf
(”\nCursor Cache Statistics\n------------------------\n”);
    printf
(”Maximum value of MAXOPENCURSORS:    %d\n”, oraca.orahoc);
    printf
(”Maximum open cursors required:      %d\n”, oraca.oramoc);
    printf
(”Current number of open cursors:     %d\n”, oraca.oracoc);
    printf
(”Number of cache reassignments:      %d\n”, oraca.oranor);
    printf
(”Number of SQL statement parses:     %d\n”, oraca.oranpr);
    printf
(”Number of SQL statement executions: %d\n”, oraca.oranex);
    exit(1);
}
```

CHAPTER

10

Using Host Arrays

This chapter looks at using arrays to simplify coding and improve program performance. You learn how to manipulate Oracle data using arrays, how to operate on all the elements of an array with a single SQL statement, and how to limit the number of array elements processed. The following questions are answered:

- What is a host array?
- Why use arrays?
- How are host arrays declared?
- How are arrays used in SQL statements?

What Is a Host Array?

An *array* is a collection of related data items, called *elements*, associated with a single variable name. When declared as a host variable, the array is called a *host array.* Likewise, an indicator variable declared as an array is called an *indicator array.* An indicator array can be associated with any host array.

Why Use Arrays?

Arrays can ease programming and offer improved performance.

When writing an application, you are usually faced with the problem of storing and manipulating large collections of data. Arrays simplify the task of naming and referencing the individual items in each collection.

Using arrays can boost the performance of your application. Arrays let you manipulate an entire collection of data items with a single SQL statement. Thus, Oracle communication overhead is reduced markedly, especially in a networked environment.

For example, suppose you want to insert information about 300 employees into the EMP table. Without arrays your program must do 300 individual INSERTs—one for each employee. With arrays, only one INSERT need be done.

Declaring Host Arrays

You declare host arrays just like scalar host variables. If MODE=ANSI, you should declare them in a Declare Section. Dimension the arrays when they are declared. The following example declares three host arrays, each with a dimension of 50 elements:

```
char  emp_name[50][10];
int   emp_number[50];
float salary[50];
```

Restrictions

You cannot declare host arrays of pointers.

Except for character arrays (strings), host arrays that might be referenced in a SQL statement are limited to one dimension. So, the two–dimensional array declared in the following example is *invalid*:

```
int hi_lo_scores[25][25];   /* not allowed */
```

Dimensioning Arrays

The maximum dimension of a host array is 32,767 elements. If you use a host array that exceeds the maximum, you get a “parameter out of range” runtime error.

If you use multiple host arrays in a single SQL statement, their dimensions should be the same. Otherwise, an “array size mismatch” warning message is issued at precompile time. If you ignore this warning, the precompiler uses the *smallest* dimension for the SQL operation.

Using Arrays in SQL Statements

The Pro*C Precompiler allows the use of host arrays in data manipulation statements. You can use host arrays as input variables in the INSERT, UPDATE, and DELETE statements and as output variables in the INTO clause of SELECT and FETCH statements.

The embedded SQL syntax used for host arrays and simple host variables is nearly the same. One difference is the optional FOR clause, which lets you control array processing. Also, there are restrictions on mixing host arrays and simple host variables in a SQL statement.

The following sections illustrate the use of host arrays in data manipulation statements.

Selecting into Arrays

You can use host arrays as output variables in the SELECT statement. If you know the maximum number of rows the SELECT will return, simply dimension the host arrays with that number of elements. In the following example, you select directly into three host arrays. Knowing the SELECT will return no more than 50 rows, you dimension the arrays with 50 elements:

```
char   emp_name[50][20];
int    emp_number[50];
float  salary[50];

EXEC SQL SELECT ENAME, EMPNO, SAL
    INTO :emp_name, :emp_number, :salary
    FROM EMP
    WHERE SAL > 1000;
```

In this example, the SELECT statement returns up to 50 rows. If there are fewer than 50 eligible rows or you want to retrieve only 50 rows, this method will suffice. However, if there are more than 50 eligible rows, you cannot retrieve all of them this way. If you re–execute the SELECT statement, it just returns the first 50 rows again, even if more are eligible. You must either dimension a larger array or declare a cursor for use with the FETCH statement.

If a SELECT INTO statement returns more rows than the number of elements you dimensioned, Oracle issues the error message

```
ORA–02112: PCC: SELECT ...INTO returns too many rows
```

unless you specify SELECT_ERROR=NO. For more information about the SELECT_ERROR option, see the section "Using the Precompiler Options" on page 7 – 9 .

Batch Fetches

If you do not know the maximum number of rows a SELECT will return, you can declare and open a cursor, then fetch from it in "batches."

Batch fetches within a loop let you retrieve a large number of rows with ease. Each FETCH returns the next batch of rows from the current active set. In the following example, you fetch in 20–row batches:

```
int   emp_number[20];
float salary[20];

EXEC SQL DECLARE emp_cursor CURSOR FOR
    SELECT empno, sal FROM emp;

EXEC SQL OPEN emp_cursor;

EXEC SQL WHENEVER NOT FOUND do break;
for (;;)
{
    EXEC SQL FETCH emp_cursor
        INTO :emp_number, :salary;
    /* process batch of rows */
    ...
}
```

See the demo program *sample3.pc* (page 3 – 30) for an additional example of batch fetching.

Number of Rows Fetched

Each FETCH returns, at most, the number of rows in the array dimension. Fewer rows are returned in the following cases:

- The end of the active set is reached. The "no data found" Oracle error code is returned to SQLCODE in the SQLCA. For example, this happens if you fetch into an array of dimension 100 but only 20 rows are returned.
- Fewer than a full batch of rows remain to be fetched. For example, this happens if you fetch 70 rows into an array of dimension 20 because after the third FETCH, only 10 rows remain to be fetched.
- An error is detected while processing a row. The FETCH fails and the applicable Oracle error code is returned to SQLCODE.

The cumulative number of rows returned can be found in the third element of *sqlerrd* in the SQLCA, called *sqlerrd[2]* in this guide. This applies to each open cursor. In the following example, notice how the status of each cursor is maintained separately:

```
EXEC SQL OPEN cursor1;
EXEC SQL OPEN cursor2;
EXEC SQL FETCH cursor1 INTO :array_of_20;
/* now running total in sqlerrd[2] is 20 */
EXEC SQL FETCH cursor2 INTO :array_of_30;
/* now running total in sqlerrd[2] is 30, not 50 */
EXEC SQL FETCH cursor1 INTO :array_of_20;
/* now running total in sqlerrd[2] is 40 (20 + 20) */
EXEC SQL FETCH cursor2 INTO :array_of_30;
/* now running total in sqlerrd[2] is 60 (30 + 30) */
```

Restrictions

Using host arrays in the WHERE clause of a SELECT statement is *not* allowed except in a subquery. For an example, see the section "Using the WHERE Clause" on page 10 – 12.

Also, you cannot mix simple host variables with host arrays in the INTO clause of a SELECT or FETCH statement. If any of the host variables is an array, all must be arrays.

Table 10 – 1 shows which uses of host arrays are valid in a SELECT INTO statement:

INTO Clause	WHERE Clause	Valid?
array	array	no
scalar	scalar	yes
array	scalar	yes
scalar	array	no

Table 10 – 1 Valid Host Arrays for SELECT INTO

Fetching Nulls

When DBMS=V6, if you SELECT or FETCH null column values into a host array not associated with an indicator array, no error is generated. So, when doing array SELECTs and FETCHes, always use indicator arrays. That way, you can test for nulls in the associated output host array.

When DBMS={V7 | V6_CHAR}, if you SELECT or FETCH a null column value into a host array not associated with an indicator array, Oracle stops processing, sets *sqlerrd[2]* to the number of rows processed, and issues the following error message:

```
ORA–01405: fetched column value is NULL
```

Fetching Truncated Values

When DBMS=V6, if you SELECT or FETCH a truncated column value into a host array not associated with an indicator array, Oracle stops processing, sets *sqlerrd[2]* to the number of rows processed, and issues the following error message:

```
ORA–01406: fetched column value was truncated
```

In either case, you can check *sqlerrd[2]* for the number of rows processed before the truncation occurred. The rows–processed count includes the row that caused the truncation error.

When DBMS={V7 | V6_CHAR}, truncation is not considered an error, so Oracle continues processing.

Again, when doing array SELECTs and FETCHes, always use indicator arrays. That way, if Oracle assigns one or more truncated column values to an output host array, you can find the original lengths of the column values in the associated indicator array.

Inserting with Arrays

You can use host arrays as input variables in an INSERT statement. Just make sure your program populates the arrays with data before executing the INSERT statement.

If some elements in the arrays are irrelevant, you can use the FOR clause to control the number of rows inserted. See the section "Using the FOR Clause" on page 10 – 10.

An example of inserting with host arrays follows:

```
char   emp_name[50][20];
int    emp_number[50];
float  salary[50];
/* populate the host arrays */
...
EXEC SQL INSERT INTO EMP (ENAME, EMPNO, SAL)
    VALUES (:emp_name, :emp_number, :salary);
```

The cumulative number of rows inserted can be found in the rows–processed count, *sqlca.sqlerrd[2]*.

In the following example, the INSERT is done one row at a time. This is much less efficient than the previous example, since a call to the server must be made for each row inserted.

```
for (i = 0; i < array_dimension; i++)
    EXEC SQL INSERT INTO emp (ename, empno, sal)
        VALUES (:emp_name[i], :emp_number[i], :salary[i]);
```

Restrictions

You cannot use an array of pointers in the VALUES clause of an INSERT statement; all array elements must be data items.

Mixing simple host variables with host arrays in the VALUES clause of an INSERT statement is *not* allowed. If any of the host variables is an array, all must be arrays.

Updating with Arrays

You can also use host arrays as input variables in an UPDATE statement, as the following example shows:

```
int   emp_number[50];
float salary[50];
/* populate the host arrays */
EXEC SQL UPDATE emp SET sal = :salary
    WHERE EMPNO = :emp_number;
```

The cumulative number of rows updated can be found in *sqlerrd[2]*. The number does *not* include rows processed by an update cascade.

If some elements in the arrays are irrelevant, you can use the embedded SQL FOR clause to limit the number of rows updated.

The last example showed a typical update using a unique key (EMP_NUMBER). Each array element qualified just one row for updating. In the following example, each array element qualifies multiple rows:

```
char  job_title [10][20];
float commission[10];

...

EXEC SQL UPDATE emp SET comm = :commission
    WHERE job = :job_title;
```

Restrictions

Mixing simple host variables with host arrays in the SET or WHERE clause of an UPDATE statement is *not* recommended. If any of the host variables is an array, all should be arrays. Furthermore, if you use a host array in the SET clause, use one of equal dimension in the WHERE clause.

You cannot use host arrays with the CURRENT OF clause in an UPDATE statement. For an alternative, see the section “Mimicking CURRENT OF” on page 10 – 13.

Table 10 – 2 shows which uses of host arrays are valid in an UPDATE statement:

SET Clause	WHERE Clause	Valid?
array	array	yes
scalar	scalar	yes
array	scalar	no
scalar	array	no

Table 10 – 2 Host Arrays Valid in an UPDATE

Deleting with Arrays

You can also use host arrays as input variables in a DELETE statement. It is like executing the DELETE statement repeatedly using successive elements of the host array in the WHERE clause. Thus, each execution might delete zero, one, or more rows from the table.

An example of deleting with host arrays follows:

```
...
int emp_number[50];

/* populate the host array */
...
EXEC SQL DELETE FROM emp
    WHERE empno = :emp_number;
```

The cumulative number of rows deleted can be found in *sqlerrd[2]*. The number does *not* include rows processed by a delete cascade.

The last example showed a typical delete using a unique key (EMP_NUMBER). Each array element qualified just one row for deletion. In the following example, each array element qualifies multiple rows:

```
...
char job_title[10][20];

/* populate the host array  */
...
EXEC SQL DELETE FROM emp
    WHERE job = :job_title;
```

Restrictions

Mixing simple host variables with host arrays in the WHERE clause of a DELETE statement is *not* allowed. If any of the host variables is an array, all must be arrays.

You cannot use host arrays with the CURRENT OF clause in a DELETE statement. For an alternative, see the section ”Mimicking CURRENT OF” on page 10 – 13.

Using Indicator Arrays

You use indicator arrays to assign nulls to input host arrays and to detect null or truncated values in output host arrays. The following example shows how to insert with indicator arrays:

```
int   emp_number[50];
int   dept_number[50];
float commission[50];
short ind_comm[50];   /* indicator array  */
/* populate the host arrays and
   populate the indicator array; to insert a null into
   the COMM column, assign –1 to the appropriate element in
   the indicator array
*/
EXEC SQL INSERT INTO emp (empno, deptno, comm)
    VALUES (:emp_number, :dept_number, :commission INDICATOR
            :ind_comm);
```

The indicator array dimension cannot be smaller than the host array dimension.

Using the FOR Clause

You can use the optional embedded SQL FOR clause to set the number of array elements processed by any of the following SQL statements:

- DELETE
- EXECUTE
- FETCH
- INSERT
- OPEN
- UPDATE

The FOR clause is especially useful in UPDATE, INSERT, and DELETE statements. With these statements you might not want to use the entire array. The FOR clause lets you limit the elements used to just the number you need, as the following example shows:

```
char  emp_name[100][20];
float salary[100];
int   rows_to_insert;

/* populate the host arrays */
rows_to_insert = 25;             /* set FOR–clause variable */
EXEC SQL FOR :rows_to_insert   /* will process only 25 rows */
    INSERT INTO emp (ename, sal)
    VALUES (:emp_name, :salary);
```

The FOR clause can use an integer host variable to count array elements, or an integer literal. A complex C expression that resolves to an integer *cannot* be used. For example, the following statement that uses an integer expression is illegal:

```
EXEC SQL FOR :rows_to_insert + 5                 /* illegal */
    INSERT INTO emp (ename, empno, sal)
        VALUES (:emp_name, :emp_number, :salary);
```

The FOR clause variable specifies the number of array elements to be processed. Make sure the number is not larger than the smallest array dimension. Also, the number must be positive. If it is negative or zero, no rows are processed and Oracle issues an error message.

Restrictions

Two restrictions keep FOR clause semantics clear. You cannot use the FOR clause in a SELECT statement or with the CURRENT OF clause.

In a SELECT Statement

If you use the FOR clause in a SELECT statement, you get the following error message:

```
PCC–E–0056:  FOR clause not allowed on SELECT statement at ...
```

The FOR clause is not allowed in SELECT statements because its meaning is unclear. Does it mean “execute this SELECT statement *n* times”? Or, does it mean “execute this SELECT statement once, but return *n* rows”? The problem in the former case is that each execution might return multiple rows. In the latter case, it is better to declare a cursor and use the FOR clause in a FETCH statement, as follows:

```
EXEC SQL FOR :limit FETCH emp_cursor INTO ...
```

With the CURRENT OF Clause

You can use the CURRENT OF clause in an UPDATE or DELETE statement to refer to the latest row returned by a FETCH statement, as the following example shows:

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
    SELECT ename, sal FROM emp WHERE empno = :emp_number;
...
EXEC SQL OPEN emp_cursor;
...
EXEC SQL FETCH emp_cursor INTO :emp_name, :salary;
...
EXEC SQL UPDATE emp SET sal = :new_salary
WHERE CURRENT OF emp_cursor;
```

However, you cannot use the FOR clause with the CURRENT OF clause. The following statements are invalid because the only logical value of *limit* is 1 (you can only update or delete the current row once):

```
EXEC SQL FOR :limit UPDATE emp SET sal = :new_salary
WHERE CURRENT OF emp_cursor;
...
EXEC SQL FOR :limit DELETE FROM emp
WHERE CURRENT OF emp_cursor;
```

Using the WHERE Clause

Oracle treats a SQL statement containing host arrays of dimension *n* like the same SQL statement executed *n* times with *n* different scalar variables (the individual array elements). The precompiler issues the following error message only when such treatment would be ambiguous:

```
PCC–S–0055: Array <name> not allowed as bind variable at ...
```

For example, assuming the declarations

```
int  mgr_number[50];
char job_title[50][20];
```

it would be ambiguous if the statement

```
EXEC SQL SELECT mgr INTO :mgr_number FROM emp
WHERE job = :job_title;
```

were treated like the imaginary statement

```
for (i = 0; i < 50; i++)
    SELECT mgr INTO :mgr_number[i] FROM emp
        WHERE job = :job_title[i];
```

because multiple rows might meet the WHERE–clause search condition, but only one output variable is available to receive data. Therefore, an error message is issued.

On the other hand, it would not be ambiguous if the statement

```
EXEC SQL UPDATE emp SET mgr = :mgr_number
    WHERE empno IN (SELECT empno FROM emp
        WHERE job = :job_title);
```

were treated like the imaginary statement

```
for (i = 0; i < 50; i++)
    UPDATE emp SET mgr = :mgr_number[i]
        WHERE empno IN (SELECT empno FROM emp
            WHERE job = :job_title[i]);
```

because there is a *mgr_number* in the SET clause for each row matching *job_title* in the WHERE clause, even if each *job_title* matches multiple rows. All rows matching each *job_title* can be SET to the same *mgr_number*. Therefore, no error message is issued.

Mimicking CURRENT OF

You use the CURRENT OF *cursor* clause in a DELETE or UPDATE statement to refer to the latest row FETCHed from the cursor. (For more information, see the section "Using the CURRENT OF Clause" on page 4 – 17.) However, you cannot use CURRENT OF with host arrays. Instead, select the ROWID of each row, then use that value to identify the current row during the update or delete. An example follows:

```
char  emp_name[20][10];
char  job_title[20][10];
char  old_title[20][10];
char  row_id[20][18];
...
EXEC SQL DECLARE emp_cursor CURSOR FOR
SELECT ename, job, rowid FROM emp;
...
EXEC SQL OPEN emp_cursor;
EXEC SQL WHENEVER NOT FOUND do break;
for (;;)
{
    EXEC SQL FETCH emp_cursor
        INTO :emp_name, :job_title, :row_id;
    ...
    EXEC SQL DELETE FROM emp
        WHERE job = :old_title AND rowid = :row_id;
    EXEC SQL COMMIT WORK;
}
```

However, the fetched rows are *not* locked because no FOR UPDATE OF clause is used. (You cannot use FOR UPDATE OF without CURRENT

OF.) So, you might get inconsistent results if another user changes a row after you read it but before you delete it.

Using *sqlca.sqlerrd[2]*

For INSERT, UPDATE, DELETE, and SELECT INTO statements, *sqlca.sqlerrd[2]* records the number of rows processed. For FETCH statements, it records the cumulative sum of rows processed.

When using host arrays with FETCH, to find the number of rows returned by the most recent iteration, subtract the current value of *sqlca.sqlerrd[2]* from its previous value (stored in another variable). In the following example, you determine the number of rows returned by the most recent fetch:

```
int  emp_number[100];
char emp_name[100][20];

int rows_to_fetch, rows_before, rows_this_time;
EXEC SQL DECLARE emp_cursor CURSOR FOR
SELECT empno, ename
FROM emp
WHERE deptno = 30;
EXEC SQL OPEN emp_cursor;
EXEC SQL WHENEVER NOT FOUND CONTINUE;
/* initialize loop variables */
rows_to_fetch = 20;   /* number of rows in each ”batch” */
rows_before = 0;      /* previous value of sqlerrd[2]  */
rows_this_time = 20;

while (rows_this_time == rows_to_fetch)
{
    EXEC SQL FOR :rows_to_fetch
    FETCH emp_cursor
        INTO :emp_number, :emp_name;
    rows_this_time = sqlca.sqlerrd[2] – rows_before;
    rows_before = sqlca.sqlerrd[2];
}
...
```

sqlca.sqlerrd[2] is also useful when an error occurs during an array operation. Processing stops at the row that caused the error, so *sqlerrd[2]* gives the number of rows processed successfully.

CHAPTER

11

Using Dynamic SQL

This chapter shows you how to use dynamic SQL, an advanced programming technique that adds flexibility and functionality to your applications. After weighing the advantages and disadvantages of dynamic SQL, you learn four methods—from simple to complex—for writing programs that accept and process SQL statements "on the fly" at run time. You learn the requirements and limitations of each method and how to choose the right method for a given job.

What Is Dynamic SQL?

Most database applications do a specific job. For example, a simple program might prompt the user for an employee number, then update rows in the EMP and DEPT tables. In this case, you know the makeup of the UPDATE statement at precompile time. That is, you know which tables might be changed, the constraints defined for each table and column, which columns might be updated, and the datatype of each column.

However, some applications must accept (or build) and process a variety of SQL statements at run time. For example, a general–purpose report writer must build different SELECT statements for the various reports it generates. In this case, the statement's makeup is unknown until run time. Such statements can, and probably will, change from execution to execution. They are aptly called *dynamic* SQL statements.

Unlike static SQL statements, dynamic SQL statements are not embedded in your source program. Instead, they are stored in character strings input to or built by the program at run time. They can be entered interactively or read from a file.

Advantages and Disadvantages of Dynamic SQL

Host programs that accept and process dynamically defined SQL statements are more versatile than plain embedded SQL programs. Dynamic SQL statements can be built interactively with input from users having little or no knowledge of SQL.

For example, your program might simply prompt users for a search condition to be used in the WHERE clause of a SELECT, UPDATE, or DELETE statement. A more complex program might allow users to choose from menus listing SQL operations, table and view names, column names, and so on. Thus, dynamic SQL lets you write highly flexible applications.

However, some dynamic queries require complex coding, the use of special data structures, and more runtime processing. While you might not notice the added processing time, you might find the coding difficult unless you fully understand dynamic SQL concepts and methods.

When to Use Dynamic SQL

In practice, static SQL will meet nearly all your programming needs. Use dynamic SQL only if you need its open–ended flexibility. Its use is suggested when one of the following items is unknown at precompile time:

- text of the SQL statement (commands, clauses, and so on)
- the number of host variables
- the datatypes of host variables
- references to database objects such as columns, indexes, sequences, tables, usernames, and views

Requirements for Dynamic SQL Statements

To represent a dynamic SQL statement, a character string must contain the text of a valid SQL statement, but *not* contain the EXEC SQL clause, or the statement terminator, or any of the following embedded SQL commands:

- CLOSE
- DECLARE
- DESCRIBE
- EXECUTE
- FETCH
- INCLUDE
- OPEN
- PREPARE
- WHENEVER

In most cases, the character string can contain *dummy* host variables. They hold places in the SQL statement for actual host variables. Because dummy host variables are just placeholders, you do not declare them and can name them anything you like. For example, Oracle makes no distinction between the following two strings:

```
'DELETE FROM EMP WHERE MGR = :mgr_number AND JOB = :job_title'
'DELETE FROM EMP WHERE MGR = :m AND JOB = :j'
```

How Dynamic SQL Statements Are Processed

Typically, an application program prompts the user for the text of a SQL statement and the values of host variables used in the statement. Then Oracle *parses* the SQL statement. That is, Oracle examines the SQL statement to make sure it follows syntax rules and refers to valid database objects. Parsing also involves checking database access rights, reserving needed resources, and finding the optimal access path.

Next, Oracle *binds* the host variables to the SQL statement. That is, Oracle gets the addresses of the host variables so that it can read or write their values.

Then Oracle *executes* the SQL statement. That is, Oracle does what the SQL statement requested, such as deleting rows from a table.

The SQL statement can be executed repeatedly using new values for the host variables.

Methods for Using Dynamic SQL

This section introduces four methods you can use to define dynamic SQL statements. It briefly describes the capabilities and limitations of each method, then offers guidelines for choosing the right method. Later sections show you how to use the methods, and include sample programs that you can study.

The four methods are increasingly general. That is, Method 2 encompasses Method 1, Method 3 encompasses Methods 1 and 2, and so on. However, each method is most useful for handling a certain kind of SQL statement, as the following table shows:

Method	Kind of SQL Statement
1	non query without host variables
2	non query with known number of input host variables
3	query with known number of select–list items and input host variables
4	query with unknown number of select–list items or input host variables

Note: The term *select–list item* includes column names and expressions such as SAL * 1.10 and MAX(SAL).

Method 1

This method lets your program accept or build a dynamic SQL statement, then immediately execute it using the EXECUTE IMMEDIATE command. The SQL statement must not be a query (SELECT statement) and must not contain any placeholders for input host variables. For example, the following host strings qualify:

```
'DELETE FROM EMP WHERE DEPTNO = 20'
'GRANT SELECT ON EMP TO scott'
```

With Method 1, the SQL statement is parsed every time it is executed.

Method 2

This method lets your program accept or build a dynamic SQL statement, then process it using the PREPARE and EXECUTE commands. The SQL statement must not be a query. The number of placeholders for input host variables and the datatypes of the input host variables must be known at precompile time. For example, the following host strings fall into this category:

```
'INSERT INTO EMP (ENAME, JOB) VALUES (:emp_name, :job_title)'
'DELETE FROM EMP WHERE EMPNO = :emp_number'
```

With Method 2, the SQL statement is parsed just once, but can be executed many times with different values for the host variables. SQL data definition statements such as CREATE and GRANT are executed when they are PREPAREd.

Method 3

This method lets your program accept or build a dynamic query, then process it using the PREPARE command with the DECLARE, OPEN, FETCH, and CLOSE cursor commands. The number of select–list items, the number of placeholders for input host variables, and the datatypes of the input host variables must be known at precompile time. For example, the following host strings qualify:

```
'SELECT DEPTNO, MIN(SAL), MAX(SAL) FROM EMP GROUP BY DEPTNO'
'SELECT ENAME, EMPNO FROM EMP WHERE DEPTNO = :dept_number'
```

Method 4

This method lets your program accept or build a dynamic SQL statement, then process it using descriptors (discussed in the section "Using Method 4" on page NO TAG). The number of select–list items, the number of placeholders for input host variables, and the datatypes of the input host variables can be unknown until run time. For example, the following host strings fall into this category:

```
'INSERT INTO EMP (<unknown>) VALUES (<unknown>)'
'SELECT <unknown> FROM EMP WHERE DEPTNO = 20'
```

Method 4 is required for dynamic SQL statements that contain an unknown number of select–list items or input host variables.

Guidelines

With all four methods, you must store the dynamic SQL statement in a character string, which must be a host variable or quoted literal. When you store the SQL statement in the string, omit the keywords EXEC SQL and the ';' statement terminator.

With Methods 2 and 3, the number of placeholders for input host variables and the datatypes of the input host variables must be known at precompile time.

Each succeeding method imposes fewer constraints on your application, but is more difficult to code. As a rule, use the simplest method you can. However, if a dynamic SQL statement will be executed repeatedly by Method 1, use Method 2 instead to avoid reparsing for each execution.

Method 4 provides maximum flexibility, but requires complex coding and a full understanding of dynamic SQL concepts. In general, use Method 4 only if you cannot use Methods 1, 2, or 3.

The decision logic in Figure 11 – 1 will help you choose the right method.

Avoiding Common Errors

If you precompile using the command–line option DBMS=V6 or DBMS=V6_CHAR, blank–pad the array before storing the SQL statement. That way, you clear extraneous characters. This is especially important when you reuse the array for different SQL statements. As a rule, always initialize (or re–initialize) the host string before storing the SQL statement. Do *not* null–terminate the host string. Oracle does not recognize the null terminator as an end–of–string sentinel. Instead, Oracle treats it as part of the SQL statement.

If you precompile with the command–line option DBMS=V7 (the default), make sure that the string is null terminated before you execute the PREPARE or EXECUTE IMMEDIATE statement.

Regardless of the value of DBMS, if you use a VARCHAR variable to store the dynamic SQL statement, make sure the length of the VARCHAR is set (or reset) correctly before you execute the PREPARE or EXECUTE IMMEDIATE statement.

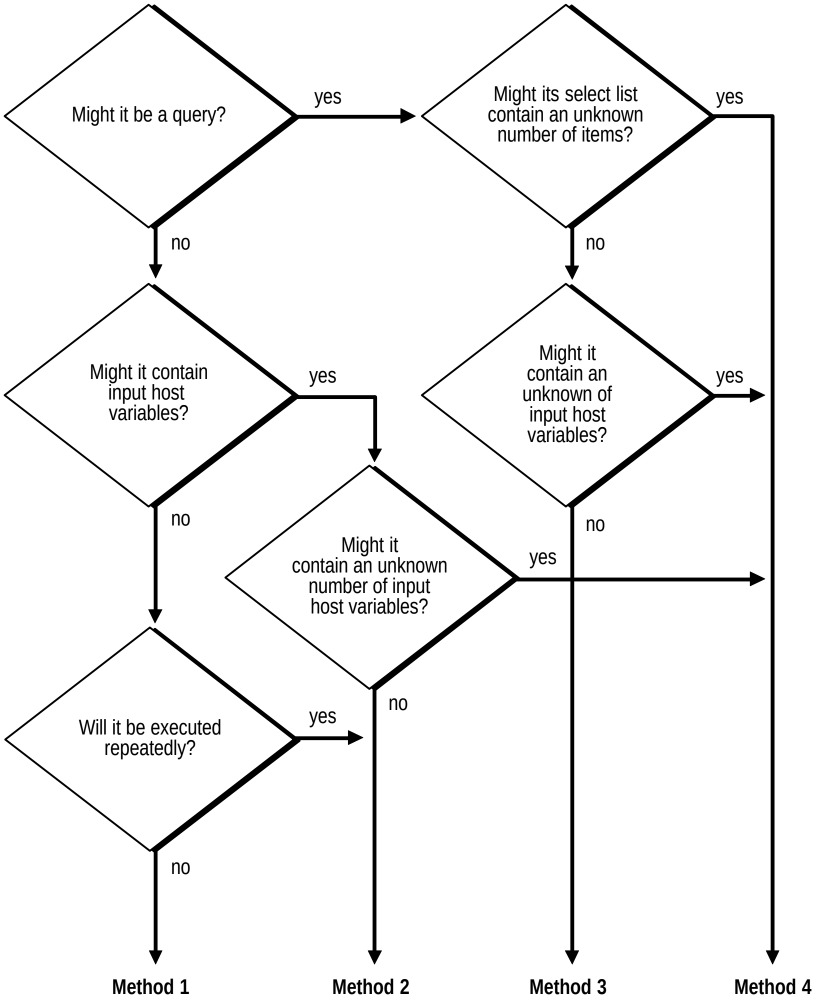

Might it be a query?
yes
Might its select list contain an unknown number of items?
yes
no
no
Might it contain input host variables?
yes
Might it contain an unknown of input host variables?
yes
no
no
Might it contain an unknown number of input host variables?
yes
Will it be executed repeatedly?
yes
no
no
Method 1
Method 2
Method 3
Method 4

Figure 11 – 1 Choosing the Right Method

Using Method 1

The simplest kind of dynamic SQL statement results only in “success” or “failure” and uses no host variables. Some examples follow:

```
’DELETE FROM table_name WHERE column_name = constant’
’CREATE TABLE table_name ...’
’DROP INDEX index_name’
’UPDATE table_name SET column_name = constant’
’GRANT SELECT ON table_name TO username’
’REVOKE RESOURCE FROM username’
```

Method 1 parses, then immediately executes the SQL statement using the EXECUTE IMMEDIATE command. The command is followed by a character string (host variable or literal) containing the SQL statement to be executed, which cannot be a query.

The syntax of the EXECUTE IMMEDIATE statement follows:

```
EXEC SQL EXECUTE IMMEDIATE { :host_string | string_literal };
```

In the following example, you use the host variable *dyn_stmt* to store SQL statements input by the user:

```
char dyn_stmt[132];
...
for (;;)
{
    printf(”Enter SQL statement: ”);
    gets(dyn_stmt);
    if (*dyn_stmt == ’\0’)
        break;
    /* dyn_stmt now contains the text of a SQL statement */
    EXEC SQL EXECUTE IMMEDIATE :dyn_stmt;
}
...
```

You can also use string literals, as the following example shows:

```
EXEC SQL EXECUTE IMMEDIATE ’REVOKE RESOURCE FROM MILLER’;
```

Because EXECUTE IMMEDIATE parses the input SQL statement before every execution, Method 1 is best for statements that are executed only once. Data definition language statements usually fall into this category.

Sample Program: Dynamic SQL Method 1

The following program uses dynamic SQL Method 1 to create a table, insert a row, commit the insert, then drop the table. This program is available on–line in your demo directory in the file *sample6.pc*.

```
/*
 *  sample6.pc: Dynamic SQL Method 1
 *
 *  This program uses dynamic SQL Method 1 to create a table,
 *  insert a row, commit the insert, then drop the table.
 */

#include <stdio.h>
#include <string.h>

/* Include the SQL Communications Area, a structure through
 * which ORACLE makes runtime status information such as error
 * codes, warning flags, and diagnostic text available to the
 * program.
 */
#include <sqlca.h>

/* Include the ORACLE Communications Area, a structure through
 * which ORACLE makes additional runtime status information
 * available to the program.
 */
#include <oraca.h>

/* The ORACA=YES option must be specified to enable you
 * to use the ORACA.
 */

EXEC ORACLE OPTION (ORACA=YES);

/* Specifying the RELEASE_CURSOR=YES option instructs Pro*C
 * to release resources associated with embedded SQL
 * statements after they are executed.  This ensures that
 * ORACLE does not keep parse locks on tables after data
 * manipulation operations, so that subsequent data definition
 * operations on those tables do not result in a parse–lock
 * error.
 */

EXEC ORACLE OPTION (RELEASE_CURSOR=YES);

void dyn_error();


main()
```

```
{
/* Declare the program host variables. */
    char    *username = ”SCOTT”;
    char    *password = ”TIGER”;
    char    *dynstmt1;
    char     dynstmt2[10];
    VARCHAR  dynstmt3[80];

/* Call routine dyn_error() if an ORACLE error occurs. */

    EXEC SQL WHENEVER SQLERROR DO dyn_error(”Oracle error:”);

/* Save text of current SQL statement in the ORACA if an
 * error occurs.
 */
    oraca.orastxtf = ORASTFERR;

/* Connect to Oracle. */

    EXEC SQL CONNECT :username IDENTIFIED BY :password;
    puts(”\nConnected to ORACLE.\n”);

/* Execute a string literal to create the table.  This
 * usage is actually not dynamic because the program does
 * not determine the SQL statement at run time.
 */
    puts(”CREATE TABLE dyn1 (col1 VARCHAR2(4))”);

    EXEC SQL EXECUTE IMMEDIATE
         ”CREATE TABLE dyn1 (col1 VARCHAR2(4))”;

/* Execute a string to insert a row.  The string must
 * be null–terminated.  This usage is dynamic because the
 * SQL statement is a string variable whose contents the
 * program can determine at run time.
 */
    dynstmt1 = ”INSERT INTO DYN1 values (’TEST’)”;
    puts(dynstmt1);

    EXEC SQL EXECUTE IMMEDIATE :dynstmt1;

/* Execute a SQL statement in a string to commit the insert.
 * Pad the unused trailing portion of the array with spaces.
 * Do NOT null–terminate it.
 */
    strncpy(dynstmt2, ”COMMIT    ”, 10);
    printf(”%.10s\n”, dynstmt2);

    EXEC SQL EXECUTE IMMEDIATE :dynstmt2;
/* Execute a VARCHAR to drop the table.  Set the .len field
 * to the length of the .arr field.
 */
    strcpy(dynstmt3.arr, ”DROP TABLE DYN1”);
    dynstmt3.len = strlen(dynstmt3.arr);
    puts((char *) dynstmt3.arr);

    EXEC SQL EXECUTE IMMEDIATE :dynstmt3;

/* Commit any outstanding changes and disconnect from Oracle. */
    EXEC SQL COMMIT RELEASE;

    puts(”\nHave a good day!\n”);

    return 0;
}


void
dyn_error(msg)
char *msg;
{
/* This is the Oracle error handler.
 * Print diagnostic text containing the error message,
 * current SQL statement, and location of error.
 */
    printf(”\n%.*s\n”,
       sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
    printf(”in \”%.*s...\”\n”,
        oraca.orastxt.orastxtl, oraca.orastxt.orastxtc);
    printf(”on line %d of %.*s.\n\n”,
        oraca.oraslnr, oraca.orasfnm.orasfnml,
        oraca.orasfnm.orasfnmc);

/* Disable Oracle error checking to avoid an infinite loop
 * should another error occur within this routine as a
 * result of the rollback.
 */
    EXEC SQL WHENEVER SQLERROR CONTINUE;

/* Roll back any pending changes and disconnect from Oracle. */
    EXEC SQL ROLLBACK RELEASE;

    exit(1);
}
```

Using Method 2

What Method 1 does in one step, Method 2 does in two. The dynamic SQL statement, which cannot be a query, is first PREPAREd (named and parsed), then EXECUTEd.

With Method 2, the SQL statement can contain placeholders for input host variables and indicator variables. You can PREPARE the SQL statement once, then EXECUTE it repeatedly using different values of the host variables. Furthermore, you need *not* rePREPARE the SQL statement after a COMMIT or ROLLBACK (unless you log off and reconnect).

Note that you can use EXECUTE for nonqueries with Method 4.

The syntax of the PREPARE statement follows:

```
EXEC SQL PREPARE statement_name
    FROM { :host_string | string_literal };
```

PREPARE parses the SQL statement and gives it a name.

The *statement_name* is an identifier used by the precompiler, *not* a host or program variable, and should not be declared in the Declare Section. It simply designates the PREPAREd statement you want to EXECUTE.

The syntax of the EXECUTE statement is

```
EXEC SQL EXECUTE statement_name [USING host_variable_list];
```

where *host_variable_list* stands for the following syntax:

```
:host_variable1[:indicator1] [, host_variable2[:indicator2], ...]
```

EXECUTE executes the parsed SQL statement, using the values supplied for each input host variable.

In the following example, the input SQL statement contains the placeholder *n*:

```
...
int emp_number    INTEGER;
char delete_stmt[120], search_cond[40];;
...
strcpy(delete_stmt, ”DELETE FROM EMP WHERE EMPNO = :n AND ”);
printf(”Complete the following statement’s search condition––\n”);
printf(”%s\n”, delete_stmt);
gets(search_cond);
strcat(delete_stmt, search_cond);

EXEC SQL PREPARE sql_stmt FROM :delete_stmt;
for (;;)
{
     printf(”Enter employee number: ”);
    gets(temp);
    emp_number = atoi(temp);
    if (emp_number == 0)
        break;
    EXEC SQL EXECUTE sql_stmt USING :emp_number;
}
...
```

With Method 2, you must know the datatypes of input host variables at precompile time. In the last example, *emp_number* was declared as an **int**. It could also have been declared as type **float**, or even a **char**, because Oracle supports all these datatype conversions to the internal Oracle NUMBER datatype.

The USING Clause

When the SQL statement is EXECUTEd, input host variables in the USING clause replace corresponding placeholders in the PREPAREd dynamic SQL statement.

Every placeholder in the PREPAREd dynamic SQL statement must correspond to a different host variable in the USING clause. So, if the same placeholder appears two or more times in the PREPAREd statement, each appearance must correspond to a host variable in the USING clause.

The names of the placeholders need not match the names of the host variables. However, the order of the placeholders in the PREPAREd dynamic SQL statement must match the order of corresponding host variables in the USING clause.

If one of the host variables in the USING clause is an array, all must be arrays.

To specify nulls, you can associate indicator variables with host variables in the USING clause. For more information, see page 4 – 3, “Using Indicator Variables”.

Sample Program: Dynamic SQL Method 2

The following program uses dynamic SQL Method 2 to insert two rows into the EMP table, then delete them. This program is available on–line in your demo directory, in the file *sample7.pc*.

```
/*
 *  sample7.pc: Dynamic SQL Method 2
 *
 *  This program uses dynamic SQL Method 2 to insert two rows into
 *  the EMP table, then delete them.
 */

#include <stdio.h>
```

```
#include <string.h>

#define USERNAME ”SCOTT”
#define PASSWORD ”TIGER”

/* Include the SQL Communications Area, a structure through
 * which ORACLE makes runtime status information such as error
 * codes, warning flags, and diagnostic text available to the
 * program.
 */
#include <sqlca.h>

/* Include the ORACLE Communications Area, a structure through
 * which ORACLE makes additional runtime status information
 * available to the program.
 */
#include <oraca.h>

/* The ORACA=YES option must be specified to enable use of
 * the ORACA.
 */
EXEC ORACLE OPTION (ORACA=YES);

char    *username = USERNAME;
char    *password = PASSWORD;
VARCHAR  dynstmt[80];
int      empno   = 1234;
int      deptno1 = 97;
int      deptno2 = 99;


/* Handle SQL runtime errors. */
void dyn_error();


main()
{
/* Call dyn_error() whenever an error occurs
 * processing an embedded SQL statement.
 */
    EXEC SQL WHENEVER SQLERROR DO dyn_error(”Oracle error”);

/* Save text of current SQL statement in the ORACA if an
 * error occurs.
 */
    oraca.orastxtf = ORASTFERR;

/* Connect to Oracle. */
```

```
    EXEC SQL CONNECT :username IDENTIFIED BY :password;
    puts(”\nConnected to Oracle.\n”);

/* Assign a SQL statement to the VARCHAR dynstmt.  Both
 * the array and the length parts must be set properly.
 * Note that the statement contains two host–variable
 * placeholders, v1 and v2, for which actual input
 * host variables must be supplied at EXECUTE time.
 */
    strcpy(dynstmt.arr,
        ”INSERT INTO EMP (EMPNO, DEPTNO) VALUES (:v1, :v2)”);
    dynstmt.len = strlen(dynstmt.arr);

/* Display the SQL statement and its current input host
 * variables.
 */
    puts((char *) dynstmt.arr);
    printf(”   v1 = %d,  v2 = %d\n”, empno, deptno1);

/* The PREPARE statement associates a statement name with
 * a string containing a SQL statement.  The statement name
 * is a SQL identifier, not a host variable, and therefore
 * does not appear in the Declare Section.

 * A single statement name can be PREPAREd more than once,
 * optionally FROM a different string variable.
 */
    EXEC SQL PREPARE S FROM :dynstmt;

/* The EXECUTE statement executes a PREPAREd SQL statement
 * USING the specified input host variables, which are
 * substituted positionally for placeholders in the
 * PREPAREd statement.  For each occurrence of a
 * placeholder in the statement there must be a variable
 * in the USING clause.  That is, if a placeholder occurs
 * multiple times in the statement, the corresponding
 * variable must appear multiple times in the USING clause.
 * The USING clause can be omitted only if the statement
 * contains no placeholders.
 *
 * A single PREPAREd statement can be EXECUTEd more
 * than once, optionally USING different input host
 * variables.
 */
    EXEC SQL EXECUTE S USING :empno, :deptno1;

/* Increment empno and display new input host variables. */

    empno++;
    printf("   v1 = %d,  v2 = %d\n", empno, deptno2);

/* ReEXECUTE S to insert the new value of empno and a
 * different input host variable, deptno2.
 * A rePREPARE is unnecessary.
 */
    EXEC SQL EXECUTE S USING :empno, :deptno2;

/* Assign a new value to dynstmt. */

    strcpy(dynstmt.arr,
        "DELETE FROM EMP WHERE DEPTNO = :v1 OR DEPTNO = :v2");
    dynstmt.len = strlen(dynstmt.arr);

/* Display the new SQL statement and its current input host
 * variables.
 */
    puts((char *) dynstmt.arr);
    printf("   v1 = %d,    v2 = %d\n", deptno1, deptno2);

/* RePREPARE S FROM the new dynstmt. */

    EXEC SQL PREPARE S FROM :dynstmt;

/* EXECUTE the new S to delete the two rows previously
 * inserted.
 */
    EXEC SQL EXECUTE S USING :deptno1, :deptno2;

/* Commit any pending changes and disconnect from Oracle. */

    EXEC SQL COMMIT RELEASE;
    puts("\nHave a good day!\n");
    exit(0);
}


void
dyn_error(msg)
char *msg;
{
/* This is the ORACLE error handler.
 * Print diagnostic text containing error message,
 * current SQL statement, and location of error.
 */
    printf("\n%s", msg);
    printf("\n%.*s\n",
        sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
    printf("in \"%.*s...\"\n",
        oraca.orastxt.orastxtl, oraca.orastxt.orastxtc);
    printf(”on line %d of %.*s.\n\n”,
        oraca.oraslnr, oraca.orasfnm.orasfnml,
        oraca.orasfnm.orasfnmc);

/* Disable ORACLE error checking to avoid an infinite loop
 * should another error occur within this routine.
 */
    EXEC SQL WHENEVER SQLERROR CONTINUE;

/* Roll back any pending changes and
 * disconnect from Oracle.
 */
    EXEC SQL ROLLBACK RELEASE;
    exit(1);
}
```

Using Method 3

Method 3 is similar to Method 2 but combines the PREPARE statement with the statements needed to define and manipulate a cursor. This allows your program to accept and process queries. In fact, if the dynamic SQL statement is a query, you *must* use Method 3 or 4.

For Method 3, the number of columns in the query select list and the number of placeholders for input host variables must be known at precompile time. However, the names of database objects such as tables and columns need not be specified until run time. Names of database objects cannot be host variables. Clauses that limit, group, and sort query results (such as WHERE, GROUP BY, and ORDER BY) can also be specified at run time.

With Method 3, you use the following sequence of embedded SQL statements:

```
PREPARE statement_name FROM { :host_string | string_literal };
DECLARE cursor_name CURSOR FOR statement_name;
OPEN cursor_name [USING host_variable_list];
FETCH cursor_name INTO host_variable_list;
CLOSE cursor_name;
```

Now let’s look at what each statement does.

PREPARE

PREPARE parses the dynamic SQL statement and gives it a name. In the following example, PREPARE parses the query stored in the character string *select_stmt* and gives it the name *sql_stmt*:

```
char select_stmt[132] =
     ”SELECT MGR, JOB FROM EMP WHERE SAL < :salary”;
EXEC SQL PREPARE sql_stmt FROM :select_stmt;
```

Commonly, the query WHERE clause is input from a terminal at run time or is generated by the application.

The identifier *sql_stmt* is *not* a host or program variable, but must be unique. It designates a particular dynamic SQL statement.

DECLARE

DECLARE defines a cursor by giving it a name and associating it with a specific query. Continuing our example, DECLARE defines a cursor named *emp_cursor* and associates it with *sql_stmt*, as follows:

```
EXEC SQL DECLARE emp_cursor CURSOR FOR sql_stmt;
```

The identifiers *sql_stmt* and *emp_cursor* are *not* host or program variables, but must be unique. If you declare two cursors using the same statement name, the precompiler considers the two cursor names synonymous.

For example, if you execute the statements

```
EXEC SQL PREPARE sql_stmt FROM :select_stmt;
EXEC SQL DECLARE emp_cursor FOR sql_stmt;
EXEC SQL PREPARE sql_stmt FROM :delete_stmt;
EXEC SQL DECLARE dept_cursor FOR sql_stmt;
```

when you OPEN *emp_cursor*, you will process the dynamic SQL statement stored in *delete_stmt*, not the one stored in *select_stmt*.

OPEN

OPEN allocates an Oracle cursor, binds input host variables, and executes the query, identifying its active set. OPEN also positions the cursor on the first row in the active set and zeroes the rows–processed count kept by the third element of *sqlerrd* in the SQLCA. Input host variables in the USING clause replace corresponding placeholders in the PREPAREd dynamic SQL statement.

In our example, OPEN allocates *emp_cursor* and assigns the host variable *salary* to the WHERE clause, as follows:

```
EXEC SQL OPEN emp_cursor USING :salary;
```

FETCH

FETCH returns a row from the active set, assigns column values in the select list to corresponding host variables in the INTO clause, and advances the cursor to the next row. If there are no more rows, FETCH returns the “no data found” Oracle error code to *sqlca.sqlcode*.

In our example, FETCH returns a row from the active set and assigns the values of columns MGR and JOB to host variables *mgr_number* and *job_title*, as follows:

```
EXEC SQL FETCH emp_cursor INTO :mgr_number, :job_title;
```

CLOSE

CLOSE disables the cursor. Once you CLOSE a cursor, you can no longer FETCH from it.

In our example, CLOSE disables *emp_cursor*, as follows:

```
EXEC SQL CLOSE emp_cursor;
```

Sample Program: Dynamic SQL Method 3

The following program uses dynamic SQL Method 3 to retrieve the names of all employees in a given department from the EMP table. This program is available on–line in your demo directory, in the file *sample8.pc*

```
/*
 *  sample8.pc:  Dynamic SQL Method 3
 *
 *  This program uses dynamic SQL Method 3 to retrieve the names
 *  of all employees in a given department from the EMP table.
 */

#include <stdio.h>
#include <string.h>

#define USERNAME ”SCOTT”
#define PASSWORD ”TIGER”

/* Include the SQL Communications Area, a structure through
 * which ORACLE makes runtime status information such as error
 * codes, warning flags, and diagnostic text available to the
 * program. Also include the ORACA.
 */
#include <sqlca.h>
#include <oraca.h>

/* The ORACA=YES option must be specified to enable use of
 * the ORACA.
 */
EXEC ORACLE OPTION (ORACA=YES);
```

```
char    *username = USERNAME;
char    *password = PASSWORD;
VARCHAR  dynstmt[80];
VARCHAR  ename[10];
int      deptno = 10;

void dyn_error();


main()
{
/* Call dyn_error() function on any error in
 * an embedded SQL statement.
 */
    EXEC SQL WHENEVER SQLERROR DO dyn_error(”Oracle error”);

/* Save text of SQL current statement in the ORACA if an
 * error occurs.
 */
    oraca.orastxtf = ORASTFERR;

/* Connect to Oracle. */

    EXEC SQL CONNECT :username IDENTIFIED BY :password;
    puts(”\nConnected to Oracle.\n”);

/* Assign a SQL query to the VARCHAR dynstmt.  Both the
 * array and the length parts must be set properly.  Note
 * that the query contains one host-variable placeholder,
 * v1, for which an actual input host variable must be
 * supplied at OPEN time.
 */
    strcpy(dynstmt.arr,
        ”SELECT ename FROM emp WHERE deptno = :v1”);
    dynstmt.len = strlen(dynstmt.arr);

/* Display the SQL statement and its current input host
 * variable.
 */
    puts((char *) dynstmt.arr);
    printf(”   v1 = %d\n”, deptno);
    printf(”\nEmployee\n”);
    printf(”--------\n”);

/* The PREPARE statement associates a statement name with
 * a string containing a SELECT statement.  The statement
 * name is a SQL identifier, not a host variable, and
 * therefore does not appear in the Declare Section.
 * A single statement name can be PREPAREd more than once,
 * optionally FROM a different string variable.
 */
    EXEC SQL PREPARE S FROM :dynstmt;

/* The DECLARE statement associates a cursor with a
 * PREPAREd statement.  The cursor name, like the statement
 * name, does not appear in the Declare Section.

 * A single cursor name can not be DECLAREd more than once.
 */
    EXEC SQL DECLARE C CURSOR FOR S;

/* The OPEN statement evaluates the active set of the
 * PREPAREd query USING the specified input host variables,
 * which are substituted positionally for placeholders in
 * the PREPAREd query.  For each occurrence of a
 * placeholder in the statement there must be a variable
 * in the USING clause.  That is, if a placeholder occurs
 * multiple times in the statement, the corresponding
 * variable must appear multiple times in the USING clause.

 * The USING clause can be omitted only if the statement
 * contains no placeholders.  OPEN places the cursor at the
 * first row of the active set in preparation for a FETCH.

 * A single DECLAREd cursor can be OPENed more than once,
 * optionally USING different input host variables.
 */
    EXEC SQL OPEN C USING :deptno;

/* Break the loop when all data have been retrieved. */

    EXEC SQL WHENEVER NOT FOUND DO break;

/* Loop until the NOT FOUND condition is detected. */

    for (;;)
    {
/* The FETCH statement places the select list of the
 * current row into the variables specified by the INTO
 * clause, then advances the cursor to the next row.  If
 * there are more select–list fields than output host
 * variables, the extra fields will not be returned.
 * Specifying more output host variables than select–list
 * fields results in an ORACLE error.
 */
        EXEC SQL FETCH C INTO :ename;
/* Null-terminate the array before output. */
        ename.arr[ename.len] = ’\0’;
        puts((char *) ename.arr);
        }

/* Print the cumulative number of rows processed by the
 * current SQL statement.
 */
    printf(”\nQuery returned %d row%s.\n\n”, sqlca.sqlerrd[2],
        (sqlca.sqlerrd[2] == 1) ? ”” : ”s”);

/* The CLOSE statement releases resources associated with
 * the cursor.
 */
    EXEC SQL CLOSE C;

/* Commit any pending changes and disconnect from Oracle. */
    EXEC SQL COMMIT RELEASE;
    puts(”Sayonara.\n”);
    exit(0);
}

void
dyn_error(msg)
char *msg;
{
    printf(”\n%s”, msg);
    sqlca.sqlerrm.sqlerrmc[sqlca.sqlerrm.sqlerrml] = ’\0’;
    oraca.orastxt.orastxtc[oraca.orastxt.orastxtl] = ’\0’;
    oraca.orasfnm.orasfnmc[oraca.orasfnm.orasfnml] = ’\0’;
    printf(”\n%s\n”, sqlca.sqlerrm.sqlerrmc);
    printf(”in \”%s...\”\n”, oraca.orastxt.orastxtc);
    printf(”on line %d of %s.\n\n”, oraca.oraslnr,
       oraca.orasfnm.orasfnmc);

/* Disable ORACLE error checking to avoid an infinite loop
 * should another error occur within this routine.
 */
    EXEC SQL WHENEVER SQLERROR CONTINUE;

/* Release resources associated with the cursor. */
    EXEC SQL CLOSE C;

/* Roll back any pending changes and disconnect from Oracle. */
    EXEC SQL ROLLBACK RELEASE;
    exit(1);
}
```

Using Method 4

This section gives an overview of dynamic SQL Method 4. For complete details, see Chapter 12.

There is a kind of dynamic SQL statement that your program cannot process using Method 3. When the number of select–list items or placeholders for input host variables is unknown until run time, your program must use a descriptor. A *descriptor* is an area of memory used by your program and Oracle to hold a complete description of the variables in a dynamic SQL statement.

Recall that for a multirow query, you FETCH selected column values INTO a list of declared output host variables. If the select list is unknown, the host–variable list cannot be established at precompile time by the INTO clause. For example, you know the following query returns two column values:

```
SELECT ename, empno FROM emp WHERE deptno = :dept_number;
```

However, if you let the user define the select list, you might not know how many column values the query will return.

Need for the SQLDA

To process this kind of dynamic query, your program must issue the DESCRIBE SELECT LIST command and declare a data structure called the SQL Descriptor Area (SQLDA). Because it holds descriptions of columns in the query select list, this structure is also called a *select descriptor.*

Likewise, if a dynamic SQL statement contains an unknown number of placeholders for input host variables, the host–variable list cannot be established at precompile time by the USING clause.

To process the dynamic SQL statement, your program must issue the DESCRIBE BIND VARIABLES command and declare another kind of SQLDA called a *bind descriptor* to hold descriptions of the placeholders for input host variables. (Input host variables are also called *bind variables*.)

If your program has more than one active SQL statement (it might have OPENed two or more cursors, for example), each statement must have its own SQLDA(s). However, non–concurrent cursors can reuse SQLDAs. There is no set limit on the number of SQLDAs in a program.

The DESCRIBE Statement

DESCRIBE initializes a descriptor to hold descriptions of select–list items or input host variables.

If you supply a select descriptor, the DESCRIBE SELECT LIST statement examines each select–list item in a PREPAREd dynamic query to determine its name, datatype, constraints, length, scale, and precision. It then stores this information in the select descriptor.

If you supply a bind descriptor, the DESCRIBE BIND VARIABLES statement examines each placeholder in a PREPAREd dynamic SQL statement to determine its name, length, and the datatype of its associated input host variable. It then stores this information in the bind descriptor for your use. For example, you might use placeholder names to prompt the user for the values of input host variables.

What Is a SQLDA?

A SQLDA is a host–program data structure that holds descriptions of select–list items or input host variables.

SQLDA variables are *not* defined in the Declare Section.

The select SQLDA contains the following information about a query select list:

- maximum number of columns that can be DESCRIBEd
- actual number of columns found by DESCRIBE
- addresses of buffers to store column values
- lengths of column values
- datatypes of column values
- addresses of indicator–variable values
- addresses of buffers to store column names
- sizes of buffers to store column names
- current lengths of column names

The bind SQLDA contains the following information about the input host variables in a SQL statement:

- maximum number of placeholders that can be DESCRIBEd
- actual number of placeholders found by DESCRIBE
- addresses of input host variables
- lengths of input host variables
- datatypes of input host variables
- addresses of indicator variables

- addresses of buffers to store placeholder names
- sizes of buffers to store placeholder names
- current lengths of placeholder names
- addresses of buffers to store indicator–variable names
- sizes of buffers to store indicator–variable names
- current lengths of indicator–variable names

The SQLDA structure and variable names are defined in Chapter 12

Implementing Method 4

With Method 4, you generally use the following sequence of embedded SQL statements:

```
EXEC SQL PREPARE statement_name
     FROM { :host_string | string_literal };
EXEC SQL DECLARE cursor_name CURSOR FOR statement_name;
EXEC SQL DESCRIBE BIND VARIABLES FOR statement_name
     INTO bind_descriptor_name;
EXEC SQL OPEN cursor_name
     [USING DESCRIPTOR bind_descriptor_name];
EXEC SQL DESCRIBE [SELECT LIST FOR] statement_name
     INTO select_descriptor_name;
EXEC SQL FETCH cursor_name
     USING DESCRIPTOR select_descriptor_name;
EXEC SQL CLOSE cursor_name;
```

However, select and bind descriptors need not work in tandem. So, if the number of columns in a query select list is known, but the number of placeholders for input host variables is unknown, you can use the Method 4 OPEN statement with the following Method 3 FETCH statement:

```
EXEC SQL FETCH emp_cursor INTO host_variable_list;
```

Conversely, if the number of placeholders for input host variables is known, but the number of columns in the select list is unknown, you can use the Method 3 OPEN statement

```
EXEC SQL OPEN cursor_name [USING host_variable_list];
```

with the Method 4 FETCH statement.

Note that EXECUTE can be used for nonqueries with Method 4.

To see how these statements allow your program to process dynamic SQL statements using descriptors, refer to page NO TAG.

Restriction

In dynamic SQL Method 4, you cannot bind a host array to a PL/SQL procedure with a parameter of type “table.”

Using the DECLARE STATEMENT Statement

With Methods 2, 3, and 4, you might need to use the statement

```
EXEC SQL [AT db_name] DECLARE statement_name STATEMENT;
```

where *db_name* and *statement_name* are identifiers used by the precompiler, *not* host or program variables.

DECLARE STATEMENT declares the name of a dynamic SQL statement so that the statement can be referenced by PREPARE, EXECUTE, DECLARE CURSOR, and DESCRIBE. It is required if you want to execute the dynamic SQL statement at a non–default database. An example using Method 2 follows:

```
EXEC SQL AT remote_db DECLARE sql_stmt STATEMENT;
EXEC SQL PREPARE sql_stmt FROM :dyn_string;
EXEC SQL EXECUTE sql_stmt;
```

In the example, *remote_db* tells Oracle where to EXECUTE the SQL statement.

With Methods 3 and 4, DECLARE STATEMENT is also required if the DECLARE CURSOR statement precedes the PREPARE statement, as shown in the following example:

```
EXEC SQL DECLARE sql_stmt STATEMENT;
EXEC SQL DECLARE emp_cursor CURSOR FOR sql_stmt;
EXEC SQL PREPARE sql_stmt FROM :dyn_string;
```

The usual sequence of statements is

```
EXEC SQL PREPARE sql_stmt FROM :dyn_string;
EXEC SQL DECLARE emp_cursor CURSOR FOR sql_stmt;
```

Using Host Arrays

The use of host arrays in static SQL and dynamic SQL is similar. For example, to use input host arrays with dynamic SQL Method 2, simply use the syntax

```
EXEC SQL EXECUTE statement_name USING host_array_list;
```

where *host_array_list* contains one or more host arrays.

Similarly, to use input host arrays with Method 3, use the following syntax:

```
OPEN cursor_name USING host_array_list;
```

To use output host arrays with Method 3, use the following syntax:

```
FETCH cursor_name INTO host_array_list;
```

With Method 4, you must use the optional FOR clause to tell Oracle the size of your input or output host array. This is described in Chapter 12.

Using PL/SQL

The Pro*C Precompiler treats a PL/SQL block like a single SQL statement. So, like a SQL statement, a PL/SQL block can be stored in a string host variable or literal. When you store the PL/SQL block in the string, omit the keywords EXEC SQL EXECUTE, the keyword END–EXEC, and the ';' statement terminator.

However, there are two differences in the way the precompiler handles SQL and PL/SQL:

- The precompiler treats all PL/SQL host variables as *input* host variables whether they serve as input or output host variables (or both) inside the PL/SQL block.
- You cannot FETCH from a PL/SQL block because it might contain any number of SQL statements.

With Method 1

If the PL/SQL block contains no host variables, you can use Method 1 to EXECUTE the PL/SQL string in the usual way.

With Method 2

If the PL/SQL block contains a known number of input and output host variables, you can use Method 2 to PREPARE and EXECUTE the PL/SQL string in the usual way.

You must put *all* host variables in the USING clause. When the PL/SQL string is EXECUTEd, host variables in the USING clause replace corresponding placeholders in the PREPAREd string. Though the precompiler treats all PL/SQL host variables as input host variables, values are assigned correctly. Input (program) values are assigned to input host variables, and output (column) values are assigned to output host variables.

Every placeholder in the PREPAREd PL/SQL string must correspond to a host variable in the USING clause. So, if the same placeholder appears two or more times in the PREPAREd string, each appearance must correspond to a host variable in the USING clause.

With Method 3

Methods 2 and 3 are the same except that Method 3 allows FETCHing. Since you cannot FETCH from a PL/SQL block, just use Method 2 instead.

With Method 4

If the PL/SQL block contains an unknown number of input or output host variables, you must use Method 4.

To use Method 4, you set up one bind descriptor for all the input and output host variables. Executing DESCRIBE BIND VARIABLES stores information about input *and* output host variables in the bind descriptor. Because the precompiler treats all PL/SQL host variables as input host variables, executing DESCRIBE SELECT LIST has no effect.

⚠ **Warning:** In dynamic SQL Method 4, you cannot bind a host array to a PL/SQL procedure with a parameter of type “table.”

The use of bind descriptors with Method 4 is detailed in Chapter 12.

Caution

Do not use ANSI–style comments (– –) in a PL/SQL block that will be processed dynamically because end–of–line characters are ignored. As a result, ANSI–style comments extend to the end of the block, not just to the end of a line. Instead, use C–style comments (/* ... */).

CHAPTER

12

Implementing Dynamic SQL Method 4

This chapter shows you how to implement dynamic SQL Method 4, which lets your program accept or build dynamic SQL statements that contain a varying number of host variables. Subjects discussed include the following:

- meeting the special requirements of Method 4
- declaring the SQL Descriptor Area (SQLDA)
- using the SQLDA variables
- converting data
- coercing datatypes
- handling null/not null datatypes
- initializing and using descriptors

Note: For a discussion of dynamic SQL Methods 1, 2, and 3, and an overview of Method 4, see Chapter NO TAG.

Meeting the Special Requirements of Method 4

Before looking into the requirements of Method 4, you should feel comfortable with the terms *select–list item* and *placeholder*. Select–list items are the columns or expressions following the keyword SELECT in a query. For example, the following dynamic query contains three select–list items:

```
SELECT ename, job, sal + comm FROM emp WHERE deptno = 20
```

Placeholders are dummy bind variables that hold places in a SQL statement for actual bind variables. You do not declare placeholders, and can name them anything you like.

Placeholders for bind variables are most often used in the SET, VALUES, and WHERE clauses. For example, the following dynamic SQL statements each contain two placeholders:

```
INSERT INTO emp (empno, deptno) VALUES (:e, :d)
DELETE FROM dept WHERE deptno = :num OR loc = :loc
```

What Makes Method 4 Special?

Unlike Methods 1, 2, and 3, dynamic SQL Method 4 lets your program

- accept or build dynamic SQL statements that contain an unknown number of select–list items or placeholders, and
- take explicit control over datatype conversion between Oracle and C types

To add this flexibility to your program, you must give the Oracle runtime library additional information.

What Information Does Oracle Need?

The Pro*C Precompiler generates calls to Oracle for all executable dynamic SQL statements. If a dynamic SQL statement contains no select–list items or placeholders, Oracle needs no additional information to execute the statement. The following DELETE statement falls into this category:

```
DELETE FROM emp WHERE deptno = 30
```

However, most dynamic SQL statements contain select–list items or placeholders for bind variables, as does the following UPDATE statement:

```
UPDATE emp SET comm = :c WHERE empno = :e
```

To execute a dynamic SQL statement that contains placeholders for bind variables or select–list items, Oracle needs information about the program variables that hold the input (bind) values, and that will hold the FETCHed values when a query is executed. The information needed by Oracle is:

- the number of bind variables and select–list items
- the length of each bind variable and select–list item
- the datatype of each bind variable and select–list item
- the address of each bind variable, and of the output variable that will receive each select–list item

Where Is the Information Stored?

All the information Oracle needs about select–list items or placeholders for bind variables, except their values, is stored in a program data structure called the SQL Descriptor Area (SQLDA). The SQLDA struct is defined in the *sqlda.h* header file.

Descriptions of select–list items are stored in a *select descriptor*, and descriptions of placeholders for bind variables are stored in a *bind descriptor*.

The values of select–list items are stored in output variables; the values of bind variables are stored in input variables. You store the addresses of these variables in the select or bind SQLDA so that Oracle knows where to write output values and read input values.

How do values get stored in these data variables? Output values are FETCHed using a cursor, and input values are typically filled in by the program, usually from information entered interactively by the user.

How is the SQLDA Referenced?

The bind and select descriptors are usually referenced by pointer. A dynamic SQL program should declare a pointer to at least one bind descriptor, and a pointer to at least one select descriptor, in the following way:

```
EXEC SQL INCLUDE sqlda;

SQLDA *bind_dp;
SQLDA *select_dp;
```

You can then use the *sqlald()* function to allocate the descriptor, as follows:

```
bind_dp = sqlald(size, name_length, ind_name_length);
```

See the section "Allocating a SQLDA" on page 12 – 5 for detailed information about *sqlald()* and its parameters.

How is the Information Obtained?

You use the DESCRIBE statement to help obtain the information Oracle needs.

The DESCRIBE SELECT LIST statement examines each select–list item to determine its name, datatype, constraints, length, scale, and precision. It then stores this information in the select SQLDA for your use. For example, you might use select–list names as column headings in a printout. The total number of select–list items is also stored in the SQLDA by DESCRIBE.

The DESCRIBE BIND VARIABLES statement examines each placeholder to determine its name and length, then stores this information in an input buffer and bind SQLDA for your use. For example, you might use placeholder names to prompt the user for the values of bind variables.

Understanding the SQLDA

This section describes the SQLDA data structure in detail. You learn how to declare it, what variables it contains, how to initialize them, and how to use them in your program.

Purpose of the SQLDA

Method 4 is required for dynamic SQL statements that contain an unknown number of select–list items or placeholders for bind variables. To process this kind of dynamic SQL statement, your program must explicitly declare SQLDAs, also called *descriptors*. Each descriptor is a **struct** which you must copy or hardcode into your program.

A *select descriptor* holds descriptions of select–list items, and the addresses of output buffers where the names and values of select–list items are stored.

Note: The "name" of a select–list item can be a column name, a column alias, or the text of an expression such as *sal + comm*.

A *bind descriptor* holds descriptions of bind variables and indicator variables, and the addresses of input buffers where the names and values of bind variables and indicator variables are stored.

Multiple SQLDAs

If your program has more than one active dynamic SQL statement, each statement must have its own SQLDA(s). You can declare any number of SQLDAs with different names. For example, you might declare three select SQLDAs named *sel_desc1*, *sel_desc2*, and *sel_desc3*, so that you can FETCH from three concurrently OPEN cursors. However, non–concurrent cursors can reuse SQLDAs.

Declaring a SQLDA

To declare a SQLDA, include the *sqlda.h* header file. The contents of the SQLDA are

```
struct SQLDA
{
    long    N;          /* Descriptor size in number of entries */
    char  **V;        Ptr to Arr of addresses of main variables */
    long   *L;              /* Ptr to Arr of lengths of buffers */
    short  *T;                /* Ptr to Arr of types of buffers */
    short **I;      * Ptr to Arr of addresses of indicator vars */
    long    F;         /* Number of variables found by DESCRIBE */
    char  **S;          /* Ptr to Arr of variable name pointers */
    short  *M;       /* Ptr to Arr of max lengths of var. names */
    short  *C;    * Ptr to Arr of current lengths of var. names */
    char  **X;         /* Ptr to Arr of ind. var. name pointers */
    short  *Y;  /* Ptr to Arr of max lengths of ind. var. names */
    short  *Z;  /* Ptr to Arr of cur lengths of ind. var. names */
};
```

Allocating a SQLDA

After declaring a SQLDA, you allocate storage space for it with the *sqlald()* library function, using the syntax

```
descriptor_name = sqlald(max_vars, max_name, max_ind_name);
```

where:

max_vars Is the maximum number of select–list items or placeholders that the descriptor can describe.

max_name Is the maximum length of select–list or placeholder names.

max_ind_name Is the maximum length of indicator variable names, which are optionally appended to placeholder names. This parameter applies to bind descriptors only, so set it to zero when allocating a select descriptor.

Besides the descriptor, *sqlald()* allocates data buffers to which descriptor variables point. For more information about *sqlald()*, see the next section “Using the SQLDA Variables” and the section “Allocate Storage Space for the Descriptors” on page 12 – 18.

Figure 12 – 1 shows whether variables are set by *sqlald()* calls, DESCRIBE commands, FETCH commands, or program assignments.

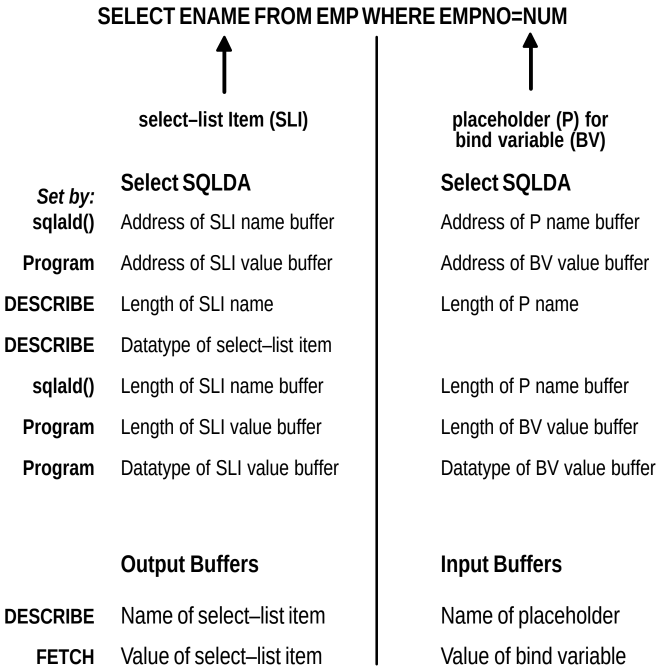

SELECT ENAME FROM EMP WHERE EMPNO=NUM
select–list Item (SLI)
placeholder (P) for bind variable (BV)
Set by:
Select SQLDA
Select SQLDA
sqlald() Address of SLI name buffer
Address of P name buffer
Program Address of SLI value buffer
Address of BV value buffer
DESCRIBE Length of SLI name
Length of P name
DESCRIBE Datatype of select–list item
sqlald() Length of SLI name buffer
Length of P name buffer
Program Length of SLI value buffer
Length of BV value buffer
Program Datatype of SLI value buffer
Datatype of BV value buffer
Output Buffers
Input Buffers
DESCRIBE Name of select–list item
Name of placeholder
FETCH Value of select–list item
Value of bind variable

Figure 12 – 1 How Variables Are Set

Using the SQLDA Variables

This section explains the purpose and use of each variable in the SQLDA.

The *N* Variable

N specifies the maximum number of select–list items or placeholders that can be DESCRIBEd. Thus, *N* determines the number of elements in the descriptor arrays.

Before issuing the optional DESCRIBE command, you must set *N* to the dimension of the descriptor arrays using the *sqlald()* library function. After the DESCRIBE, you must reset *N* to the actual number of variables DESCRIBEd, which is stored in the *F* variable.

The *V* Variable

V is a pointer to an array of addresses of data buffers that store select–list or bind–variable values.

When you allocate the descriptor, *sqlald()* zeros the elements *V[0]* through *V[N – 1]* in the array of addresses.

For select descriptors, you must allocate data buffers and set this array before issuing the FETCH command. The statement

```
EXEC SQL FETCH ... USING DESCRIPTOR ...
```

directs Oracle to store FETCHed select–list values in the data buffers to which *V[0]* through *V[N – 1]* point. Oracle stores the *i*th select–list value in the data buffer to which *V[i]* points.

For bind descriptors, you must set this array before issuing the OPEN command. The statement

```
EXEC SQL OPEN ... USING DESCRIPTOR ...
```

directs Oracle to execute the dynamic SQL statement using the bind–variable values to which *V[0]* through *V[N – 1]* point. Oracle finds the *i*th bind–variable value in the data buffer to which *V[i]* points.

The *L* Variable

L is a pointer to an array of lengths of select–list or bind–variable values stored in data buffers.

For select descriptors, DESCRIBE SELECT LIST sets the array of lengths to the maximum expected for each select–list item. However, you might want to reset some lengths before issuing a FETCH command. FETCH returns at most *n* characters, where *n* is the value of *L[i]* before the FETCH.

The format of the length differs among Oracle datatypes. For CHAR or VARCHAR2 select–list items, DESCRIBE SELECT LIST sets *L[i]* to the maximum length of the select–list item. For NUMBER select–list items, scale and precision are returned respectively in the low and next–higher bytes of the variable. You can use the library function *sqlprc()* to extract precision and scale values from *L[i]*. See the section "Extracting Precision and Scale" on page 12 – 13.

You must reset *L[i]* to the required length of the data buffer before the FETCH. For example, when coercing a NUMBER to a C **char** string, set *L[i]* to the precision of the number plus two for the sign and decimal point. When coercing a NUMBER to a C **float**, set *L[i]* to the length of **float**s on your system. For more information about the lengths of coerced datatypes, see the section "Converting Data" on page 12 – 10.

For bind descriptors, you must set the array of lengths before issuing the OPEN command. For example, you can use *strlen()* to get the lengths of bind–variable character strings entered by the user, then set the appropriate array elements.

Because Oracle accesses a data buffer indirectly, using the address stored in *V[i]*, it does not know the length of the value in that buffer. If you want to change the length Oracle uses for the *i*th select–list or bind–variable value, reset *L[i]* to the length you need. Each input or output buffer can have a different length.

The *T* Variable

T is a pointer to an array of datatype codes of select–list or bind–variable values. These codes determine how Oracle data is converted when stored in the data buffers addressed by elements of the *V* array. This topic is covered in the section “Converting Data” on page 12 – 10.

For select descriptors, DESCRIBE SELECT LIST sets the array of datatype codes to the *internal* datatype (CHAR, NUMBER, or DATE, for example) of the items in the select list.

Before FETCHing, you might want to reset some datatypes because the internal format of Oracle datatypes can be difficult to handle. For display purposes, it is usually a good idea to coerce the datatype of select–list values to VARCHAR2 or STRING. For calculations, you might want to coerce numbers from Oracle to C format. See the section “Coercing Datatypes” on page 12 – 12.

The high bit of *T[i]* is set to indicate the null / not null status of the *i*th select–list item. You must always clear this bit before issuing an OPEN or FETCH command. You use the library function *sqlnul()* to retrieve the datatype code and clear the null / not null bit. See the section “Handling Null / Not Null Datatypes” on page 12 – 15.

You should change the Oracle NUMBER internal datatype to an external datatype compatible with that of the C data buffer to which *V[i]* points.

For bind descriptors, DESCRIBE BIND VARIABLES sets the array of datatype codes to zeros. You must set the datatype code stored in each element before issuing the OPEN command. The code represents the external (C) datatype of the data buffer to which *V[i]* points. Often, bind–variable values are stored in character strings, so the datatype array elements are set to 1 (the VARCHAR2 datatype code). You can also use datatype code 5 (STRING).

To change the datatype of the *i*th select–list or bind–variable value, reset *T[i]* to the datatype you want.

The *I* Variable

I is a pointer to an array of addresses of data buffers that store indicator–variable values.

You must set the elements *I[0]* through *I[N – 1]* in the array of addresses.

For select descriptors, you must set the array of addresses before issuing the FETCH command. When Oracle executes the statement

```
EXEC SQL FETCH ... USING DESCRIPTOR ...
```

if the *i*th returned select–list value is null, the indicator–variable value to which *I[i]* points is set to –1. Otherwise, it is set to zero (the value is not null) or a positive integer (the value was truncated).

For bind descriptors, you must set the array of addresses and associated indicator variables before issuing the OPEN command. When Oracle executes the statement

```
EXEC SQL OPEN ... USING DESCRIPTOR ...
```

the data buffer to which *I[i]* points determines whether the *i*th bind variable has a null value. If the value of an indicator variable is –1, the value of its associated bind variable is null.

The *F* Variable

F is the actual number of select–list items or placeholders found by DESCRIBE.

F is set by DESCRIBE. If *F* is less than zero, DESCRIBE has found too many select–list items or placeholders for the allocated size of the descriptor. For example, if you set *N* to 10 but DESCRIBE finds 11 select–list items or placeholders, *F* is set to –11. This feature lets you dynamically reallocate a larger storage area for select–list items or placeholders if necessary.

The *S* Variable

S is a pointer to an array of addresses of data buffers that store select–list or placeholder names as they appear in dynamic SQL statements.

You use *sqlald()* to allocate the data buffers and store their addresses in the *S* array.

DESCRIBE directs Oracle to store the name of the *i*th select–list item or placeholder in the data buffer to which *S[i]* points.

The *M* Variable

M is a pointer to an array of maximum lengths of data buffers that store select–list or placeholder names. The buffers are addressed by elements of the *S* array.

When you allocate the descriptor, *sqlald()* sets the elements *M[0]* through *M[N – 1]* in the array of maximum lengths. When stored in the data buffer to which *S[i]* points, the *i*th name is truncated to the length in *M[i]* if necessary.

The *C* Variable

C is a pointer to an array of current lengths of select–list or placeholder names.

DESCRIBE sets the elements *C[0]* through *C[N – 1]* in the array of current lengths. After a DESCRIBE, the array contains the number of characters in each select–list or placeholder name.

The *X* Variable

X is a pointer to an array of addresses of data buffers that store indicator–variable names. You can associate indicator–variable *values* with select–list items and bind variables. However, you can associate indicator–variable *names* only with bind variables. So, *X* applies only to bind descriptors.

You use *sqlald()* to allocate the data buffers and store their addresses in the *X* array.

DESCRIBE BIND VARIABLES directs Oracle to store the name of the *i*th indicator variable in the data buffer to which *X[i]* points.

The *Y* Variable

Y is a pointer to an array of maximum lengths of data buffers that store indicator–variable names. Like *X*, *Y* applies only to bind descriptors.

You use *sqlald()* to set the elements *Y[0]* through *Y[N – 1]* in the array of maximum lengths. When stored in the data buffer to which *X[i]* points, the *i*th name is truncated to the length in *Y[i]* if necessary.

The *Z* Variable

Z is a pointer to an array of current lengths of indicator–variable names. Like *X* and *Y*, *Z* applies only to bind descriptors.

DESCRIBE BIND VARIABLES sets the elements *Z[0]* through *Z[N – 1]* in the array of current lengths. After a DESCRIBE, the array contains the number of characters in each indicator–variable name.

Some Preliminaries

You need a working knowledge of the following subjects to implement dynamic SQL Method 4:

- converting data
- coercing datatypes
- handling null/not null datatypes

Converting Data

This section provides more detail about the *T* (datatype) descriptor array. In host programs that use neither datatype equivalencing nor dynamic SQL Method 4, the conversion between Oracle internal and external datatypes is determined at precompile time. By default, the precompiler assigns a specific external datatype to each host variable in the Declare Section. For example, the precompiler assigns the INTEGER external datatype to host variables of type **int**.

However, Method 4 lets you control data conversion and formatting. You specify conversions by setting datatype codes in the *T* descriptor array.

Internal Datatypes

Internal datatypes specify the formats used by Oracle to store column values in database tables, as well as the formats used to represent pseudocolumn values.

When you issue a DESCRIBE SELECT LIST command, Oracle returns the internal datatype code for each select–list item to the *T* descriptor array. For example, the datatype code for the *i*th select–list item is returned to *T[i]*.

Table 12 – 1 shows the Oracle internal datatypes and their codes:

Oracle Internal Datatype	Code
VARCHAR2	1
NUMBER	2
LONG	8
ROWID	11
DATE	12
RAW	23
LONG RAW	24
CHARACTER (or CHAR)	96
MLSLABEL	106

Table 12 – 1 Oracle Internal Datatypes

External Datatypes

External datatypes specify the formats used to store values in input and output host variables.

The DESCRIBE BIND VARIABLES command sets the *T* array of datatype codes to zeros. So, you must reset the codes *before* issuing the OPEN command. The codes tell Oracle which external datatypes to expect for the various bind variables. For the *i*th bind variable, reset *T[i]* to the external datatype you want.

Table 12 – 1 shows the Oracle external datatypes and their codes, as well as the C datatype normally used with each external datatype.

External Datatype	Code	C Datatype
VARCHAR2	1	char[n]
NUMBER	2	char[n] ($n \leq 22$)

Table 12 – 2 Oracle External Datatypes and Datatype Codes

External Datatype	Code	C Datatype
INTEGER	3	int
FLOAT	4	float
STRING	5	char[n+1]
VARNUM	6	char[n] (n $\leq$ 22)
DECIMAL	7	float
LONG	8	char[n]
VARCHAR	9	char[n+2]
ROWID	11	char[n]
DATE	12	char[n]
VARRAW	15	char[n]
RAW	23	unsigned char[n]
LONG RAW	24	unsigned char[n]
UNSIGNED	68	unsigned int
DISPLAY	91	char[n]
LONG VARCHAR	94	char[n+4]
LONG VARRAW	95	unsigned char[n+4]
CHAR	96	char[n]
CHARF	96	char[n]
CHARZ	97	char[n+1]
MLSLABEL	106	char[n]

Table 12 – 2 Oracle External Datatypes and Datatype Codes

For more information about the Oracle datatypes and their formats, see Chapter 3 in this guide, and the *Oracle7 Server SQL Reference.*

Coercing Datatypes

For a select descriptor, DESCRIBE SELECT LIST can return any of the Oracle internal datatypes. Often, as in the case of character data, the internal datatype corresponds exactly to the external datatype you want to use. However, a few internal datatypes map to external datatypes that can be difficult to handle. So, you might want to reset some elements in the *T* descriptor array. For example, you might want to reset NUMBER values to FLOAT values, which correspond to **float** values in C. Oracle does any necessary conversion between internal and external datatypes at FETCH time. So, be sure to reset the datatypes *after* the DESCRIBE SELECT LIST but *before* the FETCH.

For a bind descriptor, DESCRIBE BIND VARIABLES does *not* return the datatypes of bind variables, only their number and names. Therefore, you must explicitly set the *T* array of datatype codes to tell

Oracle the external datatype of each bind variable. Oracle does any necessary conversion between external and internal datatypes at OPEN time.

When you reset datatype codes in the *T* descriptor array, you are “coercing datatypes.” For example, to coerce the *i*th select–list value to STRING, you use the following statement:

```
/* Coerce select-list value to STRING. */
select_des->T[i] = 5;
```

When coercing a NUMBER select–list value to STRING for display purposes, you must also extract the precision and scale bytes of the value and use them to compute a maximum display length. Then, before the FETCH, you must reset the appropriate element of the *L* (length) descriptor array to tell Oracle the buffer length to use. See the section “Extracting Precision and Scale” on page 12 – 13

For example, if DESCRIBE SELECT LIST finds that the *i*th select–list item is of type NUMBER, and you want to store the returned value in a C variable declared as **float**, simply set *T[i]* to 4 and *L[i]* to the length of **float**s on your system.

Caution

In some cases, the internal datatypes that DESCRIBE SELECT LIST returns might not suit your purposes. Two examples of this are DATE and NUMBER. When you DESCRIBE a DATE select–list item, Oracle returns the datatype code 12 to the *T* descriptor array. Unless you reset the code before the FETCH, the date value is returned in its 7–byte internal format. To get the date in character format (DD–MON–YY), you can change the datatype code from 12 to 1 (VARCHAR2) or 5 (STRING), and increase the *L* value from 7 to 9 or 10.

Similarly, when you DESCRIBE a NUMBER select–list item, Oracle returns the datatype code 2 to the *T* array. Unless you reset the code before the FETCH, the numeric value is returned in its internal format, which is probably not what you want. So, change the code from 2 to 1 (VARCHAR2), 3 (INTEGER), 4 (FLOAT), 5 (STRING) or some other appropriate datatype.

Extracting Precision and Scale

The library function *sqlprc()* extracts precision and scale. Normally, it is used after the DESCRIBE SELECT LIST, and its first argument is *L[i]*. You call *sqlprc()* using the following syntax:

```
sqlprc(long *length, int *precision, int *scale);
```

> **Note:** See your platform–specific *sqlcpr.h* file for the correct prototype for your platform.

where:

length Is a pointer to a long integer variable that stores the length of an Oracle NUMBER value; the length is stored in *L[i]*. The scale and precision of the value are stored respectively in the low and next–higher bytes.

precision Is a pointer to an integer variable that returns the *precision* of the NUMBER value. Precision is the number of significant digits. It is set to zero if the select–list item refers to a NUMBER of unspecified size. In this case, because the size is unspecified, you might want to assume the maximum precision (38).

scale Is a pointer to an integer variable that returns the *scale* of the NUMBER value. Scale specifies where rounding will occur. For example, a scale of 2 means the value is rounded to the nearest hundredth (3.456 becomes 3.46); a scale of –3 means the number is rounded to the nearest thousand (3456 becomes 3000).

When the scale is negative, add its absolute value to the length. For example, a precision of 3 and scale of –2 allow for numbers as large as 99900.

The following example shows how *sqlprc()* is used to compute maximum display lengths for NUMBER values that will be coerced to STRING:

```
/* Declare variables for the function call. */
sqlda         *select_des;  /* pointer to select descriptor */
int            prec;        /* precision                    */
int            scal;        /* scale                        */
extern void sqlprc();  /* Declare library function. */
/* Extract precision and scale. */
sqlprc( &(select_des–>L[i]), &prec, &scal);
/* Allow for maximum size of NUMBER. */
if (prec == 0)
     prec = 38;
/* Allow for possible decimal point and sign. */
select_des–>L[i] = prec + 2;
/* Allow for negative scale. */
if (scal < 0)
     select_des–>L[i] += –scal;
```

Notice that the first argument in this function call points to the *i*th element in the array of lengths, and that all three parameters are addresses.

The *sqlprc()* function returns zero as the precision and scale values for certain SQL datatypes. The *sqlpr2()* function is similar to *sqlprc()*, having the same argument list, and returning the same values, except in the cases of these SQL datatypes:

SQL Datatype	***Binary Precision***	***Scale***
FLOAT	126	–127
FLOAT(N)	N (range is 1 to 126)	–127
REAL	63	–127
DOUBLE PRECISION	126	–127

Handling Null/Not Null Datatypes

For every select–list column (not expression), DESCRIBE SELECT LIST returns a null / not null indication in the datatype array *T* of the select descriptor. If the *i*th select–list column is constrained to be not null, the high–order bit of *T[i]* is clear; otherwise, it is set.

Before using the datatype in an OPEN or FETCH statement, if the null / not null bit is set, you must clear it. (Never set the bit.)

You can use the library function *sqlnul()* to find out if a column allows nulls, and to clear the datatype's null / not null bit. You call *sqlnul()* using the syntax

```
sqlnul(unsigned short *value_type,
unsigned short *type_code, int *null_status);
```

where:

value_type Is a pointer to an unsigned short integer variable that stores the datatype code of a select–list column; the datatype is stored in *T[i]*.

type_code Is a pointer to an unsigned short integer variable that returns the datatype code of the select–list column with the high–order bit cleared.

null_status Is a pointer to an integer variable that returns the null status of the select–list column. 1 means the column allows nulls; 0 means it does not.

The following example shows how to use *sqlnul()*:

```
/* Declare variables for the function call. */
sqlda  *select_des;      /* pointer to select descriptor */
unsigned short   dtype;  /* datatype without null bit    */
int   nullok;            /* 1 = null, 0 = not null       */
extern void sqlnul();    /* Declare library function.    */
/* Find out whether column is not null. */
sqlnul(&select_des–>T[i], &dtype, &nullok);
```

```
if (nullok)
{
     /* Nulls are allowed. */
     ...
     /* Clear the null/not null bit. */
     sqlnul(&(select_des–>T[i]), &(select_des–>T[i]), &nullok);
}
```

Notice that the first and second arguments in the second call to the *sqlnul()* function point to the *i*th element in the array of datatypes, and that all three parameters are addresses.

The Basic Steps

Method 4 can be used to process *any* dynamic SQL statement. In the coming example, a query is processed so you can see how both input and output host variables are handled.

To process the dynamic query, our example program takes the following steps:

1. Declare a host string in the Declare Section to hold the query text.
2. Declare select and bind SQLDAs.
3. Allocate storage space for the select and bind descriptors.
4. Set the maximum number of select–list items and placeholders that can be DESCRIBEd.
5. Put the query text in the host string.
6. PREPARE the query from the host string.
7. DECLARE a cursor FOR the query.
8. DESCRIBE the bind variables INTO the bind descriptor.
9. Reset the number of placeholders to the number actually found by DESCRIBE.
10. Get values and allocate storage for the bind variables found by DESCRIBE.
11. OPEN the cursor USING the bind descriptor.
12. DESCRIBE the select list INTO the select descriptor.
13. Reset the number of select–list items to the number actually found by DESCRIBE.

14. Reset the length and datatype of each select–list item for display purposes.

15. FETCH a row from the database INTO the allocated data buffers pointed to by the select descriptor.

16. Process the select–list values returned by FETCH.

17. Deallocate storage space used for the select–list items, placeholders, indicator variables, and descriptors.

18. CLOSE the cursor.

Note: Some of these steps are unnecessary if the dynamic SQL statement contains a known number of select–list items or placeholders.

A Closer Look at Each Step

This section discusses each step in detail. Also, at the end of this chapter is a commented, full–length program illustrating Method 4.

With Method 4, you use the following sequence of embedded SQL statements:

```
EXEC SQL PREPARE statement_name
     FROM { :host_string | string_literal };
EXEC SQL DECLARE cursor_name CURSOR FOR statement_name;
EXEC SQL DESCRIBE BIND VARIABLES FOR statement_name
     INTO bind_descriptor_name;
EXEC SQL OPEN cursor_name
     [USING DESCRIPTOR bind_descriptor_name];
EXEC SQL DESCRIBE [SELECT LIST FOR] statement_name
     INTO select_descriptor_name;
EXEC SQL FETCH cursor_name
     USING DESCRIPTOR select_descriptor_name;
EXEC SQL CLOSE cursor_name;
```

If the number of select–list items in a dynamic query is known, you can omit DESCRIBE SELECT LIST and use the following Method 3 FETCH statement:

```
EXEC SQL FETCH cursor_name INTO host_variable_list;
```

Or, if the number of placeholders for bind variables in a dynamic SQL statement is known, you can omit DESCRIBE BIND VARIABLES and use the following Method 3 OPEN statement:

```
EXEC SQL OPEN cursor_name [USING host_variable_list];
```

Next, you see how these statements allow your host program to accept and process a dynamic SQL statement using descriptors.

Note: Several figures accompany the following discussion. To avoid cluttering the figures, it was necessary to do the following:

- confine descriptor arrays to 3 elements
- limit the maximum length of names to 5 characters
- limit the maximum length of values to 10 characters

Declare a Host String

Your program needs a host variable to store the text of the dynamic SQL statement. The host variable (*select_stmt* in our example) must be declared as a character string.

```
...
int      emp_number;
VARCHAR  emp_name[10];
VARCHAR  select_stmt[120];
float    bonus;
```

Declare the SQLDAs

In our example, instead of hardcoding the SQLDA data structure, you use INCLUDE to copy it into your program, as follows:

```
EXEC SQL INCLUDE sqlda;
```

Then, because the query might contain an unknown number of select–list items or placeholders for bind variables, you declare pointers to select and bind descriptors, as follows:

```
sqlda  *select_des;
sqlda  *bind_des;
```

Allocate Storage Space for the Descriptors

Recall that you allocate storage space for a descriptor with the *sqlald()* library function. The syntax, using ANSI C notation, is:

```
SQLDA *sqlald(int max_vars, size_t max_name, size_t max_ind_name);
```

The *sqlald()* function allocates the descriptor structure and the arrays addressed by the pointer variables *V*, *L*, *T*, and *I*.

If *max_name* is non–zero, arrays addressed by the pointer variables *S*, *M*, and *C* are allocated. If *max_ind_name* is non–zero, arrays addressed by the pointer variables *X*, *Y*, and *Z* are allocated. No space is allocated if *max_name* and *max_ind_name* are zero.

If *sqlald()* succeeds, it returns a pointer to the structure. If *sqlald()* fails, it returns a zero.

In our example, you allocate select and bind descriptors, as follows:

```
select_des = sqlald(3, (size_t) 5, (size_t) 0);
bind_des = sqlald(3, (size_t) 5, (size_t) 4);
```

For select descriptors, always set *max_ind_name* to zero so that no space is allocated for the array addressed by *X*.

Set the Maximum Number to DESCRIBE

Next, you set the maximum number of select–list items or placeholders that can be DESCRIBEd, as follows:

```
select_des–>N = 3;
bind_des–>N = 3;
```

Figure 12 – 2 and Figure 12 – 3 represent the resulting descriptors.

Note: In the select descriptor (Figure 12 – 2), the section for indicator–variable names is crossed out to show that it is not used.

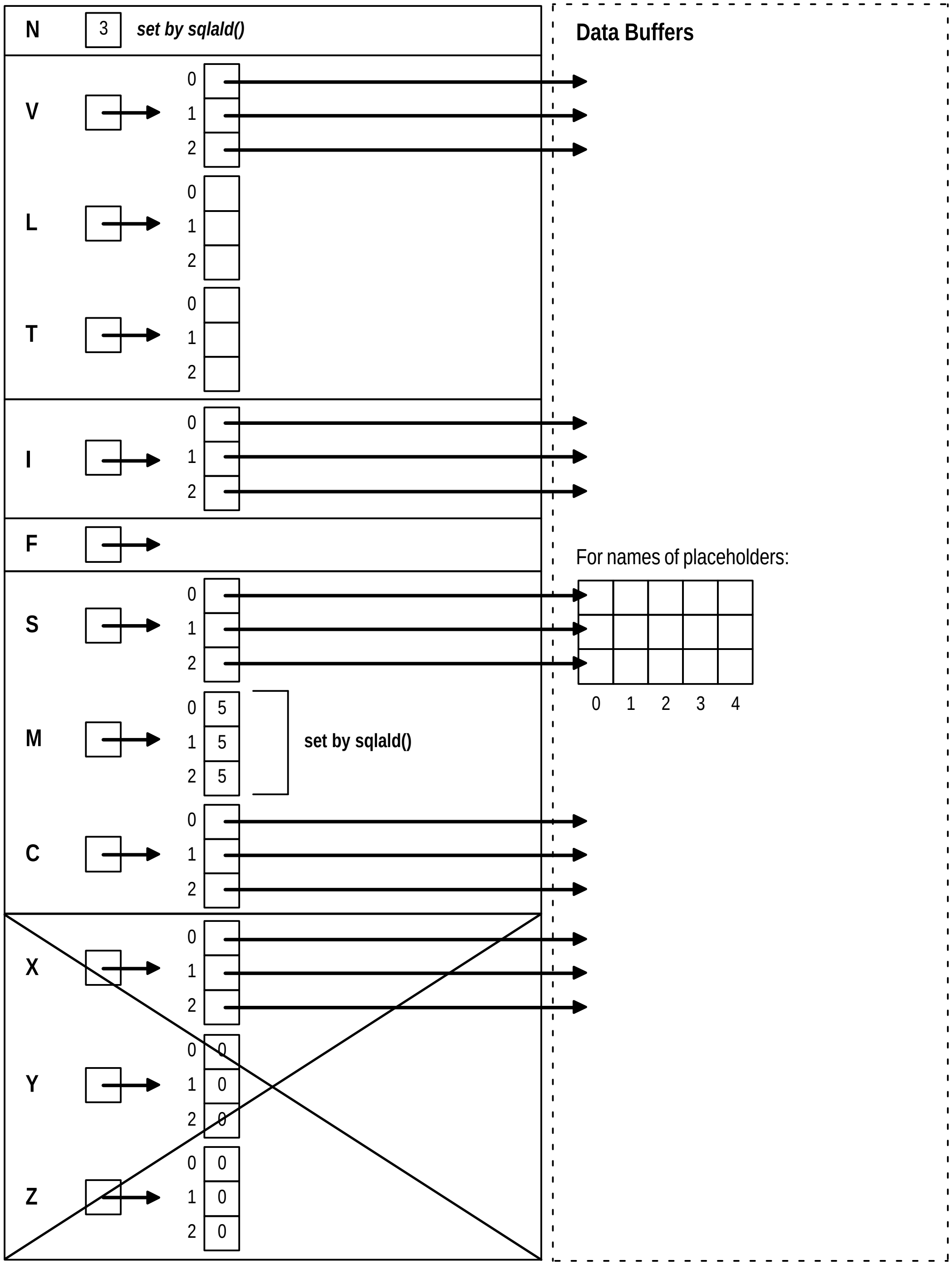


Figure 12 – 2 Initialized Select Descriptor

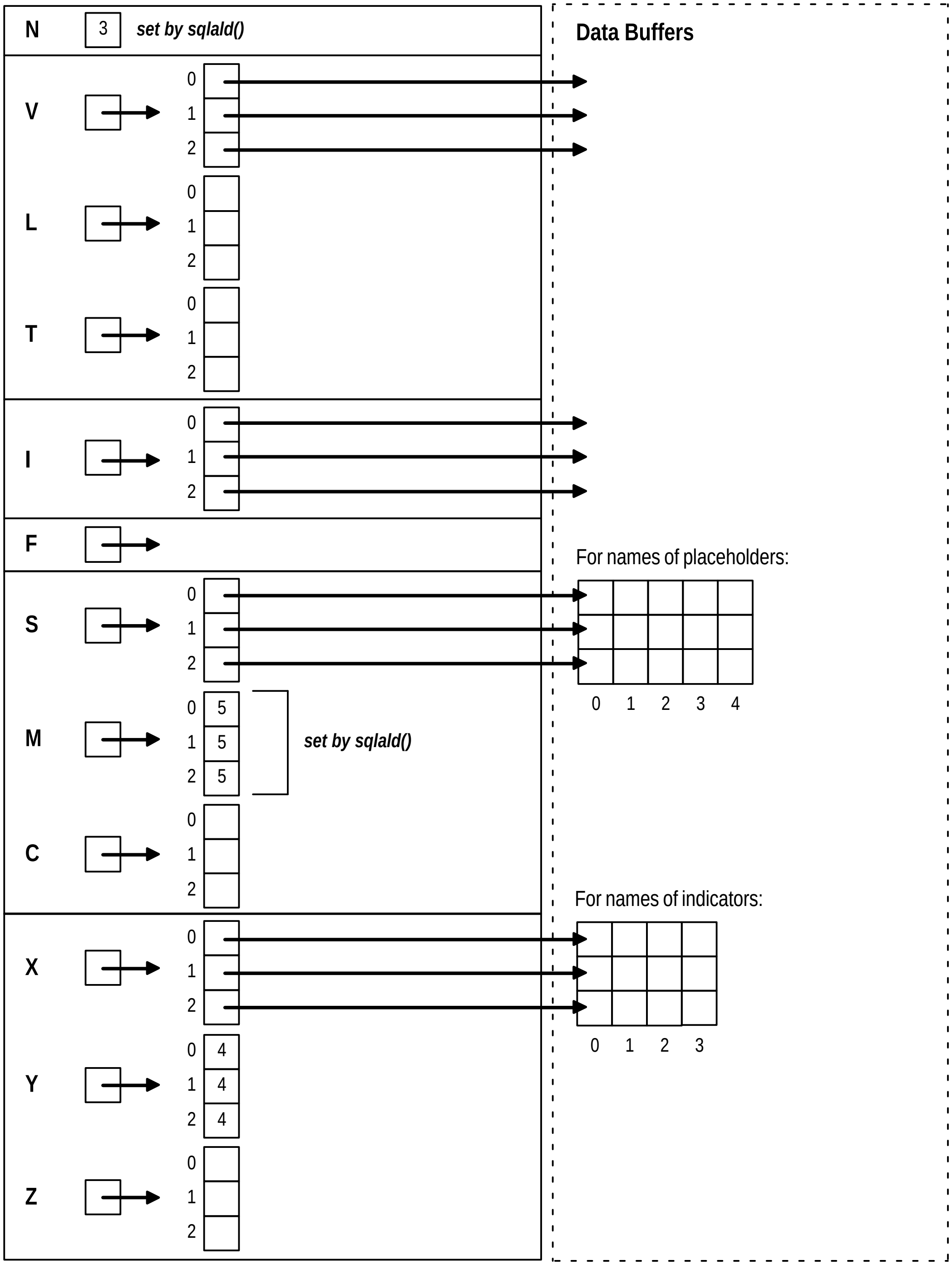


Figure 12 – 3 Initialized Bind Descriptor

Put the Query Text in the Host String

Continuing our example, you prompt the user for a SQL statement, then store the input string in *select_stmt*, as follows:

```
printf("\n\nEnter SQL statement: ");
gets(select_stmt.arr);
select_stmt.len = strlen(select_stmt.arr);
```

We assume the user entered the following string:

```
"SELECT ename, empno, comm FROM emp WHERE comm < :bonus"
```

PREPARE the Query from the Host String

PREPARE parses the SQL statement and gives it a name. In our example, PREPARE parses the host string *select_stmt* and gives it the name *sql_stmt*, as follows:

```
EXEC SQL PREPARE sql_stmt FROM :select_stmt;
```

DECLARE a Cursor

DECLARE CURSOR defines a cursor by giving it a name and associating it with a specific SELECT statement.

To declare a cursor for *static* queries, you use the following syntax:

```
EXEC SQL DECLARE cursor_name CURSOR FOR SELECT ...
```

To declare a cursor for *dynamic* queries, the statement name given to the dynamic query by PREPARE is substituted for the static query. In our example, DECLARE CURSOR defines a cursor named *emp_cursor* and associates it with *sql_stmt*, as follows:

```
EXEC SQL DECLARE emp_cursor CURSOR FOR sql_stmt;
```

Note: You must declare a cursor for all dynamic SQL statements, not just queries. With non–queries, OPENing the cursor executes the dynamic SQL statement.

DESCRIBE the Bind Variables

DESCRIBE BIND VARIABLES puts descriptions of placeholders into a bind descriptor. In our example, DESCRIBE readies *bind_des*, as follows:

```
EXEC SQL DESCRIBE BIND VARIABLES FOR sql_stmt INTO bind_des;
```

Note that *bind_des* must *not* be prefixed with a colon.

The DESCRIBE BIND VARIABLES statement must follow the PREPARE statement but precede the OPEN statement.

Figure 12 – 4 shows the bind descriptor in our example after the DESCRIBE. Notice that DESCRIBE has set *F* to the actual number of placeholders found in the processed SQL statement.

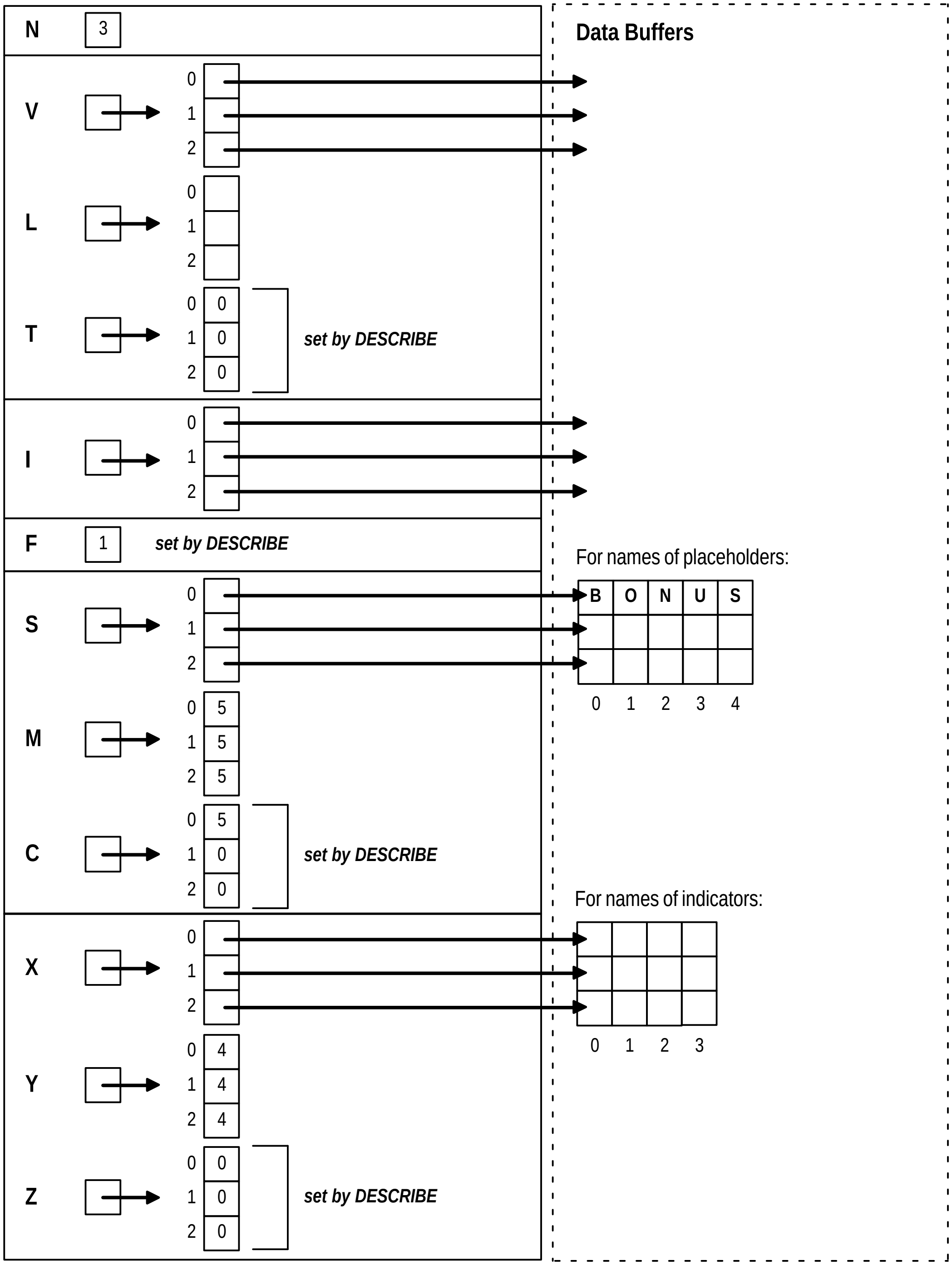


Figure 12 – 4 Bind Descriptor after the DESCRIBE

Reset Number of Placeholders

Next, you must reset the maximum number of placeholders to the number actually found by DESCRIBE, as follows:

```
bind_des–>N = bind_des–>F;
```

Get Values and Allocate Storage for Bind Variables

Your program must get values for the bind variables found in the SQL statement, and allocate memory for them. How the program gets the values is up to you. For example, they can be hardcoded, read from a file, or entered interactively.

In our example, a value must be assigned to the bind variable that replaces the placeholder *bonus* in the query WHERE clause. So, you choose to prompt the user for the value, then process it as follows:

```
for (i = 0; i < bind_des–>F; i++)
{
     printf(“\nEnter value of bind variable %.*s:\n? ”,
         (int) bind_des–>C[i], bind_des–>S[i]);
     gets(hostval);
     /* Set length of value. */
     bind_des–>L[i] = strlen(hostval);
     /* Allocate storage for value and null terminator. */
     bind_des–>V[i] = malloc(bind_des–>L[i] + 1);
     /* Allocate storage for indicator value. */
     bind_des–>I[i] = (unsigned short *) malloc(sizeof(short));
     /* Store value in bind descriptor. */
     strcpy(bind_des–>V[i], hostval);
     /* Set value of indicator variable. */
     *(bind_des–>I[i]) = 0;  /* or –1 if “null” is the value */
     /* Set datatype to STRING. */
     bind_des–>T[i] = 5;
}
```

Assuming that the user supplied a value of 625 for *bonus*, Figure 12 – 5 shows the resulting bind descriptor. Notice that the value is null–terminated.

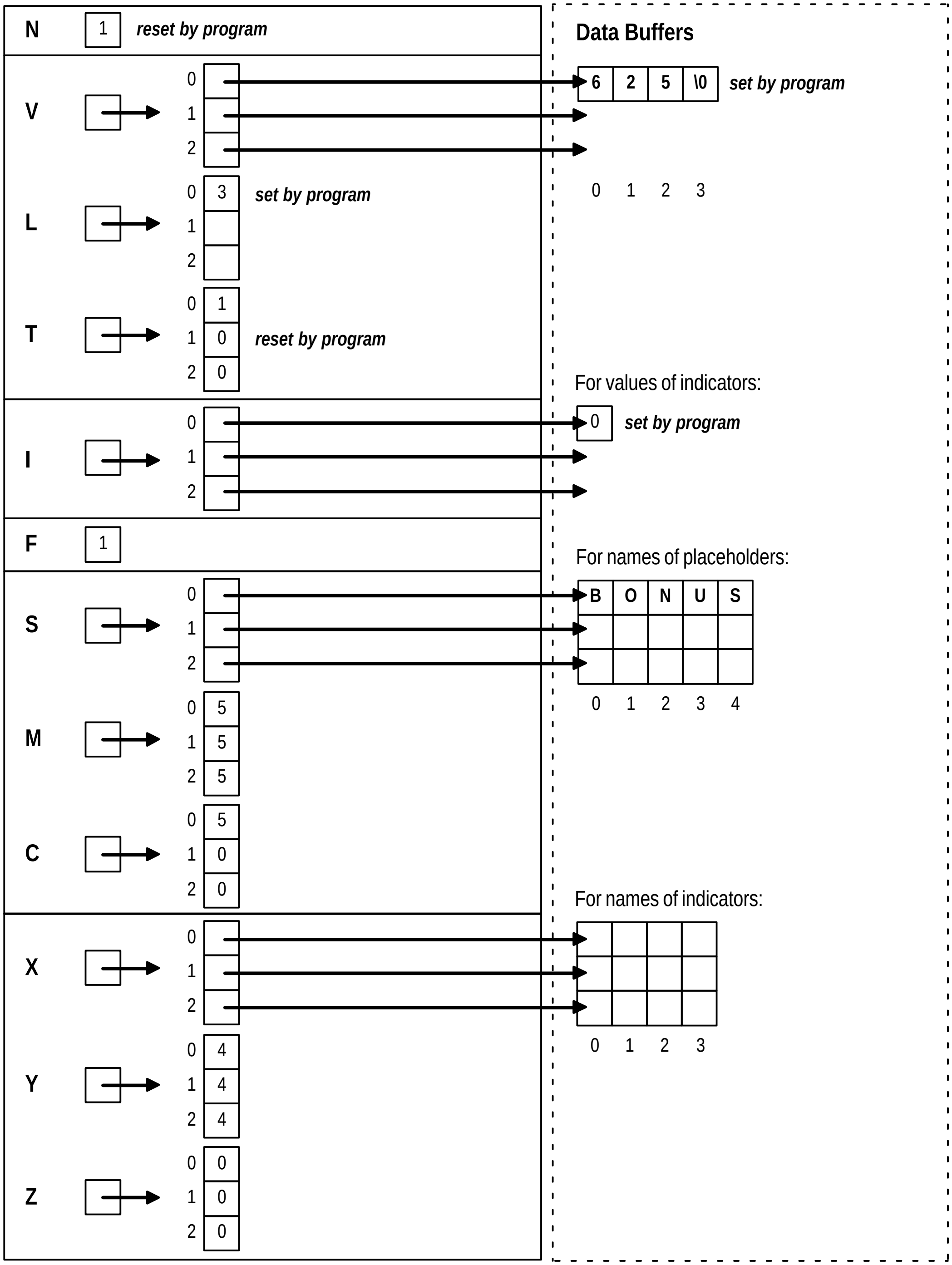


Figure 12 – 5 Bind Descriptor after Assigning Values

OPEN the Cursor

The OPEN statement used for dynamic queries is like that used for static queries except that the cursor is associated with a bind descriptor. Values determined at run time and stored in buffers addressed by elements of the bind descriptor arrays are used to evaluate the SQL statement. With queries, the values are also used to identify the active set.

In our example, OPEN associates *emp_cursor* with *bind_des*, as follows:

```
EXEC SQL OPEN emp_cursor USING DESCRIPTOR bind_des;
```

Remember, *bind_des* must *not* be prefixed with a colon.

Then, OPEN executes the SQL statement. With queries, OPEN also identifies the active set and positions the cursor at the first row.

DESCRIBE the Select List

If the dynamic SQL statement is a query, the DESCRIBE SELECT LIST statement must follow the OPEN statement but precede the FETCH statement.

DESCRIBE SELECT LIST puts descriptions of select–list items in a select descriptor. In our example, DESCRIBE readies *select_des*, as follows:

```
EXEC SQL DESCRIBE SELECT LIST FOR sql_stmt INTO select_des;
```

Accessing the Oracle data dictionary, DESCRIBE sets the length and datatype of each select–list value.

Figure 12 – 6 shows the select descriptor in our example after the DESCRIBE. Notice that DESCRIBE has set *F* to the actual number of items found in the query select list. If the SQL statement is not a query, *F* is set to zero.

Also notice that the NUMBER lengths are not usable yet. For columns defined as NUMBER, you must use the library function *sqlprc()* to extract precision and scale. See the section “Coercing Datatypes” on page 12 – 12.

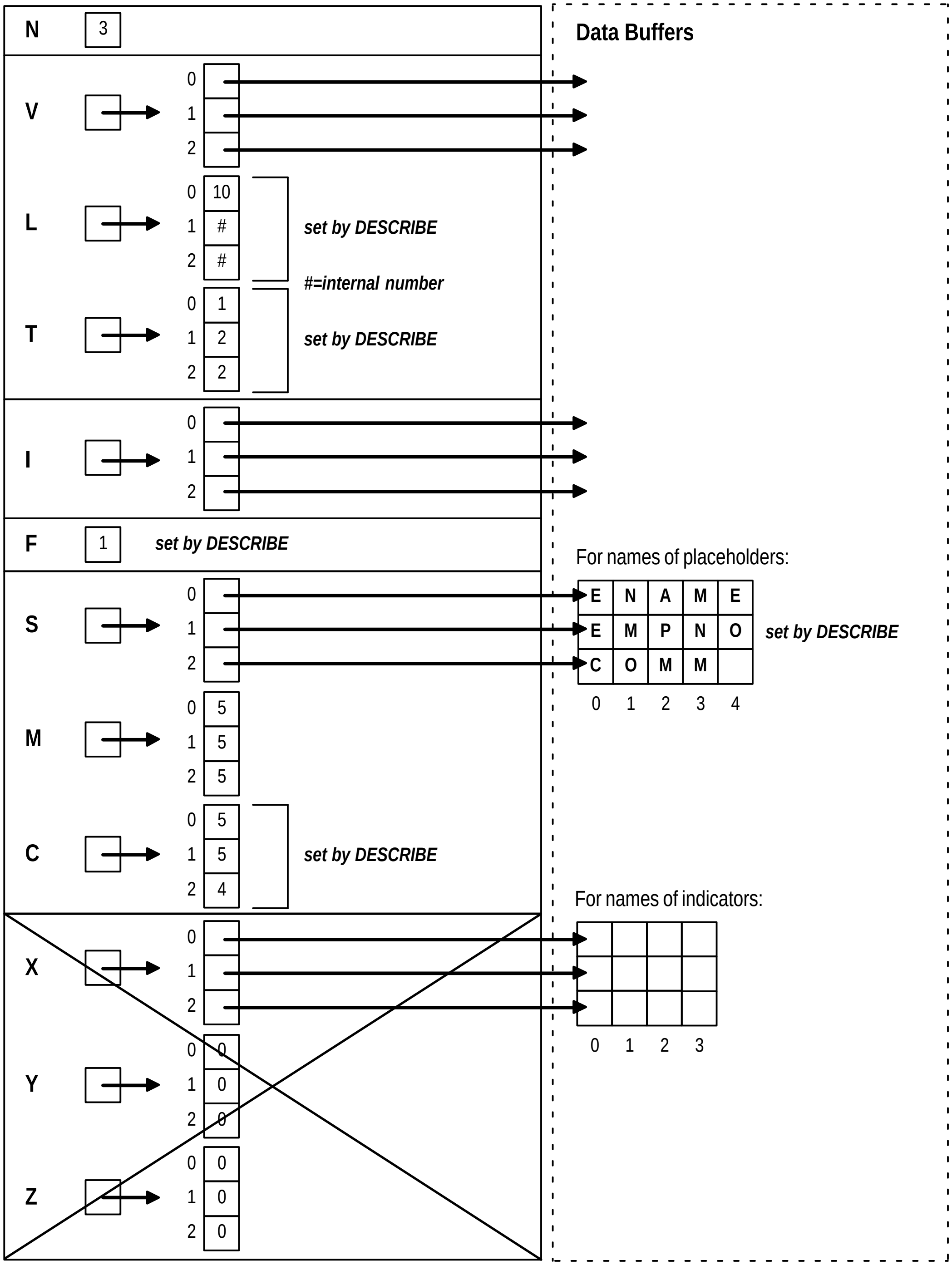


Figure 12 – 6 Select Descriptor after the DESCRIBE

Reset Number of Select–List Items

Next, you must reset the maximum number of select–list items to the number actually found by DESCRIBE, as follows:

```
select_des–>N = select_des–>F;
```

Reset Length/Datatype of Each Select–list Item

In our example, before FETCHing the select–list values, you allocate storage space for them using the library function *malloc()*. You also reset some elements in the length and datatype arrays for display purposes.

```
for (i=0; i<select_des–>F; i++)
{
     /* Clear null bit. */
     sqlnul(&(select_des–>T[i], &(select_des–>T[i], &nullok)));
     /* Reset length if necessary. */
     switch(select_des–>T[i])
     {
         case  1: break;
         case  2: sqlprc(&select_des–>L[i], &prec, &scal);
                  if (prec == 0) prec = 40;
                  select_des–>L[i] = prec + 2;
                  if (scal < 0) select_des–>L[i] += –scal;
                  break;
         case  8: select_des–>L[i] = 240;
                  break;
         case 11: select_des–>L[i] = 18;
                  break;
         case 12: select_des–>L[i] = 9;
                  break;
         case 23: break;
         case 24: select_des–>L[i] = 240;
                  break;
     }
     /* Allocate storage for select–list value. */
     select_des–>V[i] = malloc(select_des–>L[i+1]);
     /* Allocate storage for indicator value. */
     select_des–>I[i] = (short *)malloc(sizeof(short *));
     /* Coerce all datatypes except LONG RAW to STRING. */
     if (select_des–>T[i] != 24) select_des–>T[i] = 5;
}
```

Figure 12 – 7 shows the resulting select descriptor. Notice that the NUMBER lengths are now usable and that all the datatypes are STRING. The lengths in *L[1]* and *L[2]* are 6 and 9 because we increased the DESCRIBEd lengths of 4 and 7 by 2 to allow for a possible sign and decimal point.

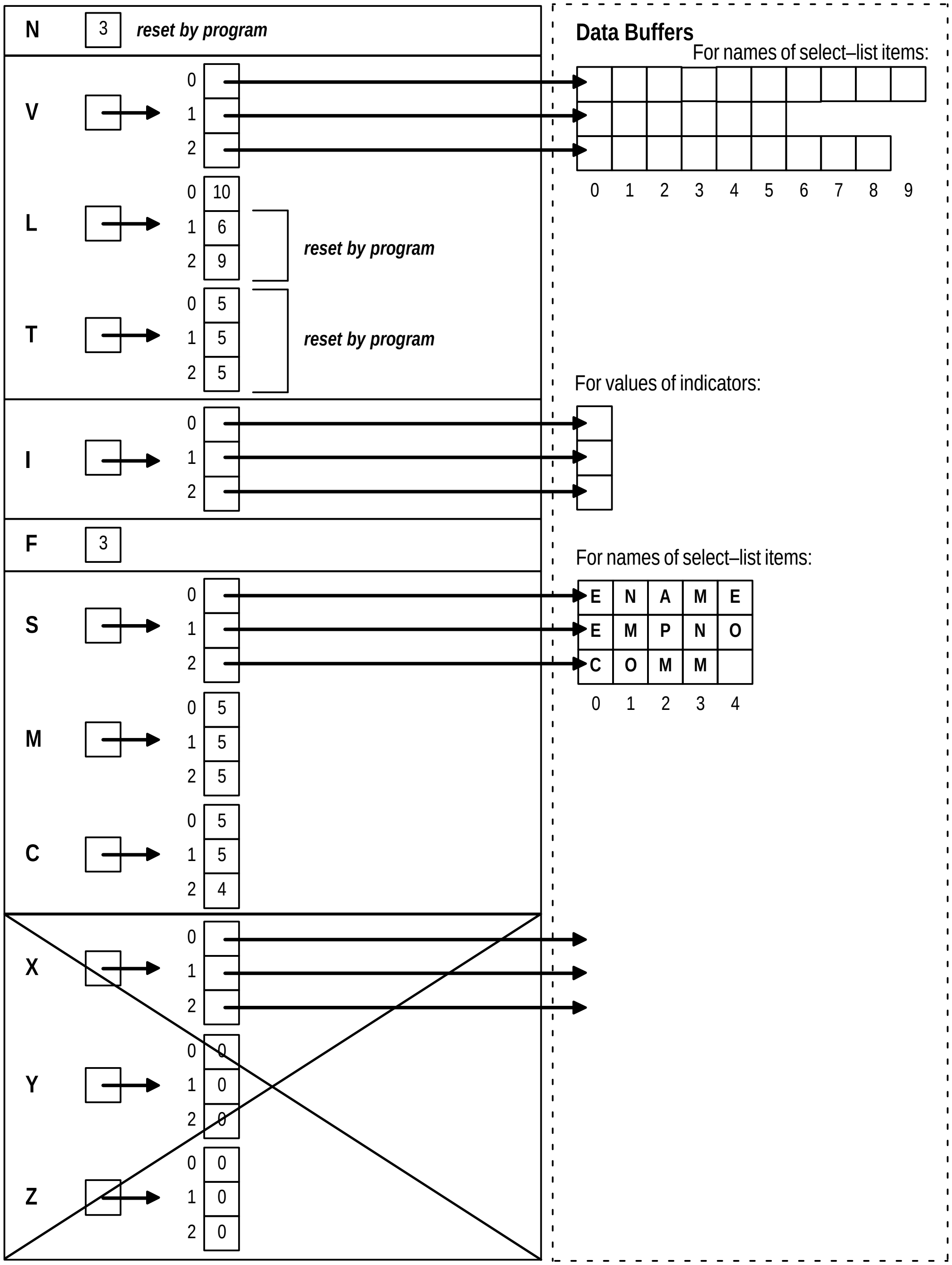


Figure 12 – 7 Select Descriptor before the FETCH

FETCH Rows from the Active Set

FETCH returns a row from the active set, stores select–list values in the data buffers, and advances the cursor to the next row in the active set. If there are no more rows, FETCH sets *sqlca.sqlcode* to the “no data found” Oracle error code. In our example, FETCH returns the values of columns ENAME, EMPNO, and COMM to *select_des*, as follows:

```
EXEC SQL FETCH emp_cursor USING DESCRIPTOR select_des;
```

Figure 12 – 8 shows the select descriptor in our example after the FETCH. Notice that Oracle has stored the select–list and indicator values in the data buffers addressed by the elements of *V* and *I*.

For output buffers of datatype 1, Oracle, using the lengths stored in the *L* array, left–justifies CHAR or VARCHAR2 data and right–justifies NUMBER data. For output buffer of type 5 (STRING), Oracle left–justifies and null terminates CHAR, VARCHAR2, and NUMBER data.

The value ‘MARTIN’ was retrieved from a VARCHAR2(10) column in the EMP table. Using the length in *L[0]*, Oracle left–justifies the value in a 10–byte field, filling the buffer.

The value 7654 was retrieved from a NUMBER(4) column and coerced to ’7654’. However, the length in *L[1]* was increased by 2 to allow for a possible sign and decimal point. So, Oracle left–justifies and null terminates the value in a 6–byte field.

The value 482.50 was retrieved from a NUMBER(7,2) column and coerced to ’482.50’. Again, the length in *L[2]* was increased by 2. So, Oracle left–justifies and null terminates the value in a 9–byte field.

Get and Process Select–List Values

After the FETCH, your program can process the returned values. In our example, values for columns ENAME, EMPNO, and COMM are processed.

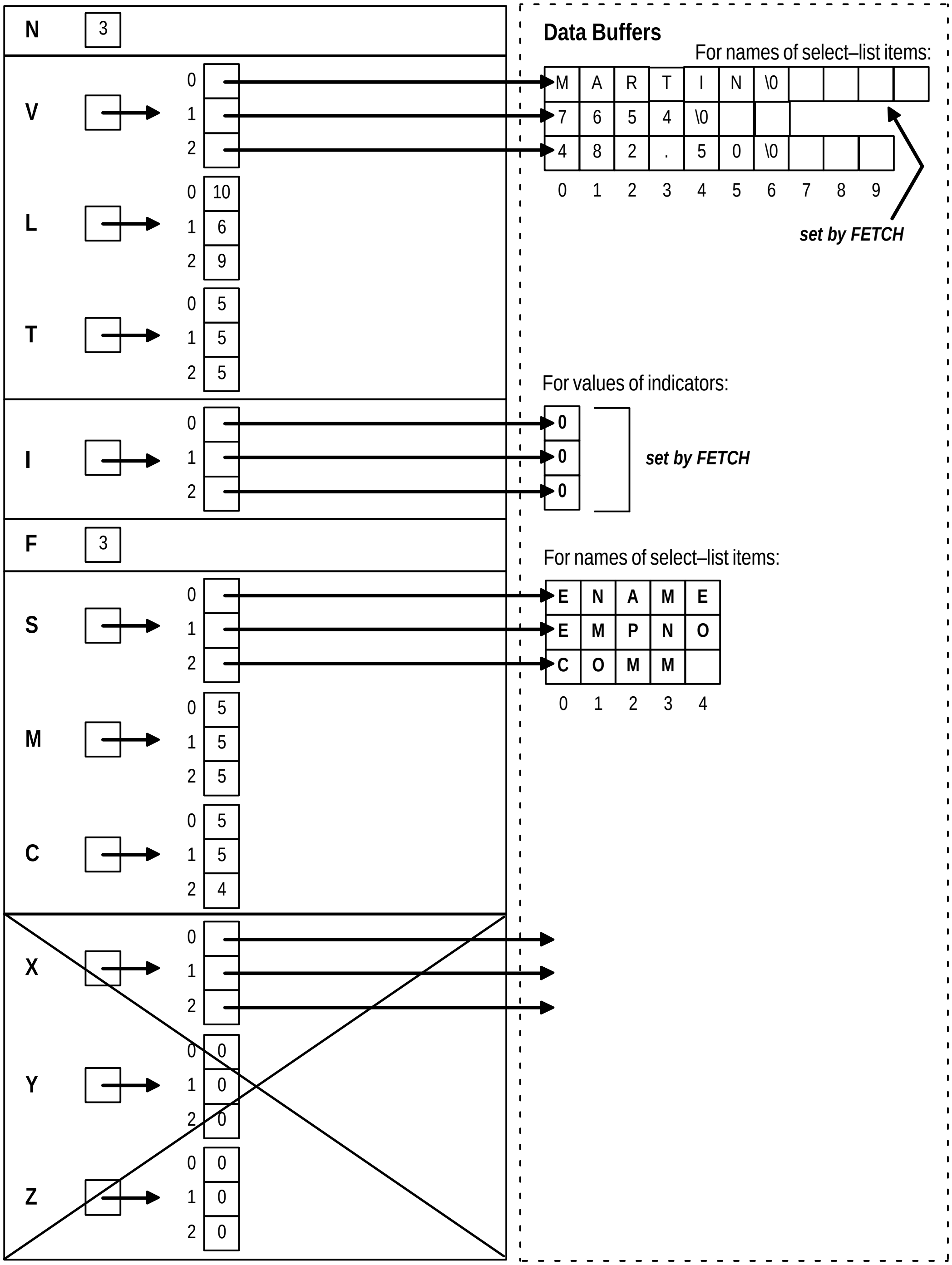


Figure 12 – 8 Selected Descriptor after the FETCH

Deallocate Storage

You use the *free()* library function to deallocate the storage space allocated by *malloc()*. The syntax is as follows:

```
free(char *pointer);
```

In our example, you deallocate storage space for the values of the select–list items, bind variables, and indicator variables, as follows:

```
for (i = 0; i < select_des–>F; i++)   /* for select descriptor */
{
     free(select_des–>V[i]);
     free(select_des–>I[i]);
}
for (i = 0; i < bind_des–>F; i++)   /* for bind descriptor */
{
     free(bind_des–>V[i]);
     free(bind_des–>I[i]);
}
```

You deallocate storage space for the descriptors themselves with the *sqlclu()* library function, using the following syntax:

```
sqlclu(descriptor_name);
```

The descriptor must have been allocated using *sqlald()*. Otherwise, the results are unpredictable.

In our example, you deallocate storage space for the select and bind descriptors as follows:

```
sqlclu(select_des);
sqlclu(bind_des);
```

CLOSE the Cursor

CLOSE disables the cursor. In our example, CLOSE disables *emp_cursor* as follows:

```
EXEC SQL CLOSE emp_cursor;
```

Using Host Arrays

To use input or output host arrays with Method 4, you must use the optional FOR clause to tell Oracle the size of your host array. For more information about the FOR clause, see Chapter 10.

You must set descriptor entries for the *i*th select–list item or bind variable using the syntax

```
V[i] = array_address;
L[i] = element_size;
```

where *array_address* is the address of the host array, and *element_size* is the size of one array element.

Then, you must use a FOR clause in the EXECUTE or FETCH statement (whichever is appropriate) to tell Oracle the number of array elements you want to process. This procedure is necessary because Oracle has no other way of knowing the size of your host array.

In the complete program example below, three input host arrays are used to INSERT rows into the EMP table. Note that EXECUTE can be used for Data Manipulation Language statements other than queries with Method 4.

```
#include <stdio.h>
#include <sqlca.h>
#include <sqlda.h>

#define NAME_SIZE    10
#define ARRAY_SIZE    5

/* connect string */
char *username = ”scott/tiger”;

char *sql_stmt =
”INSERT INTO emp (empno, ename, deptno) VALUES (:e, :n, :d)”;
int  array_size = ARRAY_SIZE;  /* must have a host variable too */

SQLDA   *binda;

char    names[ARRAY_SIZE][NAME_SIZE];
int     numbers[ARRAY_SIZE], depts[ARRAY_SIZE];

extern  SQLDA  *sqlald();
main()
{
    EXEC SQL WHENEVER SQLERROR GOTO sql_error;

/* Connect */
    EXEC SQL CONNECT :username;
    printf(”Connected.\n”);

/* Allocate the descriptors and set the N component.
   This must be done before the DESCRIBE. */
    binda = sqlald(3, ARRAY_SIZE, 0);
    binda–>N = 3;

/* Prepare and describe the SQL statement. */
    EXEC SQL PREPARE stmt FROM :sql_stmt;
    EXEC SQL DESCRIBE BIND VARIABLES FOR stmt INTO binda;

/* Initialize the descriptors. */
    binda–>V[0] = (char *) numbers;
    binda–>L[0] = (long) sizeof (int);
    binda–>T[0] = 3;
    binda–>I[0] = 0;

    binda–>V[1] = (char *) names;
    binda–>L[1] = (long) NAME_SIZE;
    binda–>T[1] = 1;
    binda–>I[1] = 0;

    binda–>V[2] = (char *) depts;
    binda–>L[2] = (long) sizeof (int);
    binda–>T[2] = 3;
    binda–>I[2] = 0;

/* Initialize the data buffers. */
    strcpy(&names[0] [0], ”ALLISON”);
    numbers[0] = 1014;
    depts[0] = 30;

    strcpy(&names[1] [0], ”TRUSDALE”);
    numbers[1] = 1015;
    depts[1] = 30;

    strcpy(&names[2] [0], ”FRAZIER”);
    numbers[2] = 1016;
    depts[2] = 30;

    strcpy(&names[3] [0], ”CARUSO”);
    numbers[3] = 1017;
    depts[3] = 30;

    strcpy(&names[4] [0], ”WESTON”);
    numbers[4] = 1018;
    depts[4] = 30;

/* Do the INSERT. */
    printf(”Adding to the Sales force...\n”);

    EXEC SQL FOR :array_size
    EXECUTE stmt USING DESCRIPTOR binda;

/*  Print rows–processed count. */
    printf(”%d rows inserted.\n\n”, sqlca.sqlerrd[2]);
    EXEC SQL COMMIT RELEASE;
    exit(0);

sql_error:
/* Print Oracle error message. */
    printf(”\n%.70s”, sqlca.sqlerrm.sqlerrmc);
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK RELEASE;
    exit(1);
}
```

Sample Program: Dynamic SQL Method 4

This program shows the basic steps required to use dynamic SQL with Method 4. After connecting to Oracle, the program allocates memory for the descriptors using *sqlald()*, prompts the user for a SQL statement, PREPAREs the statement, DECLAREs a cursor, checks for any bind variables using DESCRIBE BIND, OPENs the cursor, and DESCRIBEs any select–list items. If the input SQL statement is a query, the program FETCHes each row of data, then CLOSEs the cursor. This program is available on–line in the *demo* directory, in the file *sample10.pc*.

```
/****************************************************************
Sample Program 10:  Dynamic SQL Method 4

This program connects you to ORACLE using your username and
password, then prompts you for a SQL statement. You can enter
any legal SQL statement.

Use regular SQL syntax, not embedded SQL. Your statement will
be processed. If it is a query, the rows fetched are
displayed. You can enter multi–line statements. The limit is
1023 bytes.

This sample program only processes up to MAX_ITEMS bind
variables and MAX_ITEMS select–list items. MAX_ITEMS is
#defined to be 40.

****************************************************************/

#include <stdio.h>
#include <string.h>
#include <setjmp.h>

/* Maximum number of select–list items or bind variables. */
#define MAX_ITEMS         40

/* Maximum lengths of the _names_ of the
   select–list items or indicator variables. */
#define MAX_VNAME_LEN     30
#define MAX_INAME_LEN     30
```

```
#ifndef NULL
#define NULL  0
#endif

char *dml_commands[] = {”SELECT”, ”select”, ”INSERT”, ”insert”,
                        ”UPDATE”, ”update”, ”DELETE”, ”delete”};

EXEC SQL BEGIN DECLARE SECTION;
    char    dyn_statement[1024];
    EXEC SQL VAR dyn_statement IS STRING(1024);
EXEC SQL END DECLARE SECTION;

EXEC SQL INCLUDE sqlca;
EXEC SQL INCLUDE sqlda;

SQLDA *bind_dp;
SQLDA *select_dp;

extern SQLDA *sqlald();
extern void sqlnul();

/* Define a buffer to hold longjmp state info. */
jmp_buf jmp_continue;

/* A global flag for the error routine. */
int parse_flag = 0;

main()
{
    int oracle_connect();
    int alloc_descriptors();
    int get_dyn_statement();
    int set_bind_variables();
    int process_select_list();
    int i;

    /* Connect to the database. */
    if (oracle_connect() != 0)
        exit(1);

    /* Allocate memory for the select and bind descriptors. */
    if (alloc_descriptors(MAX_ITEMS,
                          MAX_VNAME_LEN, MAX_INAME_LEN) != 0)
        exit(1);
```

```
/* Process SQL statements. */
for (;;)
{
    i = setjmp(jmp_continue);

    /* Get the statement.  Break on ”exit”. */
    if (get_dyn_statement() != 0)
        break;

    /* Prepare the statement and declare a cursor. */
    EXEC SQL WHENEVER SQLERROR DO sql_error();

    parse_flag = 1;     /* Set a flag for sql_error(). */
    EXEC SQL PREPARE S FROM :dyn_statement;
    parse_flag = 0;     /* Unset the flag. */

    EXEC SQL DECLARE C CURSOR FOR S;

    /* Set the bind variables for any placeholders in the
       SQL statement. */
    set_bind_variables();

    /* Open the cursor and execute the statement.
     * If the statement is not a query (SELECT), the
     * statement processing is completed after the
     * OPEN.
     */

    EXEC SQL OPEN C USING DESCRIPTOR bind_dp;

    /* Call the function that processes the select–list.
     * If the statement is not a query, this function
     * just returns, doing nothing.
     */
    process_select_list();

    /* Tell user how many rows processed. */
    for (i = 0; i < 8; i++)
    {
       if (strncmp(dyn_statement, dml_commands[i], 6) == 0)
       {
           printf(”\n\n%d row%c processed.\n”,
                  sqlca.sqlerrd[2],
                  sqlca.sqlerrd[2] == 1 ? ’\0’ : ’s’);
           break;
       }
    }
}       /* end of for(;;) statement–processing loop */
```

```
    /* When done, free the memory allocated for
       pointers in the bind and select descriptors. */
    for (i = 0; i < MAX_ITEMS; i++)
    {
        if (bind_dp–>V[i] != (char *) 0)
            free(bind_dp–>V[i]);
        free(bind_dp–>I[i]);   /* MAX_ITEMS were allocated. */
        if (select_dp–>V[i] != (char *) 0)
            free(select_dp–>V[i]);
        free(select_dp–>I[i]); /* MAX_ITEMS were allocated. */
    }

    /* Free space used by the descriptors themselves. */
    sqlclu(bind_dp);
    sqlclu(select_dp);

    EXEC SQL WHENEVER SQLERROR CONTINUE;
    /* Close the cursor. */
    EXEC SQL CLOSE C;

    EXEC SQL COMMIT WORK RELEASE;
    puts(”\nHave a good day!\n”);

    EXEC SQL WHENEVER SQLERROR DO sql_error();
    return;
}


oracle_connect()
{
    EXEC SQL BEGIN DECLARE SECTION;
        VARCHAR  username[128];
        VARCHAR  password[32];
    EXEC SQL END DECLARE SECTION;

    printf(”\nusername: ”);
    fgets((char *) username.arr, sizeof username.arr, stdin);
    fflush(stdin);
    username.arr[strlen((char *) username.arr)–1] = ’\0’;
    username.len = strlen((char *) username.arr);

    printf(”password: ”);
    fgets((char *) password.arr, sizeof password.arr, stdin);
    fflush(stdin);
    password.arr[strlen((char *) password.arr) – 1] = ’\0’;
    password.len = strlen((char *) password.arr);


    EXEC SQL WHENEVER SQLERROR GOTO connect_error;
    EXEC SQL CONNECT :username IDENTIFIED BY :password;

    printf(”\nConnected to ORACLE as user %s.\n”, username.arr);

    return 0;

connect_error:
    fprintf(stderr,
            ”Cannot connect to ORACLE as user %s\n”,
            username.arr);
    return –1;
}


/*
 *  Allocate the BIND and SELECT descriptors using sqlald().
 *  Also allocate the pointers to indicator variables
 *  in each descriptor.  The pointers to the actual bind
 *  variables and the select–list items are realloc’ed in
 *  the set_bind_variables() or process_select_list()
 *  routines.  This routine allocates 1 byte for select_dp–>V[i]
 *  and bind_dp–>V[i], so the realloc will work correctly.
 */

alloc_descriptors(size, max_vname_len, max_iname_len)
int size;
int max_vname_len;
int max_iname_len;
{
    int i;

    /*
     * The first sqlald parameter determines the maximum number
     * of array elements in each variable in the descriptor. In
     * other words, it determines the maximum number of bind
     * variables or select–list items in the SQL statement.
     *
     * The second parameter determines the maximum length of
     * strings used to hold the names of select–list items
     * or placeholders.  The maximum length of column
     * names in ORACLE is 30, but you can allocate more or less
     * as needed.
     *
     * The third parameter determines the maximum length of
     * strings used to hold the names of any indicator
     * variables.  To follow ORACLE standards, the maximum
     * length of these should be 30.  But, you can allocate
     * more or less as needed.
     */

    if ((bind_dp =
            sqlald(size,
                   max_vname_len, max_iname_len)) == (SQLDA *) 0)
    {
        fprintf(stderr,
            ”Cannot allocate memory for bind descriptor.”);
        return –1;  /* Have to exit in this case. */
    }

    if ((select_dp =
        sqlald (size,
                max_vname_len, max_iname_len)) == (SQLDA *) 0)
    {
        fprintf(stderr,
            ”Cannot allocate memory for select descriptor.”);
        return –1;
    }
    select_dp–>N = MAX_ITEMS;

    /* Allocate the pointers to the indicator variables, and the
       actual data. */
    for (i = 0; i < MAX_ITEMS; i++) {
        bind_dp–>I[i] = (short *) malloc(sizeof (short));
        select_dp–>I[i] = (short *) malloc(sizeof(short));
        bind_dp–>V[i] = (char *) malloc(1);
        select_dp–>V[i] = (char *) malloc(1);
    }

    return 0;
}


get_dyn_statement()
{
    char *cp, linebuf[256];
    int iter, plsql;
    int help();


    for (plsql = 0, iter = 1; ;)
    {
        if (iter == 1)
        {
            printf(”\nSQL> ”);
            dyn_statement[0] = ’\0’;
        }
        fgets(linebuf, sizeof linebuf, stdin);
        fflush(stdin);

        cp = strrchr(linebuf, ’\n’);
        if (cp && cp != linebuf)
            *cp = ’ ’;
        else if (cp == linebuf)
            continue;

        if ((strncmp(linebuf, ”EXIT”, 4) == 0) ||
            (strncmp(linebuf, ”exit”, 4) == 0))
        {
            return –1;
        }

        else if (linebuf[0] == ’?’ ||
            (strncmp(linebuf, ”HELP”, 4) == 0) ||
            (strncmp(linebuf, ”help”, 4) == 0))
        {
            help();
            iter = 1;
            continue;
        }

        if (strstr(linebuf, ”BEGIN”) ||
            (strstr(linebuf, ”begin”)))
        {
            plsql = 1;
        }

        strcat(dyn_statement, linebuf);

        if ((plsql && (cp = strrchr(dyn_statement, ’/’))) ||
            (!plsql && (cp = strrchr(dyn_statement, ’;’))))
        {
            *cp = ’\0’;
            break;
        }
        else
        {
            iter++;
            printf(”%3d  ”, iter);
        }
    }
    return 0;
}
```

```
set_bind_variables()
{
    int i, n;
    char bind_var[64];

    /* Describe any bind variables (input host variables) */
    EXEC SQL WHENEVER SQLERROR DO sql_error();

    bind_dp–>N = MAX_ITEMS;  /* Init. count of array elements. */
    EXEC SQL DESCRIBE BIND VARIABLES FOR S INTO bind_dp;

    /* If F is negative, there were more bind variables
       than originally allocated by sqlald(). */
    if (bind_dp–>F < 0)
    {
        printf
            (”\nToo many bind variables (%d), maximum is %d.\n”,
                    –bind_dp–>F, MAX_ITEMS);
        return;
    }

    /* Set the maximum number of array elements in the
       descriptor to the number found. */
    bind_dp–>N = bind_dp–>F;

    /* Get the value of each bind variable as a
     * character string.
     *
     * C[i] contains the length of the bind variable
     *      name used in the SQL statement.
     * S[i] contains the actual name of the bind variable
     *      used in the SQL statement.
     *
     * L[i] will contain the length of the data value
     *      entered.
     *
     * V[i] will contain the address of the data value
     *      entered.
     *
     * T[i] is always set to 1 because in this sample program
     *      data values for all bind variables are entered
     *      as character strings.
     *      ORACLE converts to the table value from CHAR.
     *
     * I[i] will point to the indicator value, which is
     *      set to –1 when the bind variable value is ”null”.
     */
    for (i = 0; i < bind_dp–>F; i++)
    {
        printf (”\nEnter value for bind variable %.*s:  ”,
               (int)bind_dp–>C[i], bind_dp–>S[i]);
        fgets(bind_var, sizeof bind_var, stdin);

        /* Get length and remove the new line character. */
        n = strlen(bind_var) – 1;

        /* Set it in the descriptor. */
        bind_dp–>L[i] = n;

        /* (re–)allocate the buffer for the value.
           sqlald() reserves a pointer location for
           V[i] but does not allocate the full space for
           the pointer. */

         bind_dp–>V[i] = (char *) realloc(bind_dp–>V[i],
                         (bind_dp–>L[i] + 1));

        /* And copy it in. */
        strncpy(bind_dp–>V[i], bind_var, n);

        /* Set the indicator variable’s value. */
        if ((strncmp(bind_dp–>V[i], ”NULL”, 4) == 0) ||
                (strncmp(bind_dp–>V[i], ”null”, 4) == 0))
            *bind_dp–>I[i] = –1;
        else
            *bind_dp–>I[i] = 0;

        /* Set the bind datatype to 1 for CHAR. */
        bind_dp–>T[i] = 1;
    }
}



process_select_list()
{
    int i, null_ok, precision, scale;

    if ((strncmp(dyn_statement, ”SELECT”, 6) != 0) &&
        (strncmp(dyn_statement, ”select”, 6) != 0))
    {
        select_dp–>F = 0;
        return;
    }
```

```
/* If the SQL statement is a SELECT, describe the
    select–list items.  The DESCRIBE function returns
    their names, datatypes, lengths (including precision
    and scale), and NULL/NOT NULL statuses. */

select_dp–>N = MAX_ITEMS;

EXEC SQL DESCRIBE SELECT LIST FOR S INTO select_dp;

/* If F is negative, there were more select–list
   items than originally allocated by sqlald(). */
if (select_dp–>F < 0)
{
    printf
       (”\nToo many select–list items (%d), maximum is %d\n”,
            –(select_dp–>F), MAX_ITEMS);
    return;
}

/* Set the maximum number of array elements in the
   descriptor to the number found. */
select_dp–>N = select_dp–>F;

/* Allocate storage for each select–list item.

   sqlprc() is used to extract precision and scale
   from the length (select_dp–>L[i]).

   sqlnul() is used to reset the high–order bit of
   the datatype and to check whether the column
   is NOT NULL.

   CHAR    datatypes have length, but zero precision and
           scale.  The length is defined at CREATE time.

   NUMBER  datatypes have precision and scale only if
           defined at CREATE time.  If the column
           definition was just NUMBER, the precision
           and scale are zero, and you must allocate
           the required maximum length.

   DATE    datatypes return a length of 7 if the default
           format is used.  This should be increased to
           9 to store the actual date character string.
           If you use the TO_CHAR function, the maximum
           length could be 75, but will probably be less
           (you can see the effects of this in SQL*Plus).

   ROWID   datatype always returns a fixed length of 18 if
```

```
             coerced to CHAR.

     LONG and
     LONG RAW datatypes return a length of 0 (zero),
             so you need to set a maximum.  In this example,
             it is 240 characters.

     */

  printf (”\n”);
  for (i = 0; i < select_dp–>F; i++)
  {
      /* Turn off high–order bit of datatype (in this example,
         it does not matter if the column is NOT NULL). */
      sqlnul (&(select_dp–>T[i]),
              &(select_dp–>T[i]), &null_ok);

      switch (select_dp–>T[i])
      {
          case  1 : /* CHAR datatype: no change in length
                       needed, except possibly for TO_CHAR
                       conversions (not handled here). */
              break;
          case  2 : /* NUMBER datatype: use sqlprc() to
                       extract precision and scale. */
              sqlprc (&(select_dp–>L[i]), &precision, &scale);
                    /* Allow for maximum size of NUMBER. */
              if (precision == 0) precision = 40;
                    /* Also allow for decimal point and
                       possible sign. */
              /* convert NUMBER datatype to FLOAT if scale > 0,
                 INT otherwise. */
              if (scale > 0)
                  select_dp–>L[i] = sizeof(float);
              else
                  select_dp–>L[i] = sizeof(int);
              break;

          case  8 : /* LONG datatype */
              select_dp–>L[i] = 240;
              break;

          case 11 : /* ROWID datatype */
              select_dp–>L[i] = 18;
              break;

          case 12 : /* DATE datatype */
              select_dp–>L[i] = 9;
              break;
```

```
        case 23 : /* RAW datatype */
            break;

        case 24 : /* LONG RAW datatype */
            select_dp–>L[i] = 240;
            break;
    }
    /* Allocate space for the select–list data values.
       sqlald() reserves a pointer location for
       V[i] but does not allocate the full space for
       the pointer.  */

     if (select_dp–>T[i] != 2)
       select_dp–>V[i] = (char *) realloc(select_dp–>V[i],
                                select_dp–>L[i] + 1);
     else
       select_dp–>V[i] = (char *) realloc(select_dp–>V[i],
                                select_dp–>L[i]);

    /* Print column headings, right–justifying number
        column headings. */
    if (select_dp–>T[i] == 2)
       if (scale > 0)
         printf (”%.*s ”,select_dp–>L[i]+3, select_dp–>S[i]);
       else
         printf (”%.*s ”, select_dp–>L[i], select_dp–>S[i]);
    else
        printf (”%–.*s ”, select_dp–>L[i], select_dp–>S[i]);

    /* Coerce ALL datatypes except for LONG RAW and NUMBER to
       character. */
    if (select_dp–>T[i] != 24 && select_dp–>T[i] != 2)
        select_dp–>T[i] = 1;

    /* Coerce the datatypes of NUMBERs to float
       or int depending on the scale. */
    if (select_dp–>T[i] == 2)
      if (scale > 0)
         select_dp–>T[i] = 4;  /* float */
      else
         select_dp–>T[i] = 3;  /* int */
}
printf (”\n\n”);

/* FETCH each row selected and print the column values. */
EXEC SQL WHENEVER NOT FOUND GOTO end_select_loop;
```

```
    for (;;)
    {
        EXEC SQL FETCH C USING DESCRIPTOR select_dp;

        /* Since each variable returned has been coerced to a
           character string, int, or float very little processing
           is required here.  This routine just prints out the
           values on the terminal. */
        for (i = 0; i < select_dp->F; i++)
        {
            if (*select_dp->I[i] < 0)
                if (select_dp->T[i] == 4)
                  printf (”%-*c ”,(int)select_dp->L[i]+3, ’ ’);
                else
                  printf (”%-*c ”,(int)select_dp->L[i], ’ ’);
            else
                if (select_dp->T[i] == 3)     /* int datatype */
                  printf (”%*d ”, (int)select_dp->L[i],
                                 *(int *)select_dp->V[i]);
                else if (select_dp->T[i] == 4)/* float datatype*/
                  printf (”%*.2f ”, (int)select_dp->L[i],
                                 *(float *)select_dp->V[i]);
                else                       /* character string */
                  printf (”%-*s ”,
                          (int)select_dp->L[i], select_dp->V[i]);
        }
        printf (”\n”);
    }
end_select_loop:
    return;
}



help()
{
    puts(”\n\nEnter a SQL statement or a PL/SQL block”);
    puts(”at the SQL> prompt.”);
    puts(”Statements can be continued over several”);
    puts(”lines, except within string literals.”);
    puts(”Terminate a SQL statement with a semicolon.”);
    puts(”Terminate a PL/SQL block”);
    puts(”(which can contain embedded semicolons)”);
    puts(”with a slash (/).”);
    puts(”Typing \”exit\” (no semicolon needed)”);
    puts(”exits the program.”);
    puts(”You typed \”?\” or \”help\””);
    puts(” to get this message.\n\n”);
}
```

```
sql_error()
{
    int i;

    /* ORACLE error handler */
    printf (”\n\n%.70s\n”,sqlca.sqlerrm.sqlerrmc);
    if (parse_flag)
        printf(”Parse error at character offset %d.\n”,
           sqlca.sqlerrd[4]);

    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK WORK;
    longjmp(jmp_continue, 1);
}
```

CHAPTER

13

Writing User Exits

This chapter focuses on writing user exits for your Oracle Tools applications. You learn how C subroutines can do certain jobs more quickly and easily than SQL*Forms and Oracle Forms. The following topics are covered:

- common uses for user exits
- writing a user exit
- passing values between the tool and a user exit
- implementing a user exit
- calling a user exit
- guidelines for user exits
- using the EXEC TOOLS routines for applications that use the Oracle Toolset

This chapter is supplemental. For more information about user exits, refer to the *SQL*Forms Designer's Reference*, the *Oracle Forms Reference Manual, Vol. 2*, and your system–specific Oracle documentation.

What Is a User Exit?

A *user exit* is a C subroutine written by you and called by Oracle Forms to do special–purpose processing. You can embed SQL statements and PL/SQL blocks in your user exit, then precompile it as you would a host program.

When called by an Oracle Forms V3 trigger, the user exit runs, then returns a status code to Oracle Forms. Your exit can display messages on the Oracle Forms status line, get and set field values, do high–speed computations and table lookups, and manipulate Oracle data.

Figure 13 – 1 shows how an Oracle Forms application interacts with a user exit.

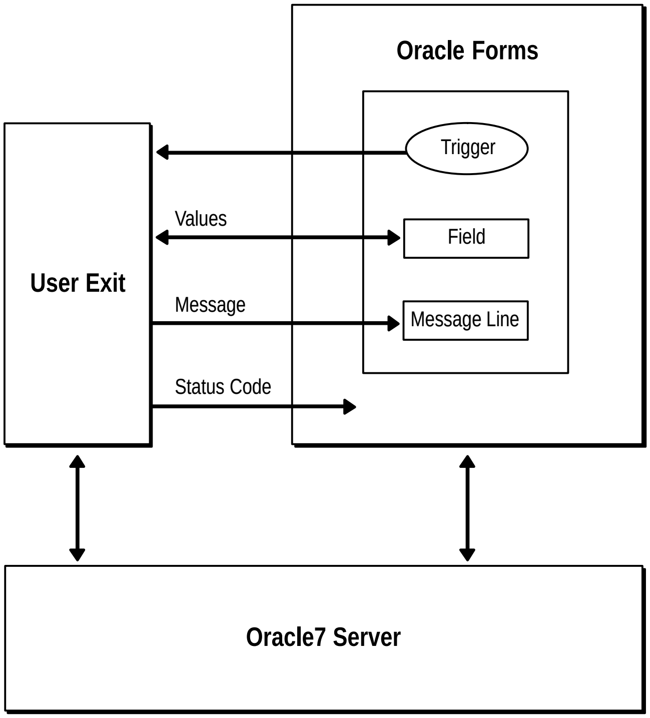


Figure 13 – 1 Oracle Forms and a User Exit

Why Write a User Exit?

SQL*Forms Version 3 allows you to use PL/SQL blocks in triggers. So, in most cases, instead of calling a user exit, you can use the procedural power of PL/SQL. If the need arises, you can call user exits from a PL/SQL block with the USER_EXIT function. User exits are harder to write and implement than SQL, PL/SQL, or SQL*Forms commands. So, you will probably use them only to do processing that is beyond the scope of SQL, PL/SQL, and SQL*Forms. Some common uses follow:

- operations more quickly or easily done in a third generation languages like C (numerical integration, for instance)
- controlling real time devices or processes (issuing a sequence of instructions to a printer or graphics device, for example)
- data manipulations that need extended procedural capabilities (recursive sorting, for example)
- special file I/O operations

Developing a User Exit

This section outlines the way to develop a SQL*Forms 3.0 user exit; later sections go into more detail. For information about the EXEC TOOLS options available with SQL*Forms 4, see the section "EXEC TOOLS Statements" on page 13 – 14. To incorporate a user exit into a form, you take the following steps:

1. Write the user exit in Pro*C.
2. Precompile the source code.
3. Compile the. c file from step 2.
4. Use the GENXTB utility to create a database table, IAPXTB.
5. Use the GENXTB form in SQL*Forms to insert your user exit information into the table.
6. Use the GENXTB utility to read the information from the table and create an IAPXIT source code module. Then compile the source code module.
7. Create a new SQL*Forms executable by linking the standard SQL*Forms modules, your user exit object, and the IAPXIT object created in step 6.

8. In the form, define a trigger to call the user exit.

9. Instruct operators to use the new IAP when running the form. This is unnecessary if the new IAP replaces the standard one. For details, see the Oracle installation or user's guide for your system.

Writing a User Exit

You can use the following kinds of statements to write your SQL*Forms user exit:

- C code
- EXEC SQL
- EXEC ORACLE
- EXEC IAF GET
- EXEC IAF PUT

This section focuses on the EXEC IAF GET and PUT statements, which let you pass values between SQL*Forms and a user exit.

Requirements for Variables

The variables used in EXEC IAF statements must correspond to field names used in the form definition. If a field reference is ambiguous because you did not specify a block name, EXEC IAF defaults to the *context block*—the block that calls the user exit. An invalid or ambiguous reference to a form field generates an error. Host variables must be prefixed with a colon (:) in EXEC IAF statements.

> **Note:** Indicator variables are *not* allowed in EXEC IAF GET and PUT statements.

The IAF GET Statement

This statement allows your user exit to "get" values from fields on a form and assign them to host variables. The user exit can then use the values in calculations, data manipulations, updates, and so on. The syntax of the GET statement follows:

```
EXEC IAF GET field_name1, field_name2, ...
     INTO :host_variable1, :host_variable2, ...;
```

where *field_name* can be any of the following SQL*Forms variables:

- field
- block.field
- system variable
- global variable
- host variable (prefixed with a colon) containing the value of a field, block.field, system variable, or global variable

If a *field_name* is not qualified, the field must be in the context block.

Using IAF GET

The following example shows how a user exit GETs a field value and assigns it to a host variable:

```
EXEC IAF GET employee.job INTO :new_job;
```

All field values are character strings. If it can, GET converts a field value to the datatype of the corresponding host variable. If an illegal or unsupported datatype conversion is attempted, an error is generated.

In the last example, a constant is used to specify *block.field*. You can also use a host string to specify block and field names, as follows:

```
char blkfld[20] = ”employee.job”;
EXEC IAF GET :blkfld INTO :new_job;
```

Unless the field is in the context block, the host string must contain the full *block.field* reference with intervening period. For example, the following usage is *invalid*:

```
char blk[20] = ”employee”;
strcpy(fld, ”job”);
EXEC IAF GET :blk.:fld INTO :new_job;
```

You can mix explicit and stored field names in a GET statement field list, but not in a single field reference. For example, the following usage is *invalid*:

```
strcpy(fld, ”job”);
EXEC IAF GET employee.:fld INTO :new_job;
```

The IAF PUT Statement

This statement allows your user exit to “put” the values of constants and host variables into fields on a form. Thus, the user exit can display on the SQL*Forms screen any value or message you like. The syntax of the PUT statement follows:

```
EXEC IAF PUT field_name1, field_name2, ...
    VALUES (:host_variable1, :host_variable2, ...);
```

where *field_name* can be any of the following SQL*Forms variables:

- field
- block.field
- system variable
- global variable
- host variable (prefixed with a colon) containing the value of a field, block.field, system variable, or global variable

Using IAF PUT

The following example shows how a user exit PUTs the values of a numeric constant, string constant, and host variable into fields on a form:

```
EXEC IAF PUT employee.number, employee.name, employee.job
    VALUES (7934, ’MILLER’, :new_job);
```

Like GET, PUT lets you use a host string to specify block and field names, as follows:

```
char blkfld[20] = ”employee.job”;
EXEC IAF PUT :blkfld VALUES (:new_job);
```

On character–mode terminals, a value PUT into a field is displayed when the user exit returns, rather than when the assignment is made, provided the field is on the current display page. On block–mode terminals, the value is displayed the next time a field is read from the device.

If a user exit changes the value of a field several times, only the last change takes effect.

Calling a User Exit

You call a user exit from a SQL*Forms trigger using a packaged procedure named USER_EXIT (supplied with SQL*Forms). The syntax you use is

```
USER_EXIT(user_exit_string [, error_string]);
```

where *user_exit_string* contains the name of the user exit plus optional parameters and *error_string* contains an error message issued by SQL*Forms if the user exit fails. For example, the following trigger command calls a user exit named LOOKUP:

```
USER_EXIT(’LOOKUP’);
```

Notice that the user exit string is enclosed by single (not double) quotes.

Passing Parameters to a User Exit

When you call a user exit, SQL*Forms passes it the following parameters automatically:

Command Line	Is the user exit string.
Command Line Length	Is the length (in characters) of the user exit string.
Error Message	Is the error string (failure message) if one is defined.
Error Message Length	Is the length of the error string.
In–Query	Is a Boolean value indicating whether the exit was called in normal or query mode.

However, the user exit string allows you to pass additional parameters to the user exit. For example, the following trigger command passes two parameters and an error message to the user exit LOOKUP:

Notice that the user exit string is enclosed by single (not double) quotes.

```
USER_EXIT(’LOOKUP 2025 A’, ’Lookup failed’);
```

You can use this feature to pass field names to the user exit, as the following example shows:

```
USER_EXIT(’CONCAT firstname, lastname, address’);
```

However, it is up to the user exit, not SQL*Forms, to parse the user exit string.

Returning Values to a Form

When a user exit returns control to SQL*Forms, it must also return a code indicating whether it succeeded, failed, or suffered a fatal error. The return code is an integer constant defined by SQL*Forms (see the next section). The three results have the following meanings:

success

The user exit encountered no errors. SQL*Forms proceeds to the *success* label or the next step, unless the Reverse Return Code switch is set by the calling trigger step.

failure

The user exit detected an error, such as an invalid value in a field. An optional message passed by the exit appears on the message line at the bottom of the SQL*Forms screen and on the Display Error screen. SQL*Forms responds as it does to a SQL statement that affects no rows.

fatal error

The user exit detected a condition that makes further processing impossible, such as an execution error in a SQL statement. An optional error message passed by the exit appears on the SQL*Forms Display Error screen. SQL*Forms responds as it does to a fatal SQL error.

If a user exit changes the value of a field, then returns a *failure* or *fatal error* code, SQL*Forms does *not* discard the change. Nor does SQL*Forms discard changes when the Reverse Return Code switch is set and a *success* code is returned.

The IAP Constants

SQL*Forms defines three symbolic constants for use as return codes. Depending on the host language, they are prefixed with IAP or SQL. For example, they might be IAPSUCC, IAPFAIL, and IAPFTL.

Using the SQLIEM Function

By calling the function SQLIEM, your user exit can specify an error message that SQL*Forms will display on the message line if the trigger step fails or on the Display Error screen if the step causes a fatal error. The specified message replaces any message defined for the step. The syntax of the SQLIEM function call is

```
SQLIEM (char *error_message, int message_length);
```

where *error_message* and *message_length* are character and integer variables, respectively. The Pro*C Precompiler generates the appropriate external function declaration for you. ***You pass both parameters by reference; that is, you pass their addresses, not their values***. SQLIEM is a SQL*Forms function; it cannot be called from other Oracle tools such as SQL*ReportWriter.

Using WHENEVER

You can use the WHENEVER statement in an exit to detect invalid datatype conversions (SQLERROR), truncated values PUT into form fields (SQLWARNING), and queries that return no rows (NOT FOUND).

An Example

The following example shows how a user exit that uses the EXEC IAF GET and PUT routines, as well as the *sqliem* function, is coded. Notice that, like a host program, the user exit has a Declare Section and a SQLCA.

```
int
myexit()
{
    char field1[20], field2[20], value1[20], value2[20];
    char result_value[20];
    char errmsg[80];
    int errlen;

    #include sqlca.h
    EXEC SQL WHENEVER SQLERROR GOTO sql_error;
    /* get field values into form */
    EXEC IAF GET :field1, :field2 INTO :value1, :value2;
    /* manipulate the values to obtain result_val */
    ...
    /* put result_val into form field result */
    EXEC IAF PUT result VALUES (:result_val);
    return IAPSUCC;   /* trigger step succeeded */

sql_error:
    strcpy(errmsg, CONCAT(”MYEXIT”, sqlca.sqlerrm.sqlerrmc);
    errlen = strlen(errmsg);
    sqliem(errmsg, &errlen); /* send error msg to Forms */
    return IAPFAIL;
```

Precompiling and Compiling a User Exit

User exits are precompiled like stand–alone host programs. Refer to Chapter 7, “Running the Pro*C Precompiler.” For instructions on compiling a user exit, see the Oracle installation or user’s guide for your system.

Sample Program: A User Exit

The example below shows a user exit.

```
/**************************************************************
Sample Program 5:  SQL*Forms User Exit

This user exit concatenates form fields.  To call the user
exit from a SQL*Forms trigger, use the syntax

   user_exit(’CONCAT field1, field2, ..., result_field’);

where user_exit is a packaged procedure supplied with SQL*Forms
and CONCAT is the name of the user exit.  A sample form named
CONCAT invokes the user exit.
**************************************************************/

#define min(a, b) ((a < b) ? a : b)
#include <stdio.h>
#include <string.h>

/* Include the SQL Communications Area, a structure through which
   Oracle makes runtime status information such as error
   codes, warning flags, and diagnostic text available to the
   program. */

EXEC SQL INCLUDE sqlca;

/* All host variables used in embedded SQL must appear in the
   Declare Section. */

VARCHAR   field[81];
VARCHAR   value[81];
VARCHAR   result[241];

int concat(cmd, cmdlen, msg, msglen, query)

char *cmd;     /* command line in trigger step (”CONCAT...”) */
int  *cmdlen;  /* length of command line */
```

```
char *msg;     /* trigger step failure message from form */
int  *msglen;  /* length of failure message */
int  *query;   /* TRUE if invoked by post–query trigger,
                  FALSE otherwise */
{
    char *cp = cmd + 7;    /* pointer to field list in
                              cmd string; 7 characters
                              are needed for ”CONCAT ” */
    char *fp = (char*)&field.arr[0]; /* pointer to a field name in
                                         cmd string */
    char  errmsg[81];      /* message returned to SQL*Forms
                              on error */
    int   errlen;          /* length of message returned
                              to SQL*Forms */

    /* Branch to label sqlerror if an Oracle error occurs. */
    EXEC SQL WHENEVER SQLERROR GOTO sqlerror;

    result.arr[0] = ’\0’;
    /* Parse field names from cmd string. */
    for (; *cp != ’\0’; cp++)
    {
       if (*cp != ’,’ && *cp != ’ ’)
           /* Copy a field name into field.arr from cmd. */
       {
           *fp = *cp;
           fp++;
       }
       else
           if (*cp == ’ ’)
           {   /* Have whole field name now. */
               *fp = ’\0’;
               field.len = strlen((char *) field.arr);
               /* Get field value from form. */
               EXEC IAF GET :field INTO :value;
               value.arr[value.len] = ’\0’;
               strcat((char *) result.arr, (char *) value.arr);
              /* Reset field pointer. *
               fp = (char *)&field.arr[0];/
           }
    }

    /* Have last field name now. */
    *fp = ’\0’;
    field.len = strlen((char *) field.arr);
    result.len = strlen((char *) result.arr);
    /* Put result into form. */
    EXEC IAF PUT :field VALUES (:result);
    return(IAPSUCC);  /* Trigger step succeeded. */
sqlerror:
    strcpy(errmsg, ”CONCAT: ”);
    strncat(errmsg, sqlca.sqlerrm.sqlerrmc, min(72,
        sqlca.sqlerrm.sqlerrml));
    errlen = strlen(errmsg);
    /* Pass error message to SQL*Forms status line. */
    sqliem(errmsg, &errlen);
    return(IAPFAIL);  /* Trigger step failed. */
}
```

Using the GENXTB Utility

The IAP program table IAPXTB in module IAPXIT contains an entry for each user exit linked into IAP. IAPXTB tells IAP the name, location, and host language of each user exit. When you add a new user exit to IAP, you must add a corresponding entry to IAPXTB. IAPXTB is derived from a database table, also named IAPXTB. You can modify the database table by running the GENXTB form on the operating system command line, as follows:

```
RUNFORM GENXTB username/password
```

A form is displayed that allows you to enter the following information for each user exit you define:

- exit name (see the section "Guidelines" on page 13 – 13)
- C–language code
- date created
- date last modified
- comments

After modifying the IAPXTB database table, use the GENXTB utility to read the table and create an Assembler or C source program that defines the module IAPXIT and the IAPXTB program table it contains. The source language used depends on your operating system. The syntax you use to run the GENXTB utility is

```
GENXTB username/password outfile
```

where *outfile* is the name you give the Assembler or C source program that GENXTB creates.

Linking a User Exit into SQL*Forms

Before running a form that calls a user exit, you must link the user exit into IAP, the SQL*Forms component that runs a form. The user exit can be linked into your standard version of IAP or into a special version for those forms that call the exit.

To produce a new executable copy of IAP, link your user exit object module, the standard IAP modules, the IAPXIT module, and any modules needed from the Oracle and C link libraries.

The details of linking are system–dependent. Check the Oracle installation or user's guide for your system.

Guidelines

The guidelines in this section will help you avoid some common pitfalls.

Naming the Exit

The name of your user exit cannot be an Oracle reserved word. Also avoid using names that conflict with the names of SQL*Forms commands, function codes, and externally defined names used by SQL*Forms. The name of the user exit entry point in the source code becomes the name of the user exit itself. The exit name must be a valid C function name, and a valid filename for your operating system.

SQL*Forms converts the name of a user exit to upper case before searching for the exit. Therefore, the exit name must be in upper case in your source code.

Connecting to Oracle

User exits communicate with Oracle via the connection made by SQL*Forms. However, a user exit can establish additional connections to any database via SQL*Net. For more information, see the section "Concurrent Connections" on page 3 – 88.

Issuing I/O Calls

File I/O is supported but screen I/O is not.

Using Host Variables

Restrictions on the use of host variables in a stand–alone program also apply to user exits. Host variables must be prefixed with a colon in EXEC SQL and EXEC IAF statements. The use of host arrays is not allowed in EXEC IAF statements.

Updating Tables

Generally, a user exit should not UPDATE database tables associated with a form. For example, suppose an operator updates a record in the SQL*Forms work space, then a user exit UPDATEs the corresponding row in the associated database table. When the transaction is COMMITted, the record in the SQL*Forms work space is applied to the table, overwriting the user exit UPDATE.

Issuing Commands

Avoid issuing a COMMIT or ROLLBACK command from your user exit because Oracle will commit or roll back work begun by the SQL*Forms operator, not just work done by the user exit. Instead, issue the COMMIT or ROLLBACK from the SQL*Forms trigger. This also applies to data definition commands (such as ALTER, CREATE, and GRANT) because they issue an implicit COMMIT before and after executing.

EXEC TOOLS Statements

EXEC TOOLS statements support the basic Oracle Toolset (Oracle Forms V4, Oracle Report V2, and Oracle Graphics V2) by providing a generic way to handle get, set, and exception callbacks from user exits. The following discussion focuses on Oracle Forms but the same concepts apply to Oracle Report and Oracle Graphics.

Writing a Toolset User Exit

Besides EXEC SQL, EXEC ORACLE, and host language statements, you can use the following EXEC TOOLS statements to write an Oracle Forms user exit:

- SET
- GET
- SET CONTEXT
- GET CONTEXT
- MESSAGE

The EXEC TOOLS GET and SET statements replace the EXEC IAF GET and PUT statements used with earlier versions of Oracle Forms. Unlike IAF GET and PUT, however, TOOLS GET and SET accept indicator variables. The EXEC TOOLS MESSAGE statement replaces the message–handling function *sqliem*. Now, let us take a brief look at all the EXEC TOOLS statements. For more information, see the *Oracle Forms Reference Manual, Vol 2*.

EXEC TOOLS SET

The EXEC TOOLS SET statement passes values from a user exit to OracleForms. Specifically, it assigns the values of host variables and constants to Oracle Forms variables and items. Values passed to form items display after the user exit returns control to the form. To code the EXEC TOOLS SET statement, you use the syntax

```
EXEC TOOLS SET form_variable[, ...]
    VALUES ({:host_variable :indicator | constant}[, ...]);
```

where *form_variable* is an Oracle Forms field, block.field, system variable, or global variable, or a host variable (prefixed with a colon) containing the value of one of the foregoing items. In the following example, a user exit passes an employee name to Oracle Forms:

```
char ename[20];
short ename_ind;

...

    strcpy(ename, ”MILLER”);
    ename_ind = 0;
    EXEC TOOLS SET emp.ename VALUES (:ename :ename_ind);
```

In this example, *emp.ename* is an Oracle Forms block.field.

EXEC TOOLS GET

The EXEC TOOLS GET statement passes values from Oracle Forms to a user exit. Specifically, it assigns the values of Oracle Forms variables and items to host variables. As soon as the values are passed, the user exit can use them for any purpose. To code the EXEC TOOLS GET statement, you use the syntax

```
EXEC TOOLS GET form_variable[, ...]
    INTO :host_variable:indicator[, ...];
```

where *form_variable* is an Oracle Forms field, block.field, system variable, or global variable, or a host variable (prefixed with a colon) containing the value of one of the foregoing items. In the following example, Oracle Forms passes an item name from a block to your user exit:

```
...
char     name_buff[20];
VARCHAR  name_fld[20];

strcpy(name_fld.arr, ”EMP.NAME”);
name_fld.len = strlen(name_fld.arr);
EXEC TOOLS GET :name_fld INTO :name_buff;
```

EXEC TOOLS SET CONTEXT

The EXEC TOOLS SET CONTEXT statement saves context information from a user exit for later use in another user exit. A pointer variable points to a block of memory in which the context information is stored. With SET CONTEXT, you need not declare a global variable to hold the information. To code the EXEC TOOLS SET CONTEXT statement, you use the syntax

```
EXEC TOOLS SET CONTEXT :host_pointer_variable
    IDENTIFIED BY context_name;
```

where *context_name* is an undeclared identifier or a character host variable (prefixed with a colon) that names the context area.

```
...
char  *context_ptr;
char  context[20];

strcpy(context, ”context1”)
EXEC TOOLS SET CONTEXT :context IDENTIFIED BY application1;
```

EXEC TOOLS GET CONTEXT

The EXEC TOOLS GET CONTEXT statement retrieves context information (saved earlier by SET CONTEXT) into a user exit. A host–language pointer variable points to a block of memory in which the context information is stored. To code the EXEC TOOLS GET CONTEXT statement, you use the syntax

```
EXEC TOOLS GET CONTEXT context_name
    INTO :host_pointer_variable;
```

where *context_name* is an undeclared identifier or a character host variable (prefixed with a colon) that names the context area. In the following example, your user exit retrieves context information saved earlier:

```
...
char  *context_ptr;

EXEC TOOLS GET CONTEXT application1 INTO :context_ptr;
```

EXEC TOOLS MESSAGE

The EXEC TOOLS MESSAGE statement passes a message from a user exit to Oracle Forms. The message is displayed on the Oracle Forms message line after the user exit returns control to the form. To code the EXEC TOOLS MESSAGE statement, you use the syntax

```
EXEC TOOLS MESSAGE message_text [severity_code];
```

where *message–text* is a quoted string or a character host variable (prefixed with a colon), and the optional *severity_code* is an integer constant or an integer host variable (prefixed with a colon). The MESSAGE statement does *not* accept indicator variables. In the following example, your user exit passes an error message to Oracle Forms:

```
EXEC TOOLS MESSAGE ’Bad field name! Please reenter.’;
```

APPENDIX

A

New Features

This appendix describes the new features offered in the Pro*C/C++ Precompiler, Release 2.2. Each new feature is described briefly, and a pointer to the more complete description in the chapters is provided.

Fetching NULLs without Using Indicator Variables

With releases 2.0 and 2.1 of the Pro*C/C++ Precompiler, source files that FETCH data into host variables without associated indicator variables return an ORA–01405 message at runtime if a NULL is returned to the host variable. With release 2.2, when you specify MODE=ORACLE and DBMS=V7, you can disable the ORA–01405 message by also specifying UNSAFE_NULL=YES.

When developing applications for the Oracle7 Server, the preferred practice is to include indicator variables for any host variable that might have a NULL returned to it. When migrating applications from Oracle Version 6 to Oracle7, however, the UNSAFE_NULL option can significantly ease the process.

For more information, see "UNSAFE_NULL" on page 7 – 36 and "Using Indicator Variables" Chapter 1 of your host–language supplement.

Using DBMS=V6

Applications precompiled with DBMS=V6 maintain full compatibility with Oracle Version 6. When upgrading to Oracle7, if you precompile with DBMS=V6 specified, your applications will be unaffected by the ORA–01405 messages.

Using DBMS=V7 and MODE=ORACLE

Applications precompiled with MODE=ORACLE and DBMS=V7 return the ORA–01405 error at runtime if a NULL is returned to a host variable when there is no associated indicator variable. When upgrading to Oracle7 with these options specified, you will need to migrate your applications in one of two ways:

- modify your source code to include the necessary indicator variables
- specify UNSAFE_NULL=YES on the command line

If you are upgrading to Oracle7 and use DBMS=V7 when precompiling, or if you intend to use new Oracle7 features that are different from Oracle Version 6, in most instances, the change requires minimal modification to your source files. However, if your application may FETCH null values into host variables without associated indicator variables, specify UNSAFE_NULL=YES to disable the ORA–01405 message and avoid adding the relevant indicator variables to your source files.

Related Error Messages

For information about precompile time messages associated with the UNSAFE_NULL option, see *Oracle7 Server Messages*..

Support for Developing Multi–threaded Applications

The Pro*C/C++ Precompiler now supports development of multi–threaded Oracle7 Server applications (on those platforms that support multi–threaded applications) using the following:

- a command–line option to generate thread–safe code
- embedded SQL statements and directives to support multi–threading
- thread–safe SQLLIB and other client–side Oracle libraries

For an introduction to threads and how to use these features, see "Developing Multi–threaded Applications" on page 3 – 99.

THREADS Option

With THREADS=YES specified on the command line, the Pro*C/C++ Precompiler ensures that the generated code is thread–safe, given that you follow the guidelines described in "Programming Considerations" on page 3 – 108. With THREADS=YES specified, Pro*C/C++ verifies that all SQL statements execute within the scope of a user–defined runtime context.

Embedded SQL Statements and Directives

To support multi–threaded development with executable embedded SQL statements, the Pro*C/C++ Precompiler use the EXEC SQL commands and directives listed in Table A – 1 and are further described on page 3 – 103.

EXEC SQL Command	Type	Purpose
`CONTEXT ALLOCATE`	SQL Extension	To allocate memory for a SQLLIB runtime context.
`CONTEXT FREE`	SQL Extension	To free memory for a SQLLIB runtime context.
`CONTEXT USE`	Directive	To specify which SQLLIB runtime context to use.
`ENABLE THREADS`	SQL Extension	To initialize a process that supports multiple threads.

Table A – 1 Precompiler Directives and Embedded SQL Commands

Thread–safe SQLLIB Public Functions

For developers of multi–threaded applications, the Pro*C/C++ Precompiler release 2.2 is accompanied by a thread–safe version of the SQLLIB runtime library. All of the public functions are included as are their thread–safe counterparts. Thread–safe public functions are named similarly to the other public functions but are suffixed with a *t*. For more information about the thread–safe SQLLIB functions, see "Thread–safe SQLLIB Public Functions" on page 3 – 107.

APPENDIX

B

Oracle Reserved Words, Keywords, and Namespaces

This appendix lists words that have a special meaning to Oracle. Each word plays a specific role in the context in which it appears. For example, in an INSERT statement, the reserved word INTO introduces the tables to which rows will be added. But, in a FETCH or SELECT statement, the reserved word INTO introduces the output host variables to which column values will be assigned.

Oracle Reserved Words

The following words are reserved by Oracle. That is, they have a special meaning to Oracle and so cannot be redefined. For this reason, you cannot use them to name database objects such as columns, tables, or indexes.

ACCESS
ADD
ALL
ALTER
AND
ANY
ARRAYLEN
AS
ASC
AUDIT
BETWEEN
BY
CHAR
CHECK
CLUSTER
COLUMN
COMMENT
COMPRESS
CONNECT
CREATE
CURRENT
DATE
DECIMAL
DEFAULT
DELETE
DESC
DISTINCT
DROP
ELSE
EXCLUSIVE
EXISTS
FILE
FLOAT
FOR
FROM
GRANT
GROUP
HAVING
IDENTIFIED
IMMEDIATE
IN
INCREMENT
INDEX
INITIAL
INSERT
INTEGER
INTERSECT
INTO
IS
LEVEL
LIKE
LOCK
LONG
MAXEXTENTS
MINUS
MODE
MODIFY
NOAUDIT
NOCOMPRESS
NOT
NOTFOUND
NOWAIT
NULL
NUMBER
OF
OFFLINE
ON
ONLINE
OPTION
OR
ORDER
PCTFREE
PRIOR
PRIVILEGES
PUBLIC
RAW
RENAME
RESOURCE
REVOKE
ROW
ROWID
ROWLABEL
ROWNUM
ROWS
START
SELECT
SESSION
SET
SHARE
SIZE
SMALLINT
SQLBUF
SUCCESSFUL
SYNONYM
SYSDATE
TABLE
THEN
TO
TRIGGER
UID
UNION
UNIQUE
UPDATE
USER
VALIDATE
VALUES
VARCHAR
VARCHAR2
VIEW
WHENEVER
WHERE
WITH

Oracle Keywords

The following words also have a special meaning to Oracle but are not reserved words and so can be redefined. However, some might eventually become reserved words.

```
ADMIN
AFTER
ALLOCATE
ANALYZE
ARCHIVE
ARCHIVELOG
AUTHORIZATION
AVG
BACKUP
BEGIN
BECOME
BEFORE
BLOCK
BODY
CACHE
CANCEL
CASCADE
CHANGE
CHARACTER
CHECKPOINT
CLOSE
COBOL
COMMIT
COMPILE
CONSTRAINT
CONSTRAINTS
CONTENTS
CONTINUE
CONTROLFILE
COUNT
CURSOR
CYCLE
DATABASE
DATAFILE
DBA
DEC
DECLARE
DISABLE
DISMOUNT
DOUBLE
DUMP
EACH
ENABLE
END
ESCAPE
EVENTS
EXCEPT
EXCEPTIONS
EXEC
EXPLAIN
EXECUTE
EXTENT
EXTERNALLY
FETCH
FLUSH
FREELIST
FREELISTS
FORCE
FOREIGN
FORTRAN
FOUND
FUNCTION
GO
GOTO
GROUPS
INCLUDING
INDICATOR
INITRANS
INSTANCE
INT
KEY
LANGUAGE
LAYER
LINK
LISTS
LOGFILE
MANAGE
MANUAL
MAX
MAXDATAFILES
MAXINSTANCES
MAXLOGFILES
MAXLOGHISTORY
MAXLOGMEMBERS
MAXTRANS
MAXVALUE
MIN
MINEXTENTS
MINVALUE
MODULE
MOUNT
NEXT
NEW
NOARCHIVELOG
NOCACHE
NOCYCLE
NOMAXVALUE
NOMINVALUE
NONE
NOORDER
NORESETLOGS
NORMAL
NOSORT
NUMERIC
OFF
OLD
ONLY
OPEN
OPTIMAL
OWN
PACKAGE
PARALLEL
PCTINCREASE
PCTUSED
PLAN
PLI
PRECISION
PRIMARY
PRIVATE
PROCEDURE
```

Oracle Keywords (continued):

PROFILE
QUOTA
READ
REAL
RECOVER
REFERENCES
REFERENCING
RESETLOGS
RESTRICTED
REUSE
ROLE
ROLES
ROLLBACK
SAVEPOINT
SCHEMA
SCN
SECTION
SEGMENT
SEQUENCE
SHARED
SNAPSHOT
SOME
SORT
SQL
SQLCODE
SQLERROR
SQLSTATE
STATEMENT_ID
STATISTICS
STOP
STORAGE
SUM
SWITCH
SYSTEM
TABLES
TABLESPACE
TEMPORARY
THREAD
TIME
TRACING
TRANSACTION
TRIGGERS
TRUNCATE
UNDER
UNLIMITED
UNTIL
USE
USING
WHEN
WRITE
WORK

PL/SQL Reserved Words

The following PL/SQL keywords may require special treatment when used in embedded SQL statements.

ABORT
ACCEPT
ACCESS
ADD
ALL
ALTER
AND
ANY
ARRAY
ARRAYLEN
AS
ASC
ASSERT
ASSIGN
AT
AUTHORIZATION
AVG
BASE_TABLE
BEGIN
BETWEEN
BINARY_INTEGER
BODY
BOOLEAN
BY
CASE
CHAR
CHAR_BASE
CHECK
CLOSE
CLUSTER
CLUSTERS
COLAUTH
COLUMNS
COMMIT
COMPRESS
CONNECT
CONSTANT
COUNT
CRASH
CREATE
CURRENT
CURRVAL
CURSOR
DATABASE
DATA_BASE
DATE
DBA
DEBUGOFF
DEBUGON
DECLARE
DECIMAL
DEFAULT
DEFINITION
DELAY
DELETE
DELTA
DESC
DIGITS
DISPOSE
DISTINCT
DO
DROP
ELSE
ELSIF
END
ENTRY
EXCEPTION
EXCEPTION_INIT
EXISTS
EXIT
FALSE
FETCH
FLOAT
FOR
FORM
FROM

PL/SQL Reserved Words (continued):

FUNCTION
GENERIC
GOTO
GRANT
GROUP
HAVING
IDENTIFIED
IF
IN
INDEX
INDEXES
INDICATOR
INSERT
INTEGER
INTERSECT
INTO
IS
LEVEL
LIKE
LIMITED
LOOP
MAX
MIN
MINUS
MLSLABEL
MOD
MODE
NATURAL
NEW
NEXTVAL
NOCOMPRESS
NOT
NULL
NUMBER
NUMBER_BASE
OF
ON
OPEN
OPTION
OR
ORDER
OTHERS
OUT
PACKAGE
PARTITION
PCTFREE
POSITIVE
PRAGMA
PRIOR
PRIVATE
PROCEDURE
PUBLIC
RAISE
RANGE
REAL
RECORD
RELEASE
REMR
RENAME
RESOURCE
RETURN
REVERSE
REVOKE
ROLLBACK
ROWID
ROWLABEL
ROWNUM
ROWTYPE
RUN
SAVEPOINT
SCHEMA
SELECT
SEPARATE
SET
SIZE
SMALLINT
SPACE
SQL
SQLCODE
SQLERRM
START
STATEMENT
STDDEV
SUBTYPE
SUM
TABAUTH
TABLE
TABLES
TASK
TERMINATE
THEN
TO
TRUE
TYPE
UNION
UNIQUE
UPDATE
USE
VALUES
VARCHAR
VARCHAR2
VARIANCE
VIEW
VIEWS
WHEN
WHERE
WHILE
WITH
WORK
XOR

Oracle Reserved Namespaces

Table B – 1 contains a list of namespaces that are reserved by Oracle. The initial characters of function names in Oracle libraries are restricted to the character strings in this list. Because of potential name conflicts, use function names that do not begin with these characters.

For example, the SQL*Net Transparent Network Service functions all begin with the characters “NS,” so you need to avoid naming functions that begin with ”NS.”

Namespace	Library
O	OCI functions
S	function names from SQLLIB and system–dependent libraries
XA	external functions for XA applications only
GEN KP L NA NC ND NL NM NR NS NT NZ TTC UPI	Internal functions

Table B – 1 Oracle Reserved Namespaces

APPENDIX

C

Performance Tuning

This appendix shows you some simple, easy–to–apply methods for improving the performance of your applications. Using these methods, you can often reduce processing time by 25% or more.

What Causes Poor Performance?

One cause of poor performance is high Oracle communication overhead. Oracle must process SQL statements one at a time. Thus, each statement results in another call to Oracle and higher overhead. In a networked environment, SQL statements must be sent over the network, adding to network traffic. Heavy network traffic can slow down your application significantly.

Another cause of poor performance is inefficient SQL statements. Because SQL is so flexible, you can get the same result with two different statements, but one statement might be less efficient. For example, the following two SELECT statements return the same rows (the name and number of every department having at least one employee):

```
EXEC SQL SELECT dname, deptno
     FROM dept
     WHERE deptno IN (SELECT deptno FROM emp);

EXEC SQL SELECT dname, deptno
     FROM dept
     WHERE EXISTS
     (SELECT deptno FROM emp WHERE dept.deptno = emp.deptno);
```

However, the first statement is slower because it does a time–consuming full scan of the EMP table for every department number in the DEPT table. Even if the DEPTNO column in EMP is indexed, the index is not used because the subquery lacks a WHERE clause naming DEPTNO.

A third cause of poor performance is unnecessary parsing and binding. Recall that before executing a SQL statement, Oracle must parse and bind it. Parsing means examining the SQL statement to make sure it follows syntax rules and refers to valid database objects. Binding means associating host variables in the SQL statement with their addresses so that Oracle can read or write their values.

Many applications manage cursors poorly. This results in unnecessary parsing and binding, which adds noticeably to processing overhead.

How Can Performance Be Improved?

If you are unhappy with the performance of your precompiled programs, there are several ways you can reduce overhead.

You can greatly reduce Oracle communication overhead, especially in networked environments, by

- using host arrays
- using embedded PL/SQL

You can reduce processing overhead—sometimes dramatically—by

- optimizing SQL statements
- using indexes
- taking advantage of row–level locking
- eliminating unnecessary parsing

The following sections look at each of these ways to cut overhead.

Using Host Arrays

Host arrays can increase performance because they let you manipulate an entire collection of data with a single SQL statement. For example, suppose you want to INSERT salaries for 300 employees into the EMP table. Without arrays your program must do 300 individual INSERTs—one for each employee. With arrays, only one INSERT is necessary. Consider the following statement:

```
EXEC SQL INSERT INTO emp (sal) VALUES (:salary);
```

If *salary* is a simple host variable, Oracle executes the INSERT statement once, inserting a single row into the EMP table. In that row, the SAL column has the value of *salary*. To insert 300 rows this way, you must execute the INSERT statement 300 times.

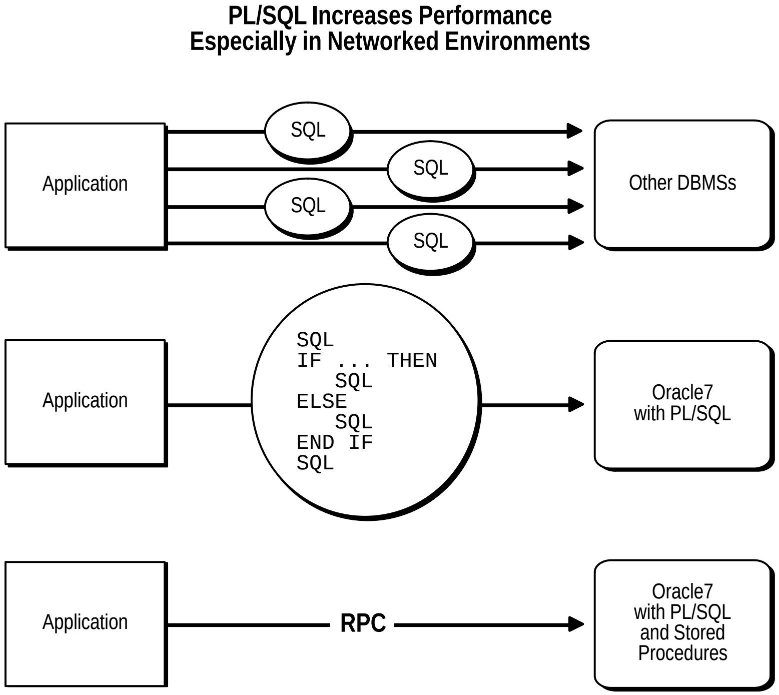
However, if *salary* is a host array of size 300, Oracle inserts all 300 rows into the EMP table at once. In each row, the SAL column has the value of an element in the *salary* array.

For more information, see Chapter 10, "Using Host Arrays."

Using Embedded PL/SQL

As Figure C – 1 shows, if your application is database–intensive, you can use control structures to group SQL statements in a PL / SQL block, then send the entire block to Oracle. This can drastically reduce communication between your application and Oracle.

Also, you can use PL / SQL subprograms to reduce calls from your application to Oracle. For example, to execute ten individual SQL statements, ten calls are required, but to execute a subprogram containing ten SQL statements, only one call is required.



```
SQL
IF ... THEN
   SQL
ELSE
   SQL
END IF
SQL
```

Figure C – 1 PL/SQL Boosts Performance

PL / SQL can also cooperate with Oracle application development tools such as SQL*Forms, SQL*Menu, and SQL*ReportWriter. By adding procedural processing power to these tools, PL / SQL boosts performance. Using PL / SQL, a tool can do any computation quickly and efficiently without calling on Oracle. This saves time and reduces network traffic.

For more information, see Chapter 5 and the *PL/SQL User's Guide and Reference*.

Optimizing SQL Statements

For every SQL statement, the Oracle optimizer generates an *execution plan*, which is a series of steps that Oracle takes to execute the statement. These steps are determined by rules given in the *Oracle7 Server Application Developer's Guide*. Following these rules will help you write optimal SQL statements.

Optimizer Hints

In some cases, you can suggest to Oracle the way to optimize a SQL statement. These suggestions, called *hints*, let you influence decisions made by the optimizer.

Hints are not directives; they merely help the optimizer do its job. Some hints limit the scope of information used to optimize a SQL statement, while others suggest overall strategies.

You can use hints to specify the

- optimization approach for a SQL statement
- access path for each referenced table
- join order for a join
- method used to join tables

Hence, hints fall into the following four categories:

- Optimization Approach
- Access Path
- Join Order
- Join Operation

For example, the two optimization approach hints, COST and NOCOST, invoke the cost–based optimizer and the rule–based optimizer, respectively.

You give hints to the optimizer by placing them in a C–style comment immediately after the verb in a SELECT, UPDATE, or DELETE statement. For instance, the optimizer uses the cost–based approach for the following statement:

```
SELECT /*+ COST */ ename, sal INTO ...
```

For C++ code, optimizer hints in the form //+ are also recognized.

For more information about optimizer hints, see the *Oracle7 Server Application Developer's Guide*.

Trace Facility

You can use the SQL trace facility and the EXPLAIN PLAN statement to identify SQL statements that might be slowing down your application.

The SQL trace facility generates statistics for every SQL statement executed by Oracle. From these statistics, you can determine which statements take the most time to process. Then, you can concentrate your tuning efforts on those statements.

The EXPLAIN PLAN statement shows the execution plan for each SQL statement in your application. An *execution plan* describes the database operations that Oracle must carry out to execute a SQL statement. You can use the execution plan to identify inefficient SQL statements.

For instructions on using these tools and analyzing their output, see the *Oracle7 Server Application Developer's Guide.*

Using Indexes

Using ROWIDs, an *index* associates each distinct value in a table column with the rows containing that value. An index is created with the CREATE INDEX statement. For details, see the *Oracle7 Server SQL Reference.*

You can use indexes to boost the performance of queries that return less than 15% of the rows in a table. A query that returns 15% or more of the rows in a table is executed faster by a *full scan,* that is, by reading all rows sequentially.

Any query that names an indexed column in its WHERE clause can use the index. For guidelines that help you choose which columns to index, see the *Oracle7 Server Application Developer's Guide.*

Taking Advantage of Row–Level Locking

By default, Oracle locks data at the row level rather than the table level. Row–level locking allows multiple users to access different rows in the same table concurrently. The resulting performance gain is significant.

You can specify table–level locking, but it lessens the effectiveness of the transaction processing option. For more information about table locking, see the section "Using LOCK TABLE" on page 8 – 12.

Applications that do online transaction processing benefit most from row–level locking. If your application relies on table–level locking, modify it to take advantage of row–level locking. In general, avoid explicit table–level locking.

Eliminating Unnecessary Parsing

Eliminating unnecessary parsing requires correct handling of cursors and selective use of the following cursor management options:

- MAXOPENCURSORS
- HOLD_CURSOR
- RELEASE_CURSOR

These options affect implicit and explicit cursors, the cursor cache, and private SQL areas.

Handling Explicit Cursors

Recall that there are two types of cursors: implicit and explicit. Oracle implicitly declares a cursor for all data definition and data manipulation statements. However, for queries that return more than one row, you must explicitly declare a cursor (or use host arrays). You use the DECLARE CURSOR statement to declare an explicit cursor. The way you handle the opening and closing of explicit cursors affects performance.

If you need to reevaluate the active set, simply reOPEN the cursor. OPEN will use any new host–variable values. You can save processing time if you do not CLOSE the cursor first.

Note: To make performance tuning easier, Oracle lets you reOPEN an already open cursor. However, this is ANSI extension. So, when MODE=ANSI, you must CLOSE a cursor before reOPENing it.

Only CLOSE a cursor when you want to free the resources (memory and locks) acquired by OPENing the cursor. For example, your program should CLOSE all cursors before exiting.

Cursor Control

In general, there are three ways to control an explicitly declared cursor:

- use DECLARE, OPEN, and CLOSE
- use PREPARE, DECLARE, OPEN, and CLOSE
- COMMIT closes the cursor when MODE=ANSI

With the first way, beware of unnecessary parsing. OPEN does the parsing, but only if the parsed statement is unavailable because the cursor was CLOSEd or never OPENed. Your program should DECLARE the cursor, reOPEN it every time the value of a host variable changes, and CLOSE it only when the SQL statement is no longer needed.

With the second way (for dynamic SQL Methods 3 and 4), PREPARE does the parsing, and the parsed statement is available until a CLOSE is executed. Your program should PREPARE the SQL statement and DECLARE the cursor, reOPEN the cursor every time the value of a host variable changes, rePREPARE the SQL statement and reOPEN the cursor if the SQL statement changes, and CLOSE the cursor only when the SQL statement is no longer needed.

When possible, avoid placing OPEN and CLOSE statements in a loop; this is a potential cause of unnecessary reparsing of the SQL statement. In the next example, both the OPEN and CLOSE statements are inside the outer **while** loop. When MODE=ANSI, the CLOSE statement must be positioned as shown, because ANSI requires a cursor to be CLOSEd before being reOPENed.

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
     SELECT ename, sal from emp where sal >  :salary and
                                     sal <= :salary + 1000;
salary = 0;
while (salary < 5000)
{
     EXEC SQL OPEN emp_cursor;
     while (SQLCODE==0)
     {
         EXEC SQL FETCH emp_cursor INTO ....
         ...
     }
     salary += 1000;
     EXEC SQL CLOSE emp_cursor;
}
```

With MODE=ORACLE, however, a CLOSE statement can execute without the cursor being OPENed. By placing the CLOSE statement outside the outer **while** loop, you can avoid possible reparsing at each iteration of the OPEN statement.

```
...
while (salary < 5000)
{
     EXEC SQL OPEN emp_cursor;
     while (sqlca.sqlcode==0)
     {
         EXEC SQL FETCH emp_cursor INTO ....
         ...
     }
     salary += 1000;
}
EXEC SQL CLOSE emp_cursor;
```

Using the Cursor Management Options

A SQL statement need be parsed only once unless you change its makeup. For example, you change the makeup of a query by adding a column to its select list or WHERE clause. The HOLD_CURSOR, RELEASE_CURSOR, and MAXOPENCURSORS options give you some control over how Oracle manages the parsing and reparsing of SQL statements. Declaring an explicit cursor gives you maximum control over parsing.

SQL Areas and Cursor Cache

When a data manipulation statement is executed, its associated cursor is linked to an entry in the Pro*C cursor cache. The cursor cache is a continuously updated area of memory used for cursor management. The cursor cache entry is in turn linked to a private SQL area.

The private SQL area, a work area created dynamically at run time by Oracle, contains the addresses of host variables, and other information needed to process the statement. An explicit cursor lets you name a SQL statement, access the information in its private SQL area, and, to some extent, control its processing.

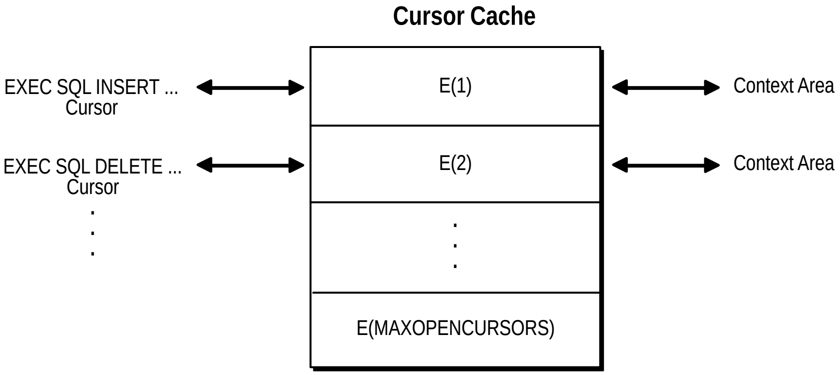


Figure C – 2 Cursors Linked via the Cursor Cache

Figure C – 2 represents the cursor cache after your program has done an INSERT and a DELETE.

Resource Use

The maximum number of open cursors per user session is set by the Oracle initialization parameter OPEN_CURSORS.

MAXOPENCURSORS specifies the *initial* size of the cursor cache. If a new cursor is needed and there are no free cache entries, Oracle tries to reuse an entry. Its success depends on the values of HOLD_CURSOR and RELEASE_CURSOR and, for explicit cursors, on the status of the cursor itself.

If the value of MAXOPENCURSORS is less than the number of cache entries actually needed, Oracle uses the first cache entry marked as reusable. For example, suppose an INSERT statement's cache entry *E*(1) is marked as reusable, and the number of cache entries already equals MAXOPENCURSORS. If the program executes a new statement, cache entry *E*(1) and its private SQL area might be reassigned to the new statement. To re–execute the INSERT statement, Oracle would have to reparse it and reassign another cache entry.

Oracle allocates an additional cache entry if it cannot find one to reuse. For example, if MAXOPENCURSORS=8 and all eight entries are active, a ninth is created. If necessary, Oracle keeps allocating additional cache entries until it runs out of memory or reaches the limit set by OPEN_CURSORS. This dynamic allocation adds to processing overhead.

Thus, specifying a low value for MAXOPENCURSORS saves memory but causes potentially expensive dynamic allocations and deallocations of new cache entries. Specifying a high value for MAXOPENCURSORS assures speedy execution but uses more memory.

Infrequent Execution

Sometimes, the link between an *infrequently* executed SQL statement and its private SQL area should be temporary.

When HOLD_CURSOR=NO (the default), after Oracle executes the SQL statement and the cursor is closed, the precompiler marks the link between the cursor and cursor cache as reusable. The link is reused as soon as the cursor cache entry to which it points is needed for another SQL statement. This frees memory allocated to the private SQL area and releases parse locks. However, because a PREPAREd cursor must remain active, its link is maintained even when HOLD_CURSOR=NO.

When RELEASE_CURSOR=YES, after Oracle executes the SQL statement and the cursor is closed, the private SQL area is automatically freed and the parsed statement lost. This might be necessary if, for example, MAXOPENCURSORS is set low at your site to conserve memory.

If a data manipulation statement precedes a data definition statement and they reference the same tables, specify RELEASE_CURSOR=YES for the data manipulation statement. This avoids a conflict between the parse lock obtained by the data manipulation statement and the exclusive lock required by the data definition statement.

When RELEASE_CURSOR=YES, the link between the private SQL area and the cache entry is immediately removed and the private SQL area freed. Even if you specify HOLD_CURSOR=YES, Oracle must still reallocate memory for a private SQL area and reparse the SQL statement before executing it because RELEASE_CURSOR=YES overrides HOLD_CURSOR=YES.

However, when RELEASE_CURSOR=YES, the reparse might still require no extra processing because Oracle caches the parsed representations of SQL statements and PL/SQL blocks in its *Shared SQL Cache*. Even if its cursor is closed, the parsed representation remains available until it is aged out of the cache.

Frequent Execution

The links between a *frequently* executed SQL statement and its private SQL area should be maintained because the private SQL area contains all the information needed to execute the statement. Maintaining access to this information makes subsequent execution of the statement much faster.

When HOLD_CURSOR=YES, the link between the cursor and cursor cache is maintained after Oracle executes the SQL statement. Thus, the parsed statement and allocated memory remain available. This is useful for SQL statements that you want to keep active because it avoids unnecessary reparsing.

When RELEASE_CURSOR=NO (the default), the link between the cache entry and the private SQL area is maintained after Oracle executes the SQL statement and is not reused unless the number of open cursors exceeds the value of MAXOPENCURSORS. This is useful for SQL statements that are executed often because the parsed statement and allocated memory remain available.

Note: With prior versions of Oracle, when RELEASE_CURSOR=NO and HOLD_CURSOR=YES, after Oracle executes a SQL statement, its parsed representation remains available. But, with Oracle7, when RELEASE_CURSOR=NO and HOLD_CURSOR=YES, the parsed representation remains available only until it is aged out of the Shared SQL Cache. Normally, this is not a problem, but you might get unexpected results if the definition of a referenced object changes before the SQL statement is reparsed.

Parameter Interactions

The following table shows how HOLD_CURSOR and RELEASE_CURSOR interact. Notice that HOLD_CURSOR=NO overrides RELEASE_CURSOR=NO and that RELEASE_CURSOR=YES overrides HOLD_CURSOR=YES.

HOLD_CURSOR	RELEASE_CURSOR	Links are ...
NO	NO	marked as reusable
YES	NO	maintained
NO	YES	removed immediately
YES	YES	removed immediately

Table C – 1 HOLD_CURSOR and RELEASE _CURSOR Interactions

APPENDIX

D

Syntactic and Semantic Checking

By checking the syntax and semantics of embedded SQL statements and PL/SQL blocks, the Pro*C Precompiler helps you quickly find and fix coding mistakes. This appendix shows you how to use the SQLCHECK option to control the type and extent of checking.

What Is Syntactic and Semantic Checking?

Rules of syntax specify how language elements are sequenced to form valid statements. Thus, *syntactic checking* verifies that keywords, object names, operators, delimiters, and so on are placed correctly in your SQL statement. For example, the following embedded SQL statements contain syntax errors:

```
EXEC SQL DELETE FROM EMP WHER DEPTNO = 20;
     –– misspelled keyword WHERE
EXEC SQL INSERT INTO EMP COMM, SAL VALUES (NULL, 1500);
     –– missing parentheses around column names COMM and SAL
```

Rules of semantics specify how valid external references are made. Thus, *semantic checking* verifies that references to database objects and host variables are valid and that host variable datatypes are correct. For example, the following embedded SQL statements contain semantic errors:

```
EXEC SQL DELETE FROM empp WHERE deptno = 20;
     –– nonexistent table, EMPP
EXEC SQL SELECT * FROM emp WHERE ename = :emp_name;
     –– undeclared host variable, emp_name
```

The rules of SQL syntax and semantics are defined in the *Oracle7 Server SQL Reference*.

Controlling the Type and Extent of Checking

You control the type and extent of checking by specifying the SQLCHECK option on the command line. With SQLCHECK, the *type* of checking can be syntactic, semantic, or both. The *extent* of checking can include the following:

- data definition statements (such as CREATE and GRANT)
- data manipulation statements (such as SELECT and INSERT)
- PL/SQL blocks

However, SQLCHECK cannot check dynamic SQL statements because they are not fully defined until run time.

You can specify the following values for SQLCHECK:

- SEMANTICS
- SYNTAX
- NONE

The default value is SYNTAX.

The use of SQLCHECK does not affect the normal syntax checking done on data control, cursor control, and dynamic SQL statements.

Specifying SQLCHECK=SEMANTICS

When SQLCHECK=SEMANTICS, the precompiler checks the syntax and semantics of

- data manipulation statements (INSERT, UPDATE, and so on)
- PL/SQL blocks
- host variable datatypes

as well as the syntax of

- data definition statements (CREATE, ALTER, and so on)

However, only syntactic checking is done on data manipulation statements that use the AT *db_name* clause.

When SQLCHECK=SEMANTICS, the precompiler gets information needed for a semantic check by using embedded DECLARE TABLE statements or if you specify the USERID option on the command line, by connecting to Oracle and accessing the data dictionary. You need not connect to Oracle if every table referenced in a data manipulation statement or PL/SQL block is defined in a DECLARE TABLE statement.

If you connect to Oracle, but some needed information cannot be found in the data dictionary, you must use DECLARE TABLE statements to supply the missing information. A DECLARE TABLE definition overrides a data dictionary definition if they conflict.

If you embed PL/SQL blocks in a host program, you *must* specify SQLCHECK=SEMANTICS.

When checking data manipulation statements, the precompiler uses the Oracle7 set of syntax rules found in the *Oracle7 Server SQL Reference*, but uses a stricter set of semantic rules. In particular, stricter datatype checking is done. As a result, existing applications written for earlier versions of Oracle might not precompile successfully when SQLCHECK=SEMANTICS.

Specify SQLCHECK=SEMANTICS when you precompile new programs or want stricter datatype checking.

Enabling a Semantic Check

When SQLCHECK=SEMANTICS, the precompiler can get information needed for a semantic check in either of the following ways:

- connect to Oracle and access the data dictionary
- use embedded DECLARE TABLE statements

Connecting to Oracle

To do a semantic check, the precompiler can connect to an Oracle database that maintains definitions of tables and views referenced in your host program.

After connecting to Oracle, the precompiler accesses the data dictionary for needed information. The *data dictionary* stores table and column names, table and column constraints, column lengths, column datatypes, and so on.

If some of the needed information cannot be found in the data dictionary (because your program refers to a table not yet created, for example), you must supply the missing information using the DECLARE TABLE statement (discussed later in this appendix).

To connect to Oracle, specify the USERID option on the command line, using the syntax

```
USERID=username/password
```

where *username* and *password* comprise a valid Oracle userid. If you omit the password, you are prompted for it.

If, instead of a username and password, you specify

```
USERID=/
```

the precompiler attempts to automatically connect to Oracle. The attempt succeeds only if an existing Oracle username matches your operating system ID prefixed with “OPS$”, or whatever value the parameter OS_AUTHENT_PREFIX is set to in the INIT.ORA file. For example, if your operating system ID is MBLAKE, an automatic connect only succeeds if OPS$MBLAKE is a valid Oracle username.

If you omit the USERID option, the precompiler must get needed information from embedded DECLARE TABLE statements.

If you try connecting to Oracle but cannot (because the database is unavailable, for example), an error message is issued and your program is not precompiled.

Using DECLARE TABLE

The precompiler can do a semantic check without connecting to Oracle. To do the check, the precompiler must get information about tables and views from embedded DECLARE TABLE statements. Thus, every table referenced in a data manipulation statement or PL/SQL block must be defined in a DECLARE TABLE statement.

The syntax of the DECLARE TABLE statement is

```
EXEC SQL DECLARE table_name TABLE
     (col_name col_datatype [DEFAULT expr] [NULL|NOT NULL], ...);
```

where *expr* is any expression that can be used as a default column value in the CREATE TABLE statement.

If you use DECLARE TABLE to define a database table that already exists, the precompiler uses your definition, ignoring the one in the data dictionary.

Specifying SQLCHECK=SYNTAX

When SQLCHECK=SYNTAX, the precompiler checks the syntax of

- data manipulation statements
- data definition statements
- host variable datatypes

No semantic check is done, and the following restrictions apply:

- No connection to Oracle is attempted and USERID becomes an invalid option. If you specify USERID, a warning message is issued.
- DECLARE TABLE statements are ignored; they serve only as documentation.
- PL/SQL blocks are not allowed. If the precompiler finds a PL/SQL block, an error message is issued.

When checking data manipulation statements, the precompiler uses Oracle7 syntax rules. These rules are downwardly compatible, so specify SQLCHECK=SYNTAX when migrating your precompiled programs.

Specifying SQLCHECK=NONE

When SQLCHECK=NONE, no semantic check is done, a minimal syntactic check is done, and the following restrictions apply:

- No connection to Oracle is attempted and USERID becomes an invalid option. If you specify USERID, a warning message is issued.
- DECLARE TABLE statements are ignored; they serve only as documentation.
- PL/SQL blocks are not allowed. If the precompiler finds a PL/SQL block, an error message is issued.
- The precompiler does not check the syntax of DML statements. This is useful if the application contains non–Oracle SQL, for example if a connection to a non–Oracle server will be made using the Open Gateway.

Specify SQLCHECK=NONE if your program references tables not yet created and is missing DECLARE TABLE statements for them or if stricter datatype checking is undesirable.

Entering the SQLCHECK Option

You can enter the SQLCHECK option inline or on the command line. However, the level of checking you specify inline cannot be higher than the level you specify (or accept by default) on the command line. For example, if you specify SQLCHECK=SYNTAX on the command line, you cannot specify SQLCHECK=SEMANTICS inline.

APPENDIX

E

System–Specific References

This appendix groups together in one place all references in this guide to system–specific information.

System–Specific Information

Each of the sections below describes briefly a reference in this guide to system–specific Pro*C behavior. System–specific information is described in the appropriate Oracle system–specific documentation for your platform.

Location of Standard Header Files

The location of the standard Pro*C header files—*sqlca.h*, *oraca.h*, and *sqlda.h*—is system specific. On UNIX systems, they are located in the *$ORACLE_HOME/sqllib/public* directory. For other operating systems, see your Oracle system–specific documentation.

Specifying Location of Included Files for the C Compiler

When you use the Pro*C command–line option INCLUDE= to specify the location of a non–standard file to be included, you should also specify the same location for the C compiler. The way you do this is system specific. See pages 3 – 8 and 3 – 9.

ANSI C Support

Some C compilers support the ANSI C standard; some do not. Make sure to use the CODE= option to make the C code that Pro*C generates compatible with your system's C compiler. See page 3 – 12.

Struct Component Alignment

C compilers vary in the way they align struct components, usually depending on the system hardware. Use the *sqlvcp()* function to determine the padding added to the *.arr* component of a VARCHAR struct. See the section "Finding the Length of a VARCHAR Array Component" on page 3 – 37. See also page 3 – 59.

External Datatypes

Some Oracle external datatypes are designed to support host datatypes in languages other than C. For example, DECIMAL and DISPLAY are used mainly for PL/I and COBOL. See the section "External Datatypes" on page 3 – 49.

Size of an Integer

The size in bytes of integer datatypes is system dependent. See the section "Integer" on page 3 – 51.

Size of ROWID

The binary external size of a ROWID datatype is system dependent. See the section "ROWID" on page 3 – 52.

Byte Ordering

The order of bytes in a word is platform dependent. See the section "Unsigned" on page 3 – 54.

Connecting to Oracle

Connecting to Oracle using the SQL*Net V1 or V2 drivers involves system–specific network protocols. See the section "Connecting to Oracle" on page 3 – 86 for more details.

Linking in an XA Library

You link in your XA library in a system–dependent way. See the section "Linking" on page 3 – 115, and your Oracle installation or user's guides, for more information.

Location of the Pro*C Executable

The location of the Pro*C Precompiler is system specific. See the section "Precompiler Command" on page 7 – 2, and your installation or user's guides, for more information.

System Configuration File

Each precompiler installation has a system configuration file. This file is not shipped with the precompiler; it must be created by the system administrator. The location (directory path) which Pro*C searches for the system configuration file is system dependent. See the section "Configuration Files" on page 7 – 4 for more information.

INCLUDE Option Syntax

The syntax for the value of the INCLUDE command–line option is system specific. See the "INCLUDE'' section on page 7 – 23.

Compiling and Linking

Compiling and linking your Pro*C output to get an executable application is always system dependent. See the section "Compiling and Linking" on page 7 – 41, and the following sections, for additional information.

User Exits

Compiling and linking Oracle Forms user exits is system specific. See Chapter 13.

APPENDIX

F Embedded SQL Commands and Directives

This appendix contains descriptions of both SQL92 embedded SQL commands and directives and the Oracle embedded SQL extensions. These commands and directives are prefaced in your source code with the keywords, EXEC SQL. Rather than trying to memorize all of the SQL syntax, simply refer to this appendix, which includes the following:

- a summary of embedded SQL commands and directives
- a section about the command descriptions
- how to read syntax diagrams
- an alphabetic listing of the commands and directives

For detailed usage notes, see the *Oracle7 Server SQL Reference.*

Summary of Precompiler Directives and Embedded SQL Commands

Embedded SQL commands place DDL, DML, and Transaction Control statements within a procedural language program. Embedded SQL is supported by the Oracle Precompilers. Table F – 1 provides a functional summary of the embedded SQL commands and directives.

The *type* column in Table F – 1 is displayed in the format, *source / type*, where:

source	is either SQL92 standard SQL (S) or an Oracle extension (O)
type	is either an executable (E) statement or a directive (D)

EXEC SQL Statement	Type	Purpose
ALLOCATE	O / E	To allocate memory for a cursor variable.
CLOSE	S / E	To disable a cursor, releasing the resources it holds.
COMMIT	S / E	To end the current transaction, making all database change permanent (optionally frees resources and disconnects from the database)
CONNECT	O / E	To log on to an Oracle7 instance.
CONTEXT ALLOCATE	O / E	To allocate memory for a SQLLIB runtime context.
CONTEXT FREE	O / E	To free memory for a SQLLIB runtime context.
CONTEXT USE	O / D	To specify which SQLLIB runtime context to use for subsequent executable SQL statements when multiple threads are used.
DECLARE CURSOR	S / D	To declare a cursor, associating it with a query.
DECLARE DATABASE	O / D	To declare an identifier for a non–default database to be accessed in subsequent embedded SQL statements.
DECLARE STATEMENT	S / D	To assign a SQL variable name to a SQL statement.
DECLARE TABLE	O / D	To declare the table structure for semantic checking of embedded SQL statements by the Oracle Precompiler.
DELETE	S / E	To remove rows from a table or from a view's base table.
DESCRIBE	S / E	To initialize a descriptor, a structure holding host variable descriptions.
ENABLE THREADS	O / E	To initialize a process that supports multiple threads.
EXECUTE...END–EXEC	O / E	To execute an anonymous PL / SQL block.
EXECUTE	S / E	To execute a prepared dynamic SQL statement.
EXECUTE IMMEDIATE	S / E	To prepare and execute a SQL statement with no host variables.
FETCH	S / E	To retrieve rows selected by a query.
INSERT	S / E	To add rows to a table or to a view's base table.
OPEN	S / E	To execute the query associated with a cursor.
PREPARE	S / E	To parse a dynamic SQL statement.

Table F – 1 Precompiler Directives and Embedded SQL Commands and Clauses

EXEC SQL Statement	Type	Purpose
ROLLBACK	S/E	To end the current transaction, discard all changes in the current transaction, and release all locks (optionally release resources and disconnect from the database).
SAVEPOINT	S/E	To identify a point in a transaction to which you can later roll back.
SELECT	S/E	To retrieve data from one or more tables, views, or snapshots, assigning the selected values to host variables.
TYPE	O/D	To assign an Oracle7 external datatype to a whole class of host variables by equivalencing the external datatype to a user–defined datatype.
UPDATE	S/E	To change existing values in a table or in a view's base table.
VAR	O/D	To override the default datatype and assign a specific Oracle7 external datatype to a host variable.
WHENEVER	S/D	To specify handling for error and warning conditions.

Table F – 1 Precompiler Directives and Embedded SQL Commands and Clauses (continued)

About The Command Descriptions

The directives, commands, and clauses appear alphabetically. The description of each contains the following sections:

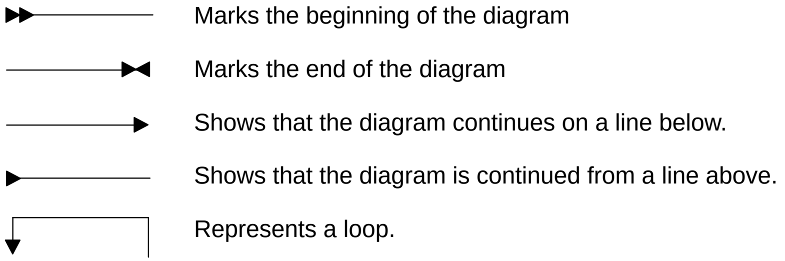
Purpose	describes the basic uses of the command.
Prerequisites	lists privileges you must have and steps that you must take before using the command. Unless otherwise noted, most commands also require that the database be open by your instance.
Syntax	shows the keywords and parameters of the command.
Keywords and Parameters	describes the purpose of each keyword and parameter.
Usage Notes	discusses how and when to use the command.
Examples	shows example statements of the command.
Related Topics	lists related commands, clauses, and sections of this manual.

How to Read Syntax Diagrams

Easy–to–understand *syntax diagrams* are used to illustrate embedded SQL syntax. They are line–and–arrow drawings that depict valid syntax. If you have never used them, do not worry. This section tells you all you need to know.

Once you understand the logical flow of a syntax diagram, it becomes a helpful guide. You can verify or construct any embedded SQL statement by tracing through its syntax diagram.

Syntax diagrams use lines and arrows to show how commands, parameters, and other language elements are sequenced to form statements. Trace each diagram from left to right, in the direction shown by the arrows. The following symbols will guide you:



Marks the beginning of the diagram

Marks the end of the diagram

Shows that the diagram continues on a line below.

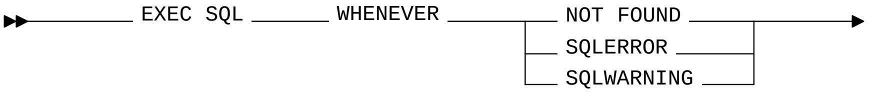
Shows that the diagram is continued from a line above.

Represents a loop.

Commands and other keywords appear in UPPER CASE. Parameters appear in lower case. Operators, delimiters, and terminators appear as usual. Following the conventions defined in the Preface, a semicolon terminates statements.

If the syntax diagram has more than one path, you can choose any path to travel.

If you have the choice of more than one keyword, operator, or parameter, your options appear in a vertical list. In the following example, you can travel down the vertical line as far as you like, then continue along any horizontal line:

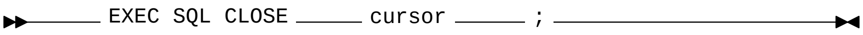


According to the diagram, all of the following statements are valid:

```
EXEC SQL WHENEVER NOT FOUND ...
EXEC SQL WHENEVER SQLERROR ...
EXEC SQL WHENEVER SQLWARNING ...
```

Required Keywords and Parameters

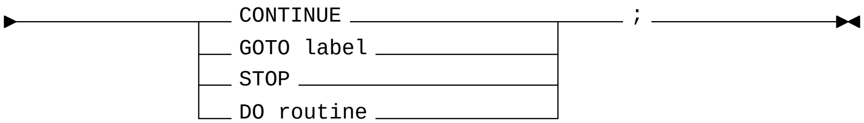
Required keywords and parameters can appear singly or in a vertical list of alternatives. Single required keywords and parameters appear on the *main path*, that is, on the horizontal line you are currently traveling. In the following example, *cursor* is a required parameter:



If there is a cursor named *emp_cursor*, then, according to the diagram, the following statement is valid:

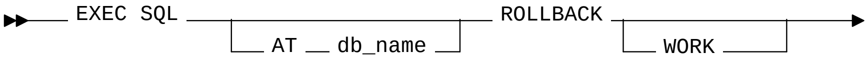
```
EXEC SQL CLOSE emp_cursor;
```

If any of the keywords or parameters in a vertical list appears on the main path, one of them is required. That is, you must choose one of the keywords or parameters, but not necessarily the one that appears on the main path. In the following example, you must choose one of the four actions:



Optional Keywords and Parameters

If keywords and parameters appear in a vertical list below the main path, they are optional. That is, you need not choose one of them. In the following example, instead of traveling down a vertical line, you can continue along the main path:

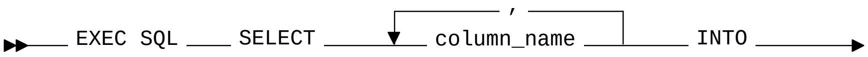


If there is a database named *oracle2*, then, according to the diagram, all of the following statements are valid:

```
EXEC SQL ROLLBACK;
EXEC SQL ROLLBACK WORK;
EXEC SQL AT oracle2 ROLLBACK;
```

Syntax Loops

Loops let you repeat the syntax within them as many times as you like. In the following example, *column_name* is inside a loop. So, after choosing one column name, you can go back repeatedly to choose another.

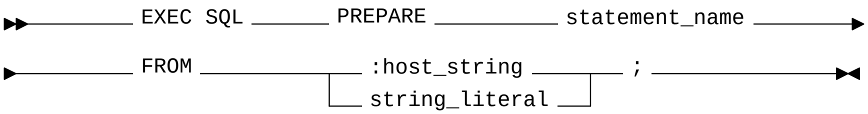


If DEBIT, CREDIT, and BALANCE are column names, then, according to the diagram, all of the following statements are valid:

```
EXEC SQL SELECT DEBIT INTO ...
EXEC SQL SELECT CREDIT, BALANCE INTO ...
EXEC SQL SELECT DEBIT, CREDIT, BALANCE INTO ...
```

Multi–part Diagrams

Read a multi–part diagram as if all the main paths were joined end–to–end. The following example is a two–part diagram:



According to the diagram, the following statement is valid:

```
EXEC SQL PREPARE sql_statement FROM :sql_string;
```

Database Objects

The names of Oracle objects, such as tables and columns, must not exceed 30 characters in length. The first character must be a letter, but the rest can be any combination of letters, numerals, dollar signs ($), pound signs (#), and underscores (_).

However, if an Oracle identifier is enclosed by quotation marks ("), it can contain any combination of legal characters, including spaces but excluding quotation marks.

Oracle identifiers are not case–sensitive except when enclosed by quotation marks.

ALLOCATE (Executable Embedded SQL Extension)

Purpose

To allocate a cursor variable to be referenced in a PL / SQL block.

Prerequisites

A cursor variable (see Chapter 3) of type SQL_CURSOR must be declared before allocating memory for the cursor variable.

Syntax

```
►►── EXEC SQL ALLOCATE :cursor_variable ──────────────────────────►◄
```

Keywords and Parameters

:cursor_variable is the cursor variable to be allocated.

Usage Notes

Whereas a cursor is static, a cursor variable is dynamic because it is not tied to a specific query. You can open a cursor variable for any type–compatible query.

For more information on this command, see *PL/SQL User's Guide and Reference* and *Oracle7 Server SQL Reference.*

Example

This partial example illustrates the use of the ALLOCATE command in a Pro*C / C++ embedded SQL program:

```
EXEC SQL BEGIN DECLARE SECTION;
   SQL_CURSOR emp_cv;
   struct{ ... } emp_rec;
EXEC SQL END DECLARE SECTION;
EXEC SQL ALLOCATE emp_cv;
EXEC SQL EXECUTE
   BEGIN
      OPEN :emp_cv FOR SELECT * FROM emp;
    END;
END-EXEC;
for (;;)
{  EXEC SQL FETCH :emp_cv INTO :emp_rec;
}
```

Related Topics

CLOSE command on F – 8
EXECUTE command on F – 32
FETCH command on F – 36

CLOSE (Executable Embedded SQL)

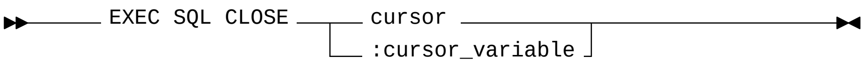
Purpose

To disable a cursor, freeing the resources acquired by opening the cursor, and releasing parse locks.

Prerequisites

The cursor or cursor variable must be open and MODE=ANSI.

Syntax



Keywords and Parameters

cursor is a cursor to be closed.

cursor_variable is a cursor variable to be closed.

Usage Notes

Rows cannot be fetched from a closed cursor. A cursor need not be closed to be reopened. The HOLD_CURSOR and RELEASE_CURSOR precompiler options alter the effect of the CLOSE command. For information on these options, see Chapter 7.

Example

This example illustrates the use of the CLOSE command:

```
EXEC SQL CLOSE emp_cursor;
```

Related Topics

PREPARE command on F – 45
DECLARE CURSOR command on F – 17
OPEN command on F – 43

COMMIT (Executable Embedded SQL)

Purpose

To end your current transaction, making permanent all its changes to the database and optionally freeing all resources and disconnecting from the Oracle7 Server.

Prerequisites

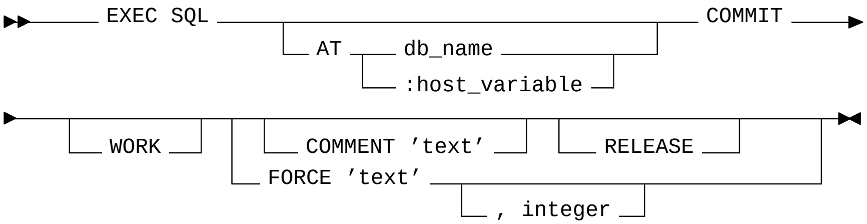
To commit your current transaction, no privileges are necessary.

To manually commit a distributed in–doubt transaction that you originally committed, you must have FORCE TRANSACTION system privilege. To manually commit a distributed in–doubt transaction that was originally committed by another user, you must have FORCE ANY TRANSACTION system privilege.

If you are using Trusted Oracle7 in DBMS MAC mode, you can only commit an in–doubt transaction if your DBMS label matches the label the transaction's label and the creation label of the user who originally committed the transaction or if you satisfy one of the following criteria:

- If the transaction's label or the user's creation label is higher than your DBMS label, you must have READUP and WRITEUP system privileges.
- If the transaction's label or the user's creation label is lower than your DBMS label, you must have WRITEDOWN system privilege.
- If the transaction's label or the user's creation label is not comparable with your DBMS label, you must have READUP, WRITEUP, and WRITEDOWN system privileges.

Syntax



Keyword and Parameters

AT

identifies the database to which the COMMIT statement is issued. The database can be identified by either:

db_name

is a database identifier declared in a previous DECLARE DATABASE statement.

:*host_variable*

is a host variable whose value is a previously declared *db_name*.

If you omit this clause, Oracle7 issues the statement to your default database.

WORK

is supported only for compliance with standard SQL. The statements COMMIT and COMMIT WORK are equivalent.

COMMENT

specifies a comment to be associated with the current transaction. The '*text*' is a quoted literal of up to 50 characters that Oracle7 stores in the data dictionary view DBA_2PC_PENDING along with the transaction ID if the transaction becomes in–doubt.

RELEASE

frees all resources and disconnects the application from the Oracle7 Server.

FORCE

manually commits an in–doubt distributed transaction. The transaction is identified by the '*text*' containing its local or global transaction ID. To find the IDs of such transactions, query the data dictionary view DBA_2PC_PENDING. You can also use the optional *integer* to explicitly assign the transaction a system change number (SCN). If you omit the *integer*, the transaction is committed using the current SCN.

Usage Notes

Always explicitly commit or rollback the last transaction in your program by using the COMMIT or ROLLBACK command and the RELEASE option. Oracle7 automatically rolls back changes if the program terminates abnormally.

The COMMIT command has no effect on host variables or on the flow of control in the program. For more information on this command, see Chapter 8.

Example

This example illustrates the use of the embedded SQL COMMIT command:

```
EXEC SQL AT sales_db COMMIT RELEASE;
```

Related Topics

ROLLBACK command on F – 46
SAVEPOINT command on F – 49

CONNECT (Executable Embedded SQL Extension)

Purpose

To log on to an Oracle7 database.

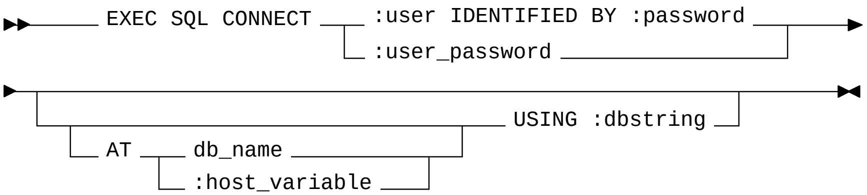
Prerequisites

You must have CREATE SESSION system privilege in the specified database.

If you are using Trusted Oracle7 in DBMS MAC mode, your operating system label must dominate both your creation label and the label at which you were granted CREATE SESSION system privilege. Your operating system label must also fall between the operating system equivalents of DBHIGH and DBLOW, inclusive.

If you are using Trusted Oracle7 in OS MAC mode, your operating system label must match the label of the database to which you are connecting.

Syntax



Keyword and Parameters

:*user*
:*password*
specifies your username and password separately.

:*user_password*
is a single host variable containing the Oracle7 username and password separated by a slash (/).

To allow Oracle7 to verify your connection through your operating system, specify "/" as the :*user_password* value.

AT
identifies the database to which the connection is made. The database can be identified by either:

db_name
is a database identifier declared in a previous DECLARE DATABASE statement.

:*host_variable*
is a host variable whose value is a previously declared *db_name*.

USING specifies the SQL*Net database specification string used to connect to a non–default database. If you omit this clause, you are connected to your default database.

Usage Notes

A program can have multiple connections, but can only connect once to your default database. For more information on this command, see Chapter 3.

Example

The following example illustrate the use of CONNECT:

```
EXEC SQL CONNECT :username
    IDENTIFIED BY :password
```

You can also use this statement in which the value of :*userid* is the value of :*username* and :*password* separated by a "/" such as 'SCOTT/TIGER':

```
EXEC SQL CONNECT :userid
```

Related Topics

COMMIT command on F – 9
DECLARE DATABASE command on F – 19
ROLLBACK command on F – 46

CONTEXT ALLOCATE (Executable Embedded SQL Extension)

Purpose

To initialize a SQLLIB runtime context that is referenced in an EXEC SQL CONTEXT USE statement.

Prerequisites

The runtime context must be declared of type **sql_context**.

Syntax

```
►►── EXEC SQL CONTEXT ALLOCATE :context ─────────────────────────►◄
```

Keywords and Parameters

:*context* is the SQLLIB runtime context for which memory is to be allocated.

Usage Notes

In a multi–threaded application, execute this function once for each runtime context.

For more information on this command, see "Developing Multi–threaded Applications" on page 3 – 99.

Example

This example illustrates the use of a CONTEXT ALLOCATE command in a Pro*C/C++ embedded SQL program:

```
EXEC SQL CONTEXT ALLOCATE :ctx1;
```

Related Topics

CONTEXT FREE on F – 15
CONTEXT USE on F – 16
ENABLE THREADS on F – 29

CONTEXT FREE (Executable Embedded SQL Extension)

Purpose

To free all memory associated with a runtime context and place a null pointer in the host program variable.

Prerequisites

The CONTEXT ALLOCATE command must be used to allocate memory for the specified runtime context before the CONTEXT FREE command can free the memory allocated for it.

Syntax

```
►►── EXEC SQL CONTEXT FREE :context ──────────────────────────────►◄
```

Keywords and Parameters

context is the allocated runtime context for which the memory is to be deallocated.

Usage Notes

For more information on this command, see “Developing Multi–threaded Applications” on page 3 – 99.

Example

This example illustrates the use of a CONTEXT FREE command in a Pro*C / C++ embedded SQL program:

```
EXEC SQL CONTEXT FREE :ctx1;
```

Related Topics

CONTEXT ALLOCATE on F – 14
CONTEXT USE on F – 16
ENABLE THREADS on F – 29

CONTEXT USE (Oracle Embedded SQL Directive)

Purpose

To instruct the precompiler to use the specified SQLLIB runtime context on subsequent executable SQL statements.

Prerequisites

The runtime context specified by the CONTEXT USE directive must be previously allocated using the CONTEXT ALLOCATE command.

Syntax

```
►►── EXEC SQL CONTEXT USE :context ──────────────────────►◄
```

Keywords and Parameters

:*context* is the allocated runtime context to use for subsequent executable SQL statements that follow it. For example, after specifying in your source code which context to use (multiple contexts can be allocated), you can connect to the Oracle Server and perform database operations within the scope of that context.

Usage Notes

This statement has no effect on declarative statements such as EXEC SQL INCLUDE or EXEC ORACLE OPTION. It works similarly to the EXEC SQL WHENEVER directive in that it affects all executable SQL statements which positionally follow it in a given source file without regard to standard C scope rules.

For more information on this command, see "Developing Multi–threaded Applications" on page 3 – 99.

Example

This example illustrates the use of a CONTEXT USE directive in a Pro*C/C++ embedded SQL program:

```
EXEC SQL CONTEXT USE :ctx1;
```

Related Topics

CONTEXT ALLOCATE on F – 14
CONTEXT FREE on F – 15
ENABLE THREADS on F – 29

DECLARE CURSOR (Embedded SQL Directive)

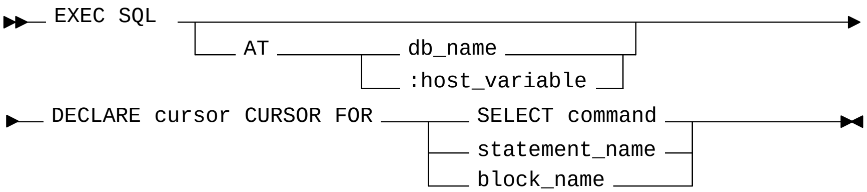
Purpose

To declare a cursor, giving it a name and associating it with a SQL statement or a PL/SQL block.

Prerequisites

If you associate the cursor with an identifier for a SQL statement or PL/SQL block, you must have declared this identifier in a previous DECLARE STATEMENT statement.

Syntax



Keywords and Parameters

AT

identifies the database on which the cursor is declared. The database can be identified by either:

db_name is a database identifier declared in a previous DECLARE DATABASE statement.

:*host_variable* is a host variable whose value is a previously declared *db_name.*

If you omit this clause, Oracle7 declares the cursor on your default database.

cursor

is the name of the cursor to be declared.

SELECT *command*

is a SELECT statement to be associated with the cursor. The following statement cannot contain an INTO clause.

statement_name
block_name

identifies a SQL statement or PL/SQL block to be associated with the cursor. The *statement_name* or *block_name m*ust be previously declared in a DECLARE STATEMENT statement.

Usage Notes

You must declare a cursor before referencing it in other embedded SQL statements. The scope of a cursor declaration is global within its precompilation unit and the name of each cursor must be unique in its scope. You cannot declare two cursors with the same name in a single precompilation unit.

You can reference the cursor in the WHERE clause of an UPDATE or DELETE statement using the CURRENT OF syntax, provided that the cursor has been opened with an OPEN statement and positioned on a row with a FETCH statement. For more information on this command, see Chapter 3.

Example

This example illustrates the use of a DECLARE CURSOR statement:

```
EXEC SQL DECLARE emp_cursor CURSOR
    FOR SELECT ename, empno, job, sal
            FROM emp
            WHERE deptno = :deptno
            FOR UPDATE OF sal
```

Related Topics

CLOSE command on F – 8
DECLARE DATABASE command on F – 19
DECLARE STATEMENT command on F – 20
DELETE command on F – 23
FETCH command on F – 36
OPEN command on F – 43
PREPARE command on F – 45
SELECT command on F – 50
UPDATE command on F – 56

DECLARE DATABASE (Oracle Embedded SQL Directive)

Purpose

To declare an identifier for a non–default database to be accessed in subsequent embedded SQL statements.

Prerequisites

You must have access to a username on the non–default database.

Syntax

```
►►— EXEC SQL DECLARE db_name DATABASE ——————————————————————————►◄
```

Keywords and Parameters

db_name is the identifier established for the non–default database.

Usage Notes

You declare a *db_name* for a non–default database so that other embedded SQL statements can refer to that database using the AT clause. Before issuing a CONNECT statement with an AT clause, you must declare a *db_name* for the non–default database with a DECLARE DATABASE statement.

For more information on this command, see Chapter 3.

Example

This example illustrates the use of a DECLARE DATABASE directive:

```
EXEC SQL DECLARE oracle3 DATABASE
```

Related Topics

COMMIT command on F – 9
CONNECT command on F – 12
DECLARE CURSOR command on F – 17
DECLARE STATEMENT command on F – 20
DELETE command on F – 23
EXECUTE command on F – 32
EXECUTE IMMEDIATE command on F – 34
INSERT command on F – 39
SELECT command on F – 50
UPDATE command on F – 56

DECLARE STATEMENT (Embedded SQL Directive)

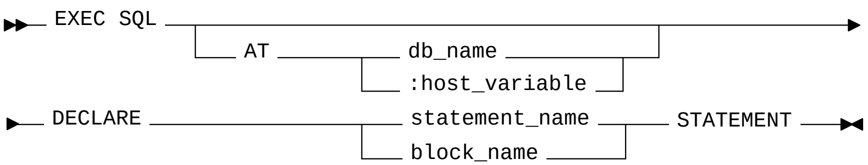
Purpose

To declare an identifier for a SQL statement or PL/SQL block to be used in other embedded SQL statements.

Prerequisites

None.

Syntax



Keywords and Parameters

AT

identifies the database on which the SQL statement or PL/SQL block is declared. The database can be identified by either:

db_name is a database identifier declared in a previous DECLARE DATABASE statement.

:*host_variable* is a host variable whose value is a previously declared *db_name.*

If you omit this clause, Oracle7 declares the SQL statement or PL/SQL block on your default database.

statement_name
block_name

is the declared identifier for the statement.

Usage Notes

You must declare an identifier for a SQL statement or PL/SQL block with a DECLARE STATEMENT statement only if a DECLARE CURSOR statement referencing the identifier appears physically (not logically) in the embedded SQL program before the PREPARE statement that parses the statement or block and associates it with its identifier.

The scope of a statement declaration is global within its precompilation unit, like a cursor declaration. For more information on this command, see Chapters 3 and 11.

Example I

This example illustrates the use of the DECLARE STATEMENT statement:

```
EXEC SQL AT remote_db
    DECLARE my_statement STATEMENT
EXEC SQL PREPARE my_statement FROM :my_string
EXEC SQL EXECUTE my_statement
```

Example II

In this example from a Pro*C/C++ embedded SQL program, the DECLARE STATEMENT statement is required because the DECLARE CURSOR statement precedes the PREPARE statement:

```
EXEC SQL DECLARE my_statement STATEMENT;
EXEC SQL DECLARE emp_cursor CURSOR FOR my_statement;
EXEC SQL PREPARE my_statement FROM :my_string;
...
```

Related Topics

CLOSE command on F – 8
DECLARE DATABASE command on F – 19
FETCH command on F – 36
PREPARE command on F – 45
OPEN command on F – 43

DECLARE TABLE (Oracle Embedded SQL Directive)

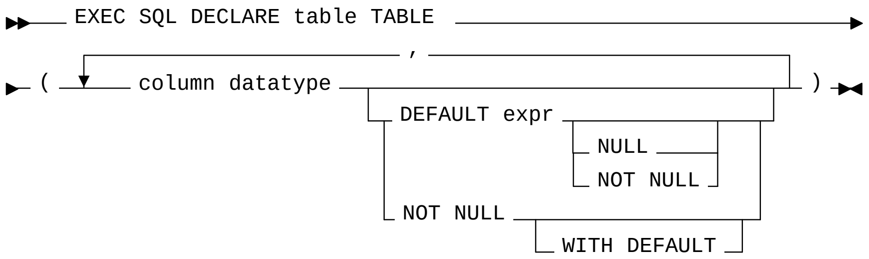
Purpose

To define the structure of a table or view, including each column's datatype, default value, and NULL or NOT NULL specification for semantic checking by the Oracle Precompilers.

Prerequisites

None.

Syntax



Keywords and Parameters

table	is the name of the declared table.
column	is a column of the *table*.
datatype	is the datatype of a *column*. For information on Oracle7 datatypes, see Chapter 3.
DEFAULT	specifies the default value of a *column*.
NULL	specifies that a *column* can contain nulls.
NOT NULL	specifies that a *column* cannot contain nulls.
WITH DEFAULT	is supported for compatibility with the IBM DB2 database.

Usage Notes

For information on using this command, see Chapter 3.

Example

The following statement declares the PARTS table with the PARTNO, BIN, and QTY columns:

```
EXEC SQL DECLARE parts TABLE
    (partno  NUMBER  NOT NULL,
     bin     NUMBER,
     qty     NUMBER)
```

Related Topics

None.

DELETE (Executable Embedded SQL)

Purpose

To remove rows from a table or from a view’s base table.

Prerequisites

For you to delete rows from a table, the table must be in your own schema or you must have DELETE privilege on the table.

For you to delete rows from the base table of a view, the owner of the schema containing the view must have DELETE privilege on the base table. Also, if the view is in a schema other than your own, you must be granted DELETE privilege on the view.

The DELETE ANY TABLE system privilege also allows you to delete rows from any table or any view’s base table.

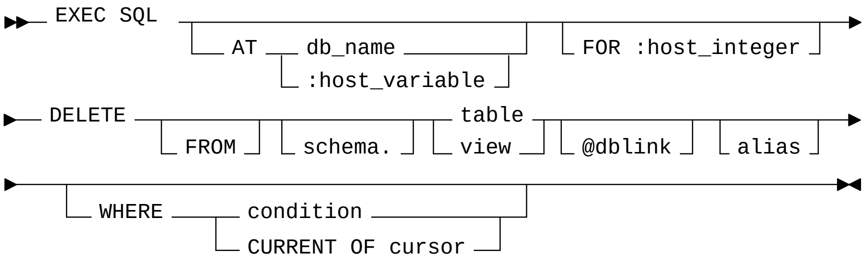
If you are using Trusted Oracle7 in DBMS MAC mode, your DBMS label must dominate the creation label of the table or view or you must meet one of the following criteria:

- If the creation label of the table or view is higher than your DBMS label, you must have READUP and WRITEUP system privileges.
- If the creation label of your table or view is not comparable to your DBMS label, you must have READUP, WRITEUP, and WRITEDOWN system privileges.

In addition, for each row to be deleted, your DBMS label must match the row’s label or you must meet one of the following criteria:

- If the row’s label is higher than your DBMS label, you must have READUP and WRITEUP system privileges.
- If the row’s label is lower than your DBMS label, you must have WRITEDOWN system privilege.
- If the row label is not comparable to your DBMS label, you must have READUP, WRITEUP, and WRITEDOWN system privileges.

Syntax



Keywords and Parameters

AT

identifies the database to which the DELETE statement is issued. The database can be identified by either:

- *db_name* — is a database identifier declared in a previous DECLARE DATABASE statement.
- *:host_variable* — is a host variable whose value is a previously declared *db_name*.

If you omit this clause, the DELETE statement is issued to your default database.

FOR *:host_integer*

limits the number of times the statement is executed if the WHERE clause contains array host variables. If you omit this clause, Oracle7 executes the statement once for each component of the smallest array.

schema

is the schema containing the table or view. If you omit *schema*, Oracle7 assumes the table or view is in your own schema.

table
view

is the name of a table from which the rows are to be deleted. If you specify *view*, Oracle7 deletes rows from the view's base table.

dblink

is the complete or partial name of a database link to a remote database where the table or view is located. For information on referring to database links, see Chapter 2 of the *Oracle7 Server SQL Reference*. You can only delete rows from a remote table or view if you are using Oracle7 with the distributed option.

If you omit *dblink*, Oracle7 assumes that the table or view is located on the local database.

alias

is an alias assigned to the table. Aliases are generally used in DELETE statements with correlated queries.

WHERE specifies which rows are deleted:

condition deletes only rows that satisfy the condition. This condition can contain host variables and optional indicator variables. See the syntax description of *condition* in Chapter 3 of the *Oracle7 Server SQL Reference.*

CURRENT OF deletes only the row most recently fetched by the *cursor*. The *cursor* cannot be associated with a SELECT statement that performs a join, unless its FOR UPDATE clause specifically locks only one table.

If you omit this clause entirely, Oracle7 deletes all rows from the table or view.

Usage Notes

The host variables in the WHERE clause must be either all scalars or all arrays. If they are scalars, Oracle7 executes the DELETE statement only once. If they are arrays, Oracle7 executes the statement once for each set of array components. Each execution may delete zero, one, or multiple rows.

Array host variables in the WHERE clause can have different sizes. In this case, the number of times Oracle7 executes the statement is determined by the smaller of the following values:

- the size of the smallest array
- the value of the :*host_integer* in the optional FOR clause

If no rows satisfy the condition, no rows are deleted and the SQLCODE returns a NOT_FOUND condition.

The cumulative number of rows deleted is returned through the SQLCA. If the WHERE clause contains array host variables, this value reflects the total number of rows deleted for all components of the array processed by the DELETE statement.

If no rows satisfy the condition, Oracle7 returns an error through the SQLCODE of the SQLCA. If you omit the WHERE clause, Oracle7 raises a warning flag in the fifth component of SQLWARN in the SQLCA. For more information on this command and the SQLCA, see Chapter 9.

You can use comments in a DELETE statement to pass instructions, or *hints*, to the Oracle7 optimizer. The optimizer uses hints to choose an execution plan for the statement. For more information on hints, see *Oracle7 Server Tuning*.

Example

This example illustrates the use of the DELETE statement within a Pro*C/C++ embedded SQL program:

```
EXEC SQL DELETE FROM emp
    WHERE deptno = :deptno
    AND job = :job; ...
EXEC SQL DECLARE emp_cursor CURSOR
    FOR SELECT empno, comm
        FROM emp;
EXEC SQL OPEN emp_cursor;
EXEC SQL FETCH c1
    INTO :emp_number, :commission;
EXEC SQL DELETE FROM emp
    WHERE CURRENT OF emp_cursor;
```

Related Topics

DECLARE DATABASE command on F – 19
DECLARE STATEMENT command on F – 20

DESCRIBE (Executable Embedded SQL)

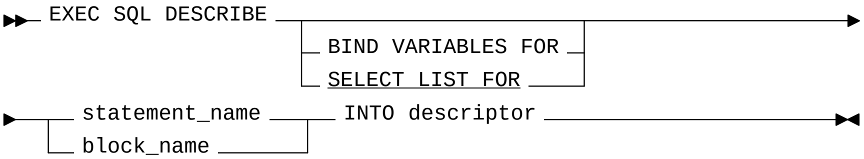
Purpose

To initialize a descriptor to hold descriptions of host variables for a dynamic SQL statement or PL / SQL block.

Prerequisites

You must have prepared the SQL statement or PL / SQL block in a previous embedded SQL PREPARE statement.

Syntax



Keywords and Parameters

BIND VARIABLES — initializes the descriptor to hold information about the input variables for the SQL statement or PL / SQL block.

SELECT LIST — initializes the descriptor to hold information about the select list of a SELECT statement.

The default is SELECT LIST FOR.

statement_name
block_name — identifies a SQL statement or PL / SQL block previously prepared with a PREPARE statement.

descriptor — is the name of the descriptor to be initialized.

Usage Notes

You must issue a DESCRIBE statement before manipulating the bind or select descriptor within an embedded SQL program.

You cannot describe both input variables and output variables into the same descriptor.

The number of variables found by a DESCRIBE statement is the total number of placeholders in the prepare SQL statement or PL / SQL block, rather than the total number of uniquely named placeholders. For more information on this command, see Chapter 11.

Example

This example illustrates the use of the DESCRIBE statement in a Pro*C embedded SQL program:

```
EXEC SQL PREPARE my_statement FROM :my_string;
EXEC SQL DECLARE emp_cursor
    FOR SELECT empno, ename, sal, comm
            FROM emp
            WHERE deptno = :dept_number
EXEC SQL DESCRIBE BIND VARIABLES FOR my_statement
    INTO bind_descriptor;
EXEC SQL OPEN emp_cursor
    USING bind_descriptor;
EXEC SQL DESCRIBE SELECT LIST FOR my_statement
    INTO select_descriptor;
EXEC SQL FETCH emp_cursor
    INTO select_descriptor;
```

Related Topics

PREPARE command on F – 45

ENABLE THREADS (Executable Embedded SQL Extension)

Purpose

To initialize a process that supports multiple threads.

Prerequisites

You must be developing a precompiler application for and compiling it on a platform that supports multi–threaded applications, and THREADS=YES must be specified on the command line.

Syntax

```
►►—— EXEC SQL ENABLE THREADS ————————————————————————————►◄
```

Keywords and Parameters

None.

Usage Notes

The ENABLE THREADS command must be executed before the first executable SQL statement and before spawning any thread. This command does not require a host–variable specification.

For more information on this command, see "Developing Multi–threaded Applications" on page 3 – 99.

Example

This example illustrates the use of the ENABLE THREADS command in a Pro*C / C++ embedded SQL program:

```
EXEC SQL ENABLE THREADS;
```

Related Topics

CONTEXT ALLOCATE on F – 14
CONTEXT FREE on F – 15
CONTEXT USE on F – 16

EXECUTE ... END–EXEC (Executable Embedded SQL Extension)

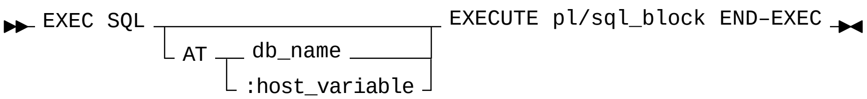
Purpose

To embed an anonymous PL/SQL block into an Oracle Precompiler program.

Prerequisites

None.

Syntax



Keywords and Parameters

AT

identifies the database on which the PL/SQL block is executed. The database can be identified by either:

db_name — is a database identifier declared in a previous DECLARE DATABASE statement.

:*host_variable* — is a host variable whose value is a previously declared *db_name*.

If you omit this clause, the PL/SQL block is executed on your default database.

pl/sql_block

For information on PL/SQL, including how to write PL/SQL blocks, see the *PL/SQL User's Guide and Reference.*

END–EXEC

must appear after the embedded PL/SQL block, regardless of which programming language your Oracle Precompiler program uses. Of course, the keyword END–EXEC must be followed by the embedded SQL statement terminator for the specific language.

Usage Notes

Since the Oracle Precompilers treat an embedded PL/SQL block like a single embedded SQL statement, you can embed a PL/SQL block anywhere in an Oracle Precompiler program that you can embed a SQL statement. For more information on embedding PL/SQL blocks in Oracle Precompiler programs, see Chapter 5.

Example

Placing this EXECUTE statement in an Oracle Precompiler program embeds a PL/SQL block in the program:

```
EXEC SQL EXECUTE
    BEGIN
        SELECT ename, job, sal
            INTO :emp_name:ind_name, :job_title, :salary
            FROM emp
            WHERE empno = :emp_number;
        IF :emp_name:ind_name IS NULL
            THEN RAISE name_missing;
        END IF;
    END;
END–EXEC
```

Related Topics

EXECUTE IMMEDIATE embedded SQL command on F – 34

EXECUTE (Executable Embedded SQL)

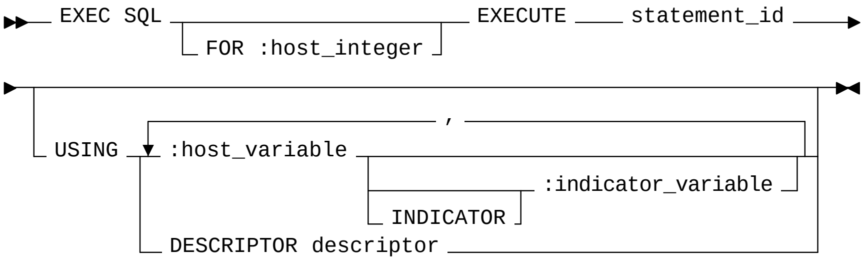
Purpose

To execute a DELETE, INSERT, or UPDATE statement or a PL/SQL block that has been previously prepared with an embedded SQL PREPARE statement.

Prerequisites

You must first prepare the SQL statement or PL/SQL block with an embedded SQL PREPARE statement.

Syntax



Keywords and Parameters

FOR :*host_integer* — limits the number of times the statement is executed when the USING clause contains array host variables If you omit this clause, Oracle7 executes the statement once for each component of the smallest array.

statement_id — is a precompiler identifier associated with the SQL statement or PL/SQL block to be executed. Use the embedded SQL PREPARE command to associate the precompiler identifier with the statement or PL/SQL block.

USING — specifies a list of host variables with optional indicator variables that Oracle7 substitutes as input variables into the statement to be executed. The host and indicator variables must be either all scalars or all arrays.

Usage Notes

For more information on this command, see Chapter 11.

Example

This example illustrates the use of the EXECUTE statement in a Pro*C/C++ embedded SQL program:

```
EXEC SQL PREPARE my_statement
    FROM :my_string;
EXEC SQL EXECUTE my_statement
    USING :my_var;
```

Related Topics

DECLARE DATABASE command on F – 19
PREPARE command on F – 45

EXECUTE IMMEDIATE (Executable Embedded SQL)

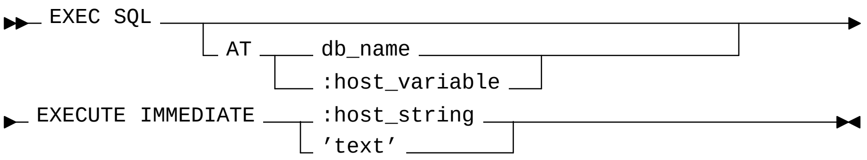
Purpose

To prepare and execute a DELETE, INSERT, or UPDATE statement or a PL/SQL block containing no host variables.

Prerequisites

None.

Syntax



Keywords and Parameters

AT

identifies the database on which the SQL statement or PL/SQL block is executed. The database can be identified by either:

db_name is a database identifier declared in a previous DECLARE DATABASE statement.

:*host_variable* is a host variable whose value is a previously declared *db_name.*

If you omit this clause, the statement or block is executed on your default database.

:*host_string*

is a host variable whose value is the SQL statement or PL/SQL block to be executed.

text

is a quoted text literal containing the SQL statement or PL/SQL block to be executed.

The SQL statement can only be a DELETE, INSERT, or UPDATE statement.

Usage Notes

When you issue an EXECUTE IMMEDIATE statement, Oracle7 parses the specified SQL statement or PL/SQL block, checking for errors, and executes it. If any errors are encountered, they are returned in the SQLCODE component of the SQLCA.

For more information on this command, see Chapter 11.

Example

This example illustrates the use of the EXECUTE IMMEDIATE statement:

```
EXEC SQL EXECUTE IMMEDIATE ’DELETE FROM emp WHERE empno = 9460’
```

Related Topics

PREPARE command on F – 45
EXECUTE command on F – 32

FETCH (Executable Embedded SQL)

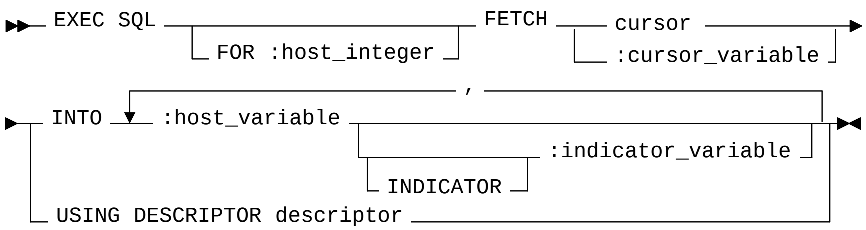
Purpose

To retrieve one or more rows returned by a query, assigning the select list values to host variables.

Prerequisites

You must first open the cursor with an the OPEN statement.

Syntax



Keywords and Parameters

FOR :*host_integer* — limits the number of rows fetched if you are using array host variables. If you omit this clause, Oracle7 fetches enough rows to fill the smallest array.

cursor — is a cursor that is declared by a DECLARE CURSOR statement. The FETCH statement returns one of the rows selected by the query associated with the cursor.

:*cursor_variable* — is a cursor variable is allocated an ALLOCATE statement. The FETCH statement returns one of the rows selected by the query associated with the cursor variable.

INTO — specifies a list of host variables and optional indicator variables into which data is fetched. These host variables and indicator variables must be declared within the program.

USING — specifies the descriptor referenced in a previous DESCRIBE statement. Only use this clause with dynamic embedded SQL, method 4. Also, the USING clause does not apply when a cursor variable is used.

Usage Notes

The FETCH statement reads the rows of the active set and names the output variables which contain the results. Indicator values are set to –1 if their associated host variable is null. The first FETCH statement for a cursor also sorts the rows of the active set, if necessary.

The number of rows retrieved is specified by the size of the output host variables and the value specified in the FOR clause. The host variables to receive the data must be either all scalars or all arrays. If they are scalars, Oracle7 fetches only one row. If they are arrays, Oracle7 fetches enough rows to fill the arrays.

Array host variables can have different sizes. In this case, the number of rows Oracle7 fetches is determined by the smaller of the following values:

- the size of the smallest array
- the value of the :*host_integer* in the optional FOR clause

Of course, the number of rows fetched can be further limited by the number of rows that actually satisfy the query.

If a FETCH statement does not retrieve all rows returned by the query, the cursor is positioned on the next returned row. When the last row returned by the query has been retrieved, the next FETCH statement results in an error code returned in the SQLCODE element of the SQLCA.

Note that the FETCH command does not contain an AT clause. You must specify the database accessed by the cursor in the DECLARE CURSOR statement.

You can only move forward through the active set with FETCH statements. If you want to revisit any of the previously fetched rows, you must reopen the cursor and fetch each row in turn. If you want to change the active set, you must assign new values to the input host variables in the cursor's query and reopen the cursor.

Example

This example illustrates the FETCH command in a pseudo–code embedded SQL program:

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
    SELECT job, sal FROM emp WHERE deptno = 30;
...
EXEC SQL WHENEVER NOT FOUND GOTO ...
LOOP
    EXEC SQL FETCH emp_cursor INTO :job_title1, :salary1;
    EXEC SQL FETCH emp_cursor INTO :job_title2, :salary2;
...
END LOOP;
...
```

Related Topics

PREPARE command on F – 45
DECLARE CURSOR command on F – 17
OPEN command on F – 43
CLOSE command on F – 8

INSERT (Executable Embedded SQL)

Purpose To add rows to a table or to a view's base table.

Prerequisites For you to insert rows into a table, the table must be in your own schema or you must have INSERT privilege on the table.

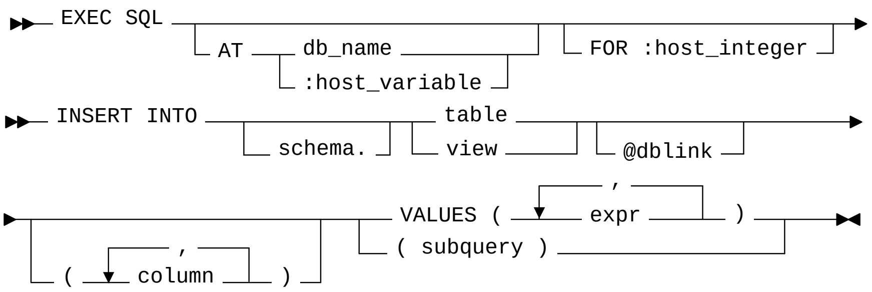
For you to insert rows into the base table of a view, the owner of the schema containing the view must have INSERT privilege on the base table. Also, if the view is in a schema other than your own, you must have INSERT privilege on the view.

The INSERT ANY TABLE system privilege also allows you to insert rows into any table or any view's base table.

If you are using Trusted Oracle7 in DBMS MAC mode, your DBMS label must match the creation label of the table or view:

- If the creation label of the table or view is higher than your DBMS label, you must have WRITEUP system privileges.
- If the creation label of the table or view is lower than your DBMS label, you must have WRITEDOWN system privilege.
- If the creation label of your table or view is not comparable to your DBMS label, you must have WRITEUP and WRITEDOWN system privileges.

Syntax



Keywords and Parameters

AT — identifies the database on which the INSERT statement is executed. The database can be identified by either:

- *db_name* — is a database identifier declared in a previous DECLARE DATABASE statement.
- :*host_variable* — is a host variable whose value is a previously declared *db_name*

If you omit this clause, the INSERT statement is executed on your default database.

FOR :*host_integer* — limits the number of times the statement is executed if the VALUES clause contains array host variables. If you omit this clause, Oracle7 executes the statement once for each component in the smallest array.

schema — is the schema containing the table or view. If you omit *schema*, Oracle7 assumes the table or view is in your own schema.

table
view — is the name of the table into which rows are to be inserted. If you specify *view*, Oracle7 inserts rows into the view's base table.

dblink — is a complete or partial name of a database link to a remote database where the table or view is located. For information on referring to database links, see Chapter 2 of the *Oracle7 Server SQL Reference*. You can only insert rows into a remote table or view if you are using Oracle7 with the distributed option.

If you omit *dblink*, Oracle7 assumes that the table or view is on the local database.

column

is a column of the table or view. In the inserted row, each column in this list is assigned a value from the VALUES clause or the query.

If you omit one of the table's columns from this list, the column's value for the inserted row is the column's default value as specified when the table was created. If you omit the column list altogether, the VALUES clause or query must specify values for all columns in the table.

VALUES

specifies a row of values to be inserted into the table or view. See the syntax description of *expr* in Chapter 3 of the *Oracle7 Server SQL Reference*. Note that the expressions can be host variables with optional indicator variables. You must specify an expression in the VALUES clause for each column in the column list.

subquery

is a subquery that returns rows that are inserted into the table. The select list of this subquery must have the same number of columns as the column list of the INSERT statement. For the syntax description of a subquery, see "SELECT" in Chapter 4 of the *Oracle7 Server SQL Reference*.

Usage Notes

Any host variables that appear in the WHERE clause must be either all scalars or all arrays. If they are scalars, Oracle7 executes the INSERT statement once. If they are arrays, Oracle7 executes the INSERT statement once for each set of array components, inserting one row each time.

Array host variables in the WHERE clause can have different sizes. In this case, the number of times Oracle7 executes the statement is determined by the smaller of the following values:

- size of the smallest array
- the value of the :*host_integer* in the optional FOR clause.

For more information on this command, see Chapter 4.

Example I

This example illustrates the use of the embedded SQL INSERT command:

```
EXEC SQL
    INSERT INTO emp (ename, empno, sal)
    VALUES (:ename, :empno, :sal);
```

Example II

This example shows an embedded SQL INSERT command with a subquery:

```
EXEC SQL
    INSERT INTO new_emp (ename, empno, sal)
    SELECT ename, empno, sal FROM emp
    WHERE deptno = :deptno;
```

Related Topics

DECLARE DATABASE command on F – 19

OPEN (Executable Embedded SQL)

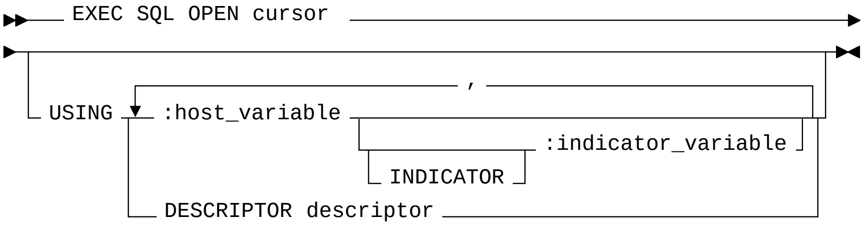
Purpose

To open a cursor, evaluating the associated query and substituting the host variable names supplied by the USING clause into the WHERE clause of the query.

Prerequisites

You must declare the cursor with a DECLARE CURSOR embedded SQL statement before opening it.

Syntax



Keywords and Parameters

cursor is the cursor to be opened.

USING specifies the host variables to be substituted into the WHERE clause of the associated query.

:*host_variable* specifies a host variable with an optional indicator variable to be substituted into the statement associated with the cursor.

DESCRIPTOR specifies a descriptor that describes the host variables to be substituted into the WHERE clause of the associated query. The *descriptor* must be initialized in a previous DESCRIBE statement.

The substitution is based on position. The host variable names specified in this statement can be different from the variable names in the associated query.

Usage Notes

The OPEN command defines the active set of rows and initializes the cursor just before the first row of the active set. The values of the host variables at the time of the OPEN are substituted in the statement. This command does not actually retrieve rows; rows are retrieved by the FETCH command.

Once you have opened a cursor, its input host variables are not reexamined until you reopen the cursor. To change any input host variables and therefore the active set, you must reopen the cursor.

All cursors in a program are in a closed state when the program is initiated or when they have been explicitly closed using the CLOSE command.

You can reopen a cursor without first closing it. For more information on this command, see Chapter 4.

Example

This example illustrates the use of the OPEN command in a Pro*C/C++ embedded SQL program:

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
    SELECT ename, empno, job, sal
    FROM emp
    WHERE deptno = :deptno;
EXEC SQL OPEN emp_cursor;
```

Related Topics

PREPARE command on F – 45
DECLARE CURSOR command on F – 17
FETCH command on F – 36
CLOSE command on F – 8

PREPARE (Executable Embedded SQL)

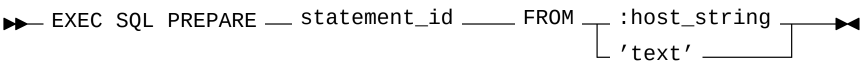
Purpose

To parse a SQL statement or PL/SQL block specified by a host variable and associate it with an identifier.

Prerequisites

None.

Syntax



Keywords and Parameters

statement_id is the identifier to be associated with the prepared SQL statement or PL/SQL block. If this identifier was previously assigned to another statement or block, the prior assignment is superseded.

:*host_string* is a host variable whose value is the text of a SQL statement or PL/SQL block to be prepared.

text is a string literal containing a SQL statement or PL/SQL block to be prepared.

Usage Notes

Any variables that appear in the :*host_string* or *text* are placeholders. The actual host variable names are assigned in the USING clause of the OPEN command (input host variables) or in the INTO clause of the FETCH command (output host variables).

A SQL statement is prepared only once, but can be executed any number of times.

Example

This example illustrates the use of a PREPARE statement in a Pro*C/C++ embedded SQL program:

```
EXEC SQL PREPARE my_statement FROM :my_string;
EXEC SQL EXECUTE my_statement;
```

Related Topics

DECLARE CURSOR command on F – 17
OPEN command on F – 43
FETCH command on F – 36
CLOSE command on F – 8

ROLLBACK (Executable Embedded SQL)

Purpose

To undo work done in the current transaction.

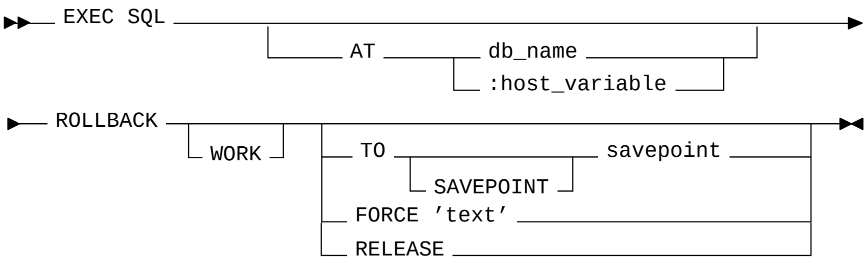
You can also use this command to manually undo the work done by an in–doubt distributed transaction.

Prerequisites

To roll back your current transaction, no privileges are necessary.

To manually roll back an in–doubt distributed transaction that you originally committed, you must have FORCE TRANSACTION system privilege. To manually roll back an in–doubt distributed transaction originally committed by another user, you must have FORCE ANY TRANSACTION system privilege.

Syntax



Keywords and Parameters

WORK — is optional and is provided for ANSI compatibility.

TO — rolls back the current transaction to the specified savepoint. If you omit this clause, the ROLLBACK statement rolls back the entire transaction.

FORCE — manually rolls back an in–doubt distributed transaction. The transaction is identified by the *text* containing its local or global transaction ID. To find the IDs of such transactions, query the data dictionary view DBA_2PC_PENDING.

ROLLBACK statements with the FORCE clause are not supported in PL/SQL.

RELEASE — frees all resources and disconnects the application from the Oracle7 Server. The RELEASE clause is not allowed with SAVEPOINT and FORCE clauses.

Usage Notes

A transaction (or a logical unit of work) is a sequence of SQL statements that Oracle7 treats as a single unit. A transaction begins with the first executable SQL statement after a COMMIT, ROLLBACK or connection to the database. A transaction ends with a COMMIT statement, a ROLLBACK statement, or disconnection (intentional or unintentional) from the database. Note that Oracle7 issues an implicit COMMIT statement before and after processing any Data Definition Language statement.

Using the ROLLBACK command without the TO SAVEPOINT clause performs the following operations:

- ends the transaction
- undoes all changes in the current transaction
- erases all savepoints in the transaction
- releases the transaction's locks

Using the ROLLBACK command with the TO SAVEPOINT clause performs the following operations:

- rolls back just the portion of the transaction after the savepoint.
- loses all savepoints created after that savepoint. Note that the named savepoint is retained, so you can roll back to the same savepoint multiple times. Prior savepoints are also retained.
- releases all table and row locks acquired since the savepoint. Note that other transactions that have requested access to rows locked after the savepoint must continue to wait until the transaction is committed or rolled back. Other transactions that have not already requested the rows can request and access the rows immediately.

It is recommended that you explicitly end transactions in application programs using either a COMMIT or ROLLBACK statement. If you do not explicitly commit the transaction and the program terminates abnormally, Oracle7 rolls back the last uncommitted transaction.

Example I

The following statement rolls back your entire current transaction:

```
EXEC SQL ROLLBACK;
```

Example II

The following statement rolls back your current transaction to savepoint SP5:

```
EXEC SQL ROLLBACK TO SAVEPOINT sp5;
```

Distributed Transactions

Oracle7 with the distributed option allows you to perform distributed transactions, or transactions that modify data on multiple databases. To commit or roll back a distributed transaction, you need only issue a COMMIT or ROLLBACK statement as you would any other transaction.

If there is a network failure during the commit process for a distributed transaction, the state of the transaction may be unknown, or in–doubt. After consultation with the administrators of the other databases involved in the transaction, you may decide to manually commit or roll back the transaction on your local database. You can manually roll back the transaction on your local database by issuing a ROLLBACK statement with the FORCE clause.

For more information on when to roll back in–doubt transactions, see *Oracle7 Server Distributed Systems, Volume I.*

You cannot manually roll back an in–doubt transaction to a savepoint.

A ROLLBACK statement with a FORCE clause only rolls back the specified transaction. Such a statement does not affect your current transaction.

Example III

The following statement manually rolls back an in–doubt distributed transaction:

```
EXEC SQL
    ROLLBACK WORK
    FORCE '25.32.87';
```

Related Topics

COMMIT command on F – 9
SAVEPOINT command on F – 49

SAVEPOINT (Executable Embedded SQL)

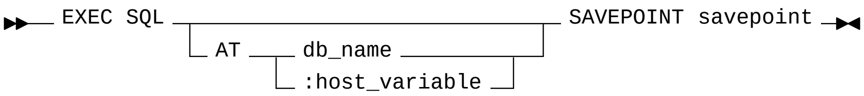
Purpose

To identify a point in a transaction to which you can later roll back.

Prerequisites

None.

Syntax



Keywords and Parameters

AT — identifies the database on which the savepoint is created. The database can be identified by either:

db_name — is a database identifier declared in a previous DECLARE DATABASE statement.

:*host_variable* — is a host variable whose value is a previously declared *db_name.*

If you omit this clause, the savepoint is created on your default database.

savepoint — is the name of the savepoint to be created.

Usage Notes

For more information on this command, see Chapter 8.

Example

This example illustrates the use of the embedded SQL SAVEPOINT command:

```
EXEC SQL SAVEPOINT save3;
```

Related Topics

COMMIT command on F – 9
ROLLBACK command on F – 46

SELECT (Executable Embedded SQL)

Purpose

To retrieve data from one or more tables, views, or snapshots, assigning the selected values to host variables.

Prerequisites

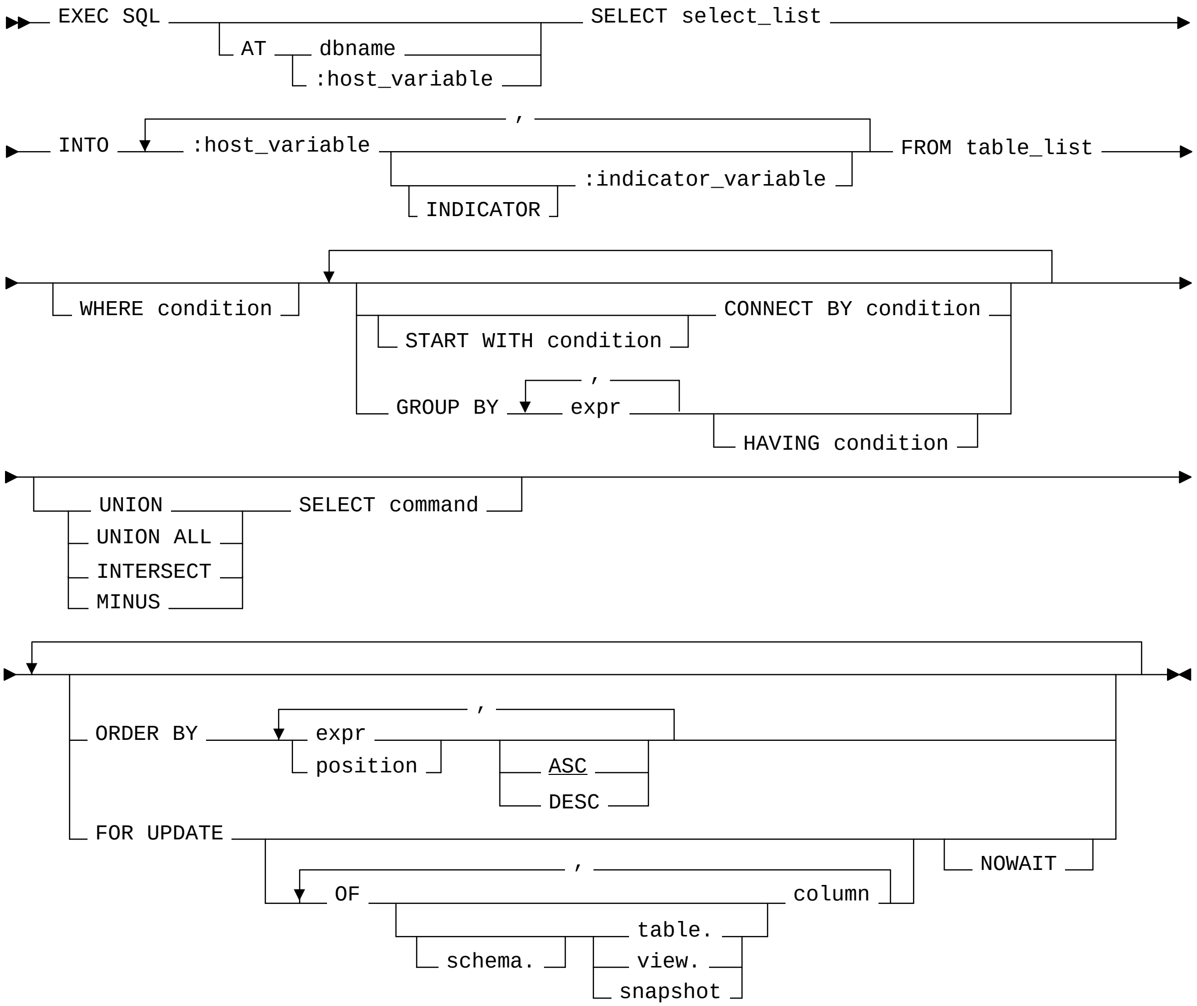
For you to select data from a table or snapshot, the table or snapshot must be in your own schema or you must have SELECT privilege on the table or snapshot.

For you to select rows from the base tables of a view, the owner of the schema containing the view must have SELECT privilege on the base tables. Also, if the view is in a schema other than your own, you must have SELECT privilege on the view.

The SELECT ANY TABLE system privilege also allows you to select data from any table or any snapshot or any view's base table.

If you are using Trusted Oracle7 in DBMS MAC mode, your DBMS label must dominate the creation label of each queried table, view, or snapshot or you must have READUP system privileges.

Syntax


EXEC SQL
AT
dbname
:host_variable
SELECT select_list
INTO
,
:host_variable
INDICATOR
:indicator_variable
FROM table_list
WHERE condition
START WITH condition
CONNECT BY condition
GROUP BY
,
expr
HAVING condition
UNION
UNION ALL
INTERSECT
MINUS
SELECT command
ORDER BY
,
expr
position
ASC
DESC
FOR UPDATE
,
OF
schema.
table.
view.
snapshot
column
NOWAIT

Keywords and Parameters

AT

identifies the database to which the SELECT statement is issued. The database can be identified by either:

db_name

is a database identifier declared in a previous DECLARE DATABASE statement.

:*host_variable*

is a host variable whose value is a previously declared *db_name.*

If you omit this clause, the SELECT statement is issued to your default database.

select_list

identical to the non–embedded SELECT command except that a host variables can be used in place of literals.

INTO

specifies output host variables and optional indicator variables to receive the data returned by the SELECT statement. Note that these variables must be either all scalars or all arrays, but arrays need not have the same size.

WHERE

restricts the rows returned to those for which the condition is TRUE. See the syntax description of *condition* in Chapter 3 of the *Oracle7 Server SQL Reference.* The *condition* can contain host variables, but cannot contain indicator variables. These host variables can be either scalars or arrays.

All other keywords and parameters are identical to the non–embedded SQL SELECT command.

Usage Notes

If no rows meet the WHERE clause condition, no rows are retrieved and Oracle7 returns an error code through the SQLCODE component of the SQLCA.

You can use comments in a SELECT statement to pass instructions, or *hints*, to the Oracle7 optimizer. The optimizer uses hints to choose an execution plan for the statement. For more information on hints, see *Oracle7 Server Tuning*.

Example

This example illustrates the use of the embedded SQL SELECT command:

```
EXEC SQL SELECT ename, sal + 100, job
    INTO :ename, :sal, :job
    FROM emp
    WHERE empno = :empno
```

Related Topics

DECLARE DATABASE command on F – 19
DECLARE CURSOR command on F – 17
EXECUTE command on F – 32
FETCH command on F – 36
PREPARE command on F – 46

TYPE (Oracle Embedded SQL Directive)

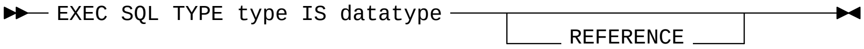
Purpose

To perform *user–defined type equivalencing*, or to assign an Oracle7 external datatype to a whole class of host variables by equivalencing the external datatype to a user–defined datatype.

Prerequisites

The user–defined datatype must be previously declared in an embedded SQL program.

Syntax



Keywords and Parameters

type is the user–defined datatype to be equivalenced with an Oracle7 external datatype.

datatype is an Oracle7 external datatype recognized by the Oracle Precompilers (not an Oracle7 internal datatype). The datatype may include a length, precision, or scale. This external datatype is equivalenced to the user–defined *type* and assigned to all host variables assigned the *type*. For a list of external datatypes, see Chapter 3.

REFERENCE makes the equivalenced type a pointer type.

Usage Notes

User defined type equivalencing is one kind of datatype equivalencing. You can only perform user–defined type equivalencing with the embedded SQL TYPE command in a Pro*C/C++ Precompiler program. You may want to use datatype equivalencing for one of the following purposes:

- to automatically null–terminate a character host variable
- to store program data as binary data in the database
- to override default datatype conversion

All Oracle Precompilers also support the embedded SQL VAR command for host variable equivalencing.

Example

This example shows an embedded SQL TYPE statement in a Pro*C / C++ Precompiler program:

```
struct screen {
    short len;
    char  buff[4002];
};

typedef struct screen graphics;

EXEC SQL BEGIN DECLARE SECTION;
    EXEC SQL TYPE graphics IS VARRAW (4002);
    graphics crt;  -- host variable of type graphics
    ...
EXEC SQL END DECLARE SECTION;
```

Related Topics

VAR directive on page F – 60

UPDATE (Executable Embedded SQL)

Purpose

To change existing values in a table or in a view's base table.

Prerequisites

For you to update values in a table or snapshot, the table must be in your own schema or you must have UPDATE privilege on the table.

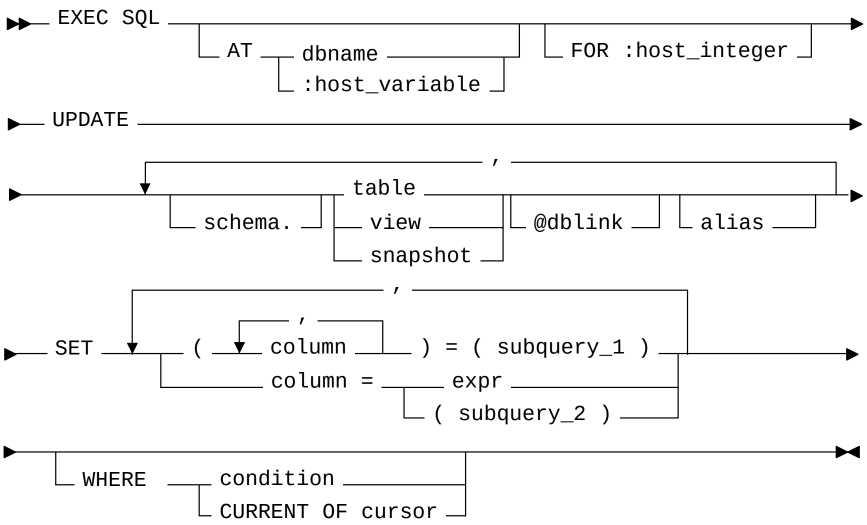
For you to update values in the base table of a view, the owner of the schema containing the view must have UPDATE privilege on the base table. Also, if the view is in a schema other than your own, you must have UPDATE privilege on the view.

The UPDATE ANY TABLE system privilege also allows you to update values in any table or any view's base table.

If you are using Trusted Oracle7 in DBMS MAC mode, your DBMS label must match the creation label of the table or view:

- If the creation label of the table or view is higher than your DBMS label, you must have READUP and WRITEUP system privileges
- If the creation label of the table or view is lower than your DBMS label, you must have WRITEDOWN system privilege.
- If the creation label of your table or view is not comparable to your DBMS label, you must have READUP, WRITEUP, and WRITEDOWN system privileges.

Syntax



Keywords and Parameters

AT

identifies the database to which the UPDATE statement is issued. The database can be identified by either:

db_name is a database identifier declared in a previous DECLARE DATABASE statement.

:*host_variable* is a host variable whose value is a previously declared *db_name.*

If you omit this clause, the UPDATE statement is issued to your default database.

FOR :*host_integer*

limits the number of times the UPDATE statement is executed if the SET and WHERE clauses contain array host variables. If you omit this clause, Oracle7 executes the statement once for each component of the smallest array.

schema

is the schema containing the table or view. If you omit *schema,* Oracle7 assumes the table or view is in your own schema.

table
view

is the name of the table to be updated. If you specify *view,* Oracle7 updates the view's base table.

dblink

is a complete or partial name of a database link to a remote database where the table or view is located. For information on referring to database links, see Chapter 2 of the *Oracle7 Server SQL Reference.* You can only use a database link to update a remote table or view if you are using Oracle7 with the distributed option.

alias

is a name used to reference the table, view, or subquery elsewhere in the statement.

column

is the name of a column of the table or view that is to be updated. If you omit a column of the table from the SET clause, that column's value remains unchanged.

expr is the new value assigned to the corresponding column. This expression can contain host variables and optional indicator variables. See the syntax of *expr* in Chapter 3 of the *Oracle7 Server SQL Reference.*

subquery_1 is a subquery that returns new values that are assigned to the corresponding columns. For the syntax of a subquery, see “SELECT” in Chapter 4 of the *Oracle7 Server SQL Reference.*

subquery_2 is a subquery that return a new value that is assigned to the corresponding column. For the syntax of a subquery, see “SELECT” in Chapter 4 of the *Oracle7 Server SQL Reference.*

WHERE specifies which rows of the table or view are updated:

condition updates only rows for which this condition is true. This condition can contain host variables and optional indicator variables. See the syntax of *condition* in Chapter 3 of the *Oracle7 Server SQL Reference.*

CURRENT OF updates only the row most recently fetched by the *cursor.* The *cursor* cannot be associated with a SELECT statement that performs a join unless its FOR UPDATE clause explicitly locks only one table.

If you omit this clause entirely, Oracle7 updates all rows of the table or view.

Usage Notes

Host variables in the SET and WHERE clauses must be either all scalars or all arrays. If they are scalars, Oracle7 executes the UPDATE statement only once. If they are arrays, Oracle7 executes the statement once for each set of array components. Each execution may update zero, one, or multiple rows.

Array host variables can have different sizes. In this case, the number of times Oracle7 executes the statement is determined by the smaller of the following values:

- the size of the smallest array
- the value of the *:host_integer* in the optional FOR clause

The cumulative number of rows updated is returned through the third element of the SQLERRD component of the SQLCA. When arrays are used as input host variables, this count reflects the total number of updates for all components of the array processed in the UPDATE statement. If no rows satisfy the condition, no rows are updated and Oracle7 returns an error message through the SQLCODE element of the SQLCA. If you omit the WHERE clause, all rows are updated and Oracle7 raises a warning flag in the fifth component of the SQLWARN element of the SQLCA.

You can use comments in an UPDATE statement to pass instructions, or *hints*, to the Oracle7 optimizer. The optimizer uses hints to choose an execution plan for the statement. For more information on hints, see *Oracle7 Server Tuning*.

For more information on this command, see Chapters 4 and 8.

Examples

The following examples illustrate the use of the embedded SQL UPDATE command:

```
EXEC SQL UPDATE emp
    SET sal = :sal, comm = :comm INDICATOR :comm_ind
    WHERE ename = :ename;

EXEC SQL UPDATE emp
    SET (sal, comm) =
        (SELECT AVG(sal)*1.1, AVG(comm)*1.1
         FROM emp)
    WHERE ename = ’JONES’;
```

Related Topics

DECLARE DATABASE command on F – 19

VAR (Oracle Embedded SQL Directive)

Purpose

To perform *host variable equivalencing*, or to assign a specific Oracle7 external datatype to an individual host variable, overriding the default datatype assignment.

Prerequisites

The host variable is optionally declared in the Declare Section of the embedded SQL program.

Syntax

```
►►— EXEC SQL VAR host_variable IS datatype ——————————————————►◄
```

Keywords and Parameters

host_variable is the host variable to be assigned an Oracle7 external datatype.

datatype is an Oracle7 external datatype recognized by the Oracle Precompilers (not an Oracle7 internal datatype). The datatype may include a length, precision, or scale. This external datatype is assigned to the *host_variable*. For a list of external datatypes, see Chapter 3.

Usage Notes

Host variable equivalencing is one kind of datatype equivalencing. Datatype equivalencing is useful for any of the following purposes:

- to automatically null–terminate a character host variable
- to store program data as binary data in the database
- to override default datatype conversion

The Pro*C/C++ Precompiler also supports the precompiler TYPE directive for user–defined type equivalencing.

Example

This example equivalences the host variable DEPT_NAME to the datatype STRING and the host variable BUFFER to the datatype RAW(2000):

```
EXEC SQL BEGIN DECLARE SECTION;
    ...
    char dept_name[15];  –– default datatype is CHAR
    EXEC SQL VAR dept_name IS STRING; –– reset to STRING
    ...
    char buffer[200]; –– default datatype is CHAR
    EXEC SQL VAR buffer IS RAW(200); –– refer to RAW
    ...
EXEC SQL END DECLARE SECTION;
```

Related Topics

TYPE directive on page F – 54

WHENEVER (Embedded SQL Directive)

Purpose

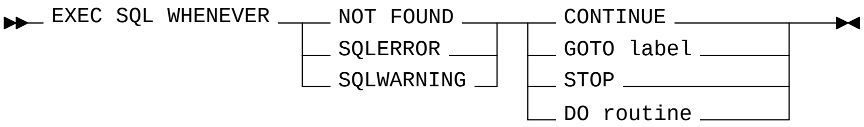
To specify the action to be taken when an error or warning results from executing an embedded SQL program.

Prerequisites

None.

Syntax

The following syntax diagram shows how to construct a WHENEVER statement:



Keywords and Parameters

NOT FOUND — identifies any exception condition that returns an error code of +1403 to SQLCODE (or a +100 code when MODE=ANSI).

SQLERROR — identifies a condition that results in a negative return code.

SQLWARNING — identifies a non–fatal warning condition.

CONTINUE — indicates that the program should progress to the next statement.

GOTO — indicates that the program should branch to the statement named by *label*.

STOP — stops program execution.

DO — indicates that the program should call a C function.

Usage Notes

The WHENEVER command allows your program to transfer control to an error handling routine in the event an embedded SQL statement results in an error or warning.

The scope of a WHENEVER statement is positional, rather than logical. A WHENEVER statement applies to all embedded SQL statements that textually follow it in the source file, not in the flow of the program logic. A WHENEVER statement remains in effect until it is superseded by another WHENEVER statement checking for the same condition.

For more information on this command, see Chapter 8. Do not confuse the WHENEVER embedded SQL command with the WHENEVER SQL*Plus command.

Example

The following example illustrates the use of the WHENEVER command in a Pro*C/C++ embedded SQL program:

```
EXEC SQL WHENEVER NOT FOUND CONTINUE;
...
EXEC SQL WHENEVER SQLERROR GOTO sql_error:
...
sql_error:
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK RELEASE;
```

Related Topics

None

Index

Reader's Comment Form

Programmer's Guide to the Oracle Pro*C/C++ Precompiler
Part No. A32548–1

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this publication. Your input is an important part of the information used for revision.

- Did you find any errors?
- Is the information clearly presented?
- Do you need more information? If so, where?
- Are the examples correct? Do you need more examples?
- What features did you like most about this manual?

If you find any errors or have any other suggestions for improvement, please indicate the topic, chapter, and page number below:

Please send your comments to:

Languages Documentation Manager
Oracle Corporation
500 Oracle Parkway
Redwood City, CA 94065   U.S.A.
Fax 415–506–7200

If you would like a reply, please give your name, address, and telephone number below:

Thank you for helping us improve our documentation.