Pro*COBOL® Supplement to the

Oracle® Precompilers Guide

Release 1.8
Pro*COBOL® Supplement to the Oracle® Precompilers Guide, Release 1.8
Part No. A42524–1
Copyright © Oracle Corporation 1989, 1996
All rights reserved. Printed in the U.S.A.
Primary Author: Tom Portfolio
Contributing Author: Jack Godwin
Contributors: Stephen Arnold, Sanford Dreskin, Pierre Dufour, Steve Faris, Radhakrishna Hari, Nancy Ikeda, Ken Jacobs, Maura Joglekar, Phil Locke, Valerie Moore, Lee Osborne, Jacqui Pons, Tim Smith, Gael Turk, Scott Urman, Peter Vasterd

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.

This software/documentation contains proprietary information of Oracle Corporation; it is provided under a license agreement containing restrictions on use and disclosure and is also protected by copyright law. Reverse engineering of the software is prohibited.

If this software/documentation is delivered to a U.S. Government Agency of the Department of Defense, then it is delivered with Restricted Rights and the following legend is applicable:

Restricted Rights Legend Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of DFARS 252.227–7013, Rights in Technical Data and Computer Software (October 1988). Oracle Corporation, 500 Oracle Parkway, Redwood City, CA 94065.

If this software/documentation is delivered to a U.S. Government Agency not within the Department of Defense, then it is delivered with “Restricted Rights”, as defined in FAR 52.227–14, Rights in Data – General, including Alternate III (June 1987).

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.

Oracle, Pro*COBOL, SQL*Net, and SQL*Plus are registered trademarks of Oracle Corporation.
Oracle7, PL/SQL, Pro*C, Pro*C/C++, and Trusted Oracle7 are trademarks of Oracle Corporation.
VMS is a registered trademark of Digital Equipment Corporation. CMS is a registered trademark of International Business Machines Corporation. All other products or company names are used for identification purposes only, and may be trademarks of their respective owners.
Preface

This companion book to the *Programmer’s Guide to the Oracle Precompilers* shows you how to write COBOL programs that use the powerful database language SQL to access and manipulate Oracle data. It provides easy-to-follow examples, instructions, and programming tips, as well as several full-length programs to better your understanding and demonstrate the usefulness of embedded SQL.
What This Manual Has to Offer

This manual shows you how the Oracle Pro*COBOL 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 manual is its emphasis on getting the most out of Pro*COBOL and embedded SQL. To help you master these tools, this manual, accompanied by the Programmer’s Guide to the Oracle Precompilers, shows you all the “tricks of the trade” including ways to improve program performance.

Note: You will not find installation instructions or system–specific information in this manual. For that kind of information, refer to your system–specific Oracle documentation.

Who Should Read This Manual?

Anyone developing new COBOL applications or converting existing COBOL applications to run in the Oracle environment will benefit from reading this manual. Written especially for programmers, it will also be of value to systems analysts, project managers, and others interested in embedded SQL applications.

To use this manual effectively, you need a working knowledge of the following subjects:

• applications programming in COBOL
• the concepts, terminology, and methods discussed in the Programmer’s Guide to the Oracle Precompilers
• the SQL database language
• Oracle database concepts and terminology
What’s New in This Edition?

Release 1.8 of the Pro*COBOL Precompiler introduces a new command-line option, UNSAFE_NULL. With UNSAFE_NULL=YES, you can disable ORA-01405 messages when precompiling applications that fetch data into host variables that do not have associated indicator variables.

For more information, see Appendix A of the Programmer’s Guide to the Oracle Precompilers.

How This Guide Is Organized

This manual contains four chapters and an appendix. A brief summary of what you will find in each chapter and appendix follows:

Chapter 1: Writing a Pro*COBOL Program
This chapter provides the basic information you need to write a Pro*COBOL program. You learn programming guidelines, coding conventions, language-specific features and restrictions, how to equivalence datatypes, and how to connect to Oracle.

Chapter 2: Error Handling and Diagnostics
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 3: Sample Programs
This chapter provides several embedded SQL programs to guide you in writing your own. These well-commented programs illustrate the key concepts and features of Pro*COBOL programming.

Chapter 4: Implementing Dynamic SQL Method 4
This chapter shows you how to implement dynamic SQL Method 4, an advanced programming technique that lets you write highly flexible applications. Numerous examples, including a full-length sample program, are used to illustrate the method.

Appendix A: Operating System Dependencies
Some details of Pro*COBOL programming vary from one system to another. So, you are occasionally referred to other manuals for system-specific information. For convenience, this appendix collects all such external references.
Conventions Used in This Manual

Important terms being defined for the first time are italicized. In discussions, UPPERCASE is used for COBOL code, database objects, SQL keywords, and the names of variables, constants, and parameters.

**Notation**

The following notational conventions are used in some code examples:

- `< >` Angle brackets enclose the name of a syntactic element.
- `[ ]` Square brackets enclose optional items.
- `{ }` Braces enclose items only one of which is required.
- `|` A vertical bar separates options within brackets or braces.
- `...` An ellipsis shows that the preceding argument or parameter can be repeated, or that statements or clauses irrelevant to the discussion were left out.
- `#` This character is used in text to represent blank spaces when referring to the content of a database column.

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:

Languages Documentation Manager  
Oracle Corporation  
500 Oracle Parkway  
Redwood Shores, CA 94065  
Fax: (415) 506-7200
Chapter 1 Writing a Pro*COBOL Program ........................................ 1–1
Programming Guidelines ...................................................... 1–2
   Abbreviations ................................................................. 1–2
   COBOL Versions ............................................................. 1–2
   Coding Area ................................................................. 1–2
   Commas ........................................................................ 1–2
   Comments ....................................................................... 1–2
   Continuation Lines ......................................................... 1–3
   Delimiters ....................................................................... 1–3
   Embedded SQL Syntax ..................................................... 1–4
   Figurative Constants ....................................................... 1–4
   File Length ...................................................................... 1–4
   Host Variable Names ....................................................... 1–4
   Hyphenated Names ......................................................... 1–4
   Level Numbers .................................................................. 1–4
   MAXLITERAL Default ....................................................... 1–4
   Multi–Byte NLS Datatypes ............................................... 1–5
   Dynamic SQL ................................................................. 1–5
   Embedded DDL ............................................................... 1–5
   Nulls ............................................................................... 1–5
   Paragraph Names ............................................................ 1–5
   REDEFINES Clause ......................................................... 1–6
   Relational Operators ....................................................... 1–6
   Sentence Terminator ....................................................... 1–6
Chapter 2

Error Handling and Diagnostics ............................................. 2 – 1
   Error Handling Alternatives ........................................... 2 – 2
   SQLCODE and SQLSTATE .................................................. 2 – 2
   SQLCA ............................................................................ 2 – 3
   ORACA ............................................................................. 2 – 3
   Using Status Variables when MODE=[ANSI | ANSI14] .......... 2 – 3
   Some Historical Information .............................................. 2 – 4
   Declaring Status Variables ............................................... 2 – 5
   Status Variable Combinations ........................................... 2 – 6
   Using the SQL Communications Area ................................. 2 – 9
   What’s in the SQLCA? ....................................................... 2 – 10
   Declaring the SQLCA ....................................................... 2 – 11
   Key Components of Error Reporting ................................ 2 – 11
   Getting the Full Text of Error Messages ......................... 2 – 12
   Using the WHENEVER Statement ..................................... 2 – 13
   Using the Oracle Communications Area ........................... 2 – 17
   What’s in the ORACA? ....................................................... 2 – 17
   Declaring the ORACA ....................................................... 2 – 18
   Enabling the ORACA ......................................................... 2 – 18

Chapter 3

Sample Programs ................................................................. 3 – 1
   Sample Program 1: Simple Query ..................................... 3 – 2
   Sample Program 2: Cursor Operations ............................. 3 – 5
   Sample Program 3: Fetching in Batches ............................ 3 – 8
   Sample Program 4: Datatype Equivalencing ..................... 3 – 11
   Sample Program 5: Oracle Forms User Exit ....................... 3 – 16
   Sample Program 6: Dynamic SQL Method 1 ...................... 3 – 19
   Sample Program 7: Dynamic SQL Method 2 ...................... 3 – 22
   Sample Program 8: Dynamic SQL Method 3 ...................... 3 – 26
   Sample Program 9: Calling a Stored Procedure ................. 3 – 30
CHAPTER 1

Writing a Pro*COBOL Program

This chapter provides the basic information you need to write a Pro*COBOL program, including:

• programming guidelines
• coding conventions
• language-specific features and restrictions
• how to declare and reference host variables, indicator variables, host tables, and variable-length strings
• multi-byte NLS features
• how to equivalence datatypes
• how to connect to Oracle
Programming Guidelines

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

Abbreviations

You can use the standard COBOL abbreviations, such as PIC for PICTURE IS and COMP for USAGE IS COMPUTATIONAL.

COBOL Versions

The Pro*COBOL Precompiler supports the standard implementation of COBOL for your operating system (usually COBOL–85 or COBOL–74). Some platforms may support both COBOL implementations. For more information, see your Oracle system-specific documentation.

Coding Area

You must code EXEC SQL and EXEC ORACLE statements in columns 12 through 72 (columns 73 through 80 are ignored).

Note: The precompiler option FORMAT specifies the format of COBOL input lines. If you specify FORMAT=ANSI (default), columns 1 through 6 can contain an optional sequence number, column 7 indicates comments or continuation lines, paragraph names begin in columns 8 through 11, and statements begin in columns 12 through 72.

If you specify FORMAT=TERMINAL, columns 1 through 6 are dropped, making column 7 the leftmost column. In this manual, program examples reflect the FORMAT=TERMINAL setting. The sample programs are in ANSI format.

Commas

In SQL, you must use commas to separate list items, as the following example shows:

```sql
EXEC SQL SELECT ENAME, JOB, SAL
   INTO :EMP–NAME, :JOB–TITLE, :SALARY
   FROM EMP
   WHERE EMPNO = :EMP–NUMBER
END–EXEC.
```

In COBOL, you can use commas or blanks to separate list items. For example, the following two statements are equivalent:

```
ADD AMT1, AMT2, AMT3 TO TOTAL–AMT.
ADD AMT1 AMT2 AMT3 TO TOTAL–AMT.
```

Comments

You can place COBOL comment lines within SQL statements. COBOL comment lines start with an asterisk (*) in column 7. You can also place ANSI SQL–style comments (– …) within SQL statements at the end of a line (but not after the last line of the SQL statement), and you can place C–style comments (/ * … */ ) in SQL statements.
The following example shows all three styles of comments:

```cobol
EXEC SQL SELECT ENAME, SAL
* assign column values to output host variables
  INTO :EMP-NAME, :SALARY -- output host variables
/* column values assigned to output host variables */
FROM EMP
WHERE DEPTNO = :DEPT-NUMBER
END-EXEC. -- illegal comment
```

However, you cannot nest comments or place them on the last line of a SQL statement after the terminator END-EXEC.

### Continuation Lines

You can continue SQL statements from one line to the next, according to the rules of COBOL, as this example shows:

```cobol
EXEC SQL SELECT ENAME, SAL INTO :EMP-NAME, :SALARY FROM EMP
WHERE DEPTNO = :DEPT-NUMBER
END-EXEC.
```

No continuation indicator is needed.

To continue a string literal from one line to the next, code the literal through column 72. On the next line, code a hyphen (–) in column 7, a quote in column 12 or beyond, and then the rest of the literal. An example follows:

```cobol
WORKING-STORAGE SECTION.
...
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
...
01 UPDATE-STATEMENT PIC X(80) VALUE "UPDATE EMP SET BON
  US = 500 WHERE DEPTNO = 20".
EXEC SQL END DECLARE SECTION END-EXEC.
```

### Delimiters

The LITDELIM option specifies the delimiter for COBOL string constants and literals. If you specify LITDELIM=APOST, the precompiler uses apostrophes when generating COBOL code. If you specify LITDELIM=QUOTE (default), quotation marks are used, as in

```cobol
CALL "SQLROL" USING SQL-TMP0.
```

In SQL statements, you must use quotation marks to delimit identifiers containing special or lowercase characters, as in

```cobol
EXEC SQL CREATE TABLE "Emp2" END-EXEC.
```

However, you must use apostrophes to delimit string constants, as in

```cobol
EXEC SQL SELECT ENAME FROM EMP WHERE JOB = 'CLERK' END-EXEC.
```

Regardless of which delimiter is used in the Pro*COBOL source file, the precompiler generates the delimiter specified by the LITDELIM value.
Embedded SQL Syntax  To use a SQL statement in your host program, precede the SQL statement with the EXEC SQL clause, and end the statement with the END–EXEC keyword. Embedded SQL syntax is described in the Oracle7 Server SQL Reference. The precompiler translates all EXEC SQL statements into calls to the runtime library SQLLIB.

Figurative Constants  Figurative constants, such as HIGH–VALUE, ZERO, and SPACE, cannot be used in SQL statements. For example, the following is invalid:

```
EXEC SQL DELETE FROM EMP WHERE COMM = ZERO END–EXEC.
```

Instead, use the following:

```
EXEC SQL DELETE FROM EMP WHERE COMM = 0 END–EXEC.
```

File Length  The Pro*COBOL Precompiler cannot process arbitrarily long source files. Some of the variables used internally limit the size of the generated file. There is no absolute limit to the number of lines allowed, but 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.

Host Variable Names  Host–variable names must consist only of letters, digits, and hyphens, and must begin with a letter. They can be any length, but only the first 31 characters are significant. Your compiler might require a shorter length, so check your COBOL compiler user’s guide.

Hyphenated Names  You can use hyphenated host–variable names in static SQL statements but not in dynamic SQL. For example, the following usage is invalid:

```
MOVE "DELETE FROM EMP WHERE EMPNO = :EMP–NUMBER" TO SQLSTMT.
EXEC SQL PREPARE STMT1 FROM SQLSTMT END–EXEC.
```

Level Numbers  When declaring host variables in the Declare Section, you can use level numbers 01, 02 through 49, and 77.

MAXLITERAL Default  With the MAXLITERAL option, you can specify the maximum length of string literals generated by the precompiler, so that compiler limits are not exceeded. For Pro*COBOL, the default value is 256, but you might have to specify a lower value.
For example, if your COBOL compiler cannot handle string literals longer than 132 characters, specify MAXLITERAL=132. Check your COBOL compiler user’s guide.

Multi-Byte NLS Datatypes

ANSI standard National Language Support (NLS) datatypes are supported for handling multi-byte character data. For example, the PIC N clause declares variables that store fixed-length NLS strings. You can store variable-length, multi-byte NLS strings using COBOL group items consisting of a length field and a string field.

Dynamic SQL

Because dynamic SQL statements are not processed at precompile time, and since the Oracle7 Server, Release 7.3 does not itself process multi-byte NLS strings, you cannot embed multi-byte NLS strings in dynamic SQL statements.

Embedded DDL

Columns storing multi-byte NLS data cannot be used in embedded data definition language (DDL) statements. This restriction cannot be enforced when precompiling, so the use of these column types within embedded DDL statements results in an execution error rather than a precompile error.

Nulls

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

Paragraph Names

You can associate standard COBOL paragraph names with SQL statements, as shown in the following example:

```
LOAD-DATA.
  EXEC SQL
  INSERT INTO EMP (EMPNO, ENAME, DEPTNO)
  VALUES (:EMP-NUMBER, :EMP-NAME, :DEPT-NUMBER)
END-EXEC.
```

Also, you can reference paragraph names in a WHENEVER ... DO or WHENEVER ... GOTO statement, as the next example shows:

```
PROCEDURE DIVISION.
MAIN.
  EXEC SQL WHENEVER SQLERROR GOTO SQL-ERROR END-EXEC.
  ...
  SQL-ERROR.
  ...
```

You must begin all paragraph names in columns 8 through 11.
You can use the **REDEFINES** clause to redefine elementary items, but not group items. For example, the following declaration is valid:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01  REC-ID   PIC X(4).
  01  REC-NUM REDEFINES REC-ID PIC S9(4) COMP.
EXEC SQL END DECLARE SECTION END-EXEC.
```

However, the next declaration is **invalid**:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01  STOCK.
    05  DIVIDEND     PIC X(5).
    05  PRICE        PIC X(6).
  01  BOND REDEFINES STOCK.
    05  COUPON-RATE PIC X(4).
    05  PRICE        PIC X(7).
EXEC SQL END DECLARE SECTION END-EXEC.
```

**COBOL** relational operators differ from their SQL equivalents, as shown in Table 1–1. Furthermore, **COBOL** allows the use of words instead of symbols, whereas SQL does not.

<table>
<thead>
<tr>
<th>SQL Operators</th>
<th>COBOL Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>=, EQUAL TO</td>
</tr>
<tr>
<td>&lt;, &gt;, &lt;=, ^=</td>
<td>NOT=, NOT EQUAL TO</td>
</tr>
<tr>
<td>&gt;</td>
<td>&gt;, GREATER THAN</td>
</tr>
<tr>
<td>&lt;</td>
<td>&lt;, LESS THAN</td>
</tr>
<tr>
<td>&gt;=</td>
<td>&gt;=, GREATER THAN OR EQUAL TO</td>
</tr>
<tr>
<td>&lt;=</td>
<td>&lt;=, LESS THAN OR EQUAL TO</td>
</tr>
</tbody>
</table>

**Table 1–1 Relational Operators**

**Sentence Terminator**

A **COBOL** *sentence* includes one or more **COBOL** and/or SQL statements and ends with a period. In conditional sentences, only the last statement must end with a period, as the following example shows:

```
IF EMP–NUMBER = ZERO
  MOVE FALSE TO VALID–DATA
  PERFORM GET–EMP–NUM UNTIL VALID–DATA = TRUE
ELSE
  EXEC SQL DELETE FROM EMP
    WHERE EMPNO = :EMP–NUMBER
  END–EXEC
  ADD 1 TO DELETE–TOTAL.
END–IF.
```

With **COBOL–74**, however, if you use **WHENEVER ... GOTO** or **WHENEVER ... STOP** to handle errors for a **SQL** statement, the **SQL** statement must be terminated by a period or followed by an **ELSE**.
The DELETE statement below is repositioned to meet this requirement:

```sql
EXEC SQL WHENEVER SQLERROR GOTO SQL-ERROR END-EXEC.
IF EMP-NUMBER = ZERO
  MOVE FALSE TO VALID-DATA
  PERFORM GET-EMP-NUM UNTIL VALID-DATA = TRUE
ELSE
  ADD 1 TO DELETE-TOTAL
  EXEC SQL DELETE FROM EMP
  WHERE EMPNO = :EMP-NUMBER
END-EXEC.
```

Alternatively, you can place the SQL statement in a separate paragraph and PERFORM that paragraph.

### Required Declarations and SQL Statements

Passing data between Oracle and your application program requires host variables and event handling. This section shows you how to meet these requirements.

#### The Declare Section

You must declare all program variables to be used in SQL statements in the *Declare Section*, which begins with the statement

```sql
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
```

and ends with the statement

```sql
EXEC SQL END DECLARE SECTION END-EXEC.
```

Between these two statements only the following are allowed:

- host-variable and indicator-variable declarations
- EXEC SQL DECLARE statements
- EXEC SQL INCLUDE statements
- EXEC SQL VAR statements
- EXEC ORACLE statements
- COBOL comments

#### Using the INCLUDE Statement

The INCLUDE statement lets you copy files into your host program, as the following example shows:

* Copy in the SQL Communications Area (SQLCA)
  EXEC SQL INCLUDE SQLCA END-EXEC.
* Copy in the Oracle Communications Area (ORACA)
  EXEC SQL INCLUDE ORACA END-EXEC.
You can INCLUDE any file. When you precompile your Pro*COBOL program, each EXEC SQL INCLUDE statement is replaced by a copy of the file named in the statement.

Do not confuse the SQL command INCLUDE with the COBOL COPY command. If a file contains embedded SQL, you must INCLUDE it because only INCLUDED files are precompiled.

Filename Extensions
If your system uses file extensions but you do not specify one, the Pro*COBOL Precompiler assumes the default extension for source files (usually COB). The default extension is system–dependent. For more information, see your Oracle system–specific documentation.
Search Paths

If your system uses directories, you can set a search path for
INCLUDED files using the INCLUDE option, as follows:

```
INCLUDE=path
```

where `path` defaults to the current directory.

The precompiler first searches the current directory, then the directory
specified by the INCLUDE option, and finally the directory for
standard INCLUDE files. You need not specify a path for standard files
such as the SQLCA and ORACA. However, a path is required for
nonstandard files unless they are stored in the current directory.

You can also specify multiple paths on the command line, as follows:

```
... INCLUDE=<path1> INCLUDE=<path2> ...
```

When multiple paths are specified, the precompiler searches the current
directory first, then the `path1` directory, then the `path2` directory, and so
on. The directory containing standard INCLUDE files is searched last.
The path syntax is system specific. For more information, see your
Oracle system–specific documentation.

Caution

Remember, the precompiler searches for a file in the current directory
first even if you specify a search path. If the file you want to INCLUDE
is in another directory, make sure no file with the same name is in the
current directory or any other directory that precedes it in the search
path. Also, if your operating system is case sensitive, be sure to specify
the same upper/lowercase filename under which the file is stored.

Event and Error Handling

Pro*C OBOL provides forward and backward compatibility when
checking the outcome of executing SQL statements. However, there are
restrictions on using SQLCA, SQLCODE, and SQLSTATE depending
on the MODE and DBMS option settings. For more information, see
Chapter 2 of this manual and Chapter 8 of the Programmer’s Guide to the
Oracle Precompilers.

Host Variables

Host variables are the key to communication between your host
program and Oracle. Typically, a host 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.
Declaring Host Variables

Host variables are declared according to COBOL rules, using the COBOL datatypes that are supported by Oracle. COBOL datatypes must be compatible with the source/target database column. The supported COBOL datatypes are shown in Table 1 – 2.

<table>
<thead>
<tr>
<th>Variable Declaration</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIC X...X</td>
<td>fixed–length string of 1–byte characters (1)</td>
</tr>
<tr>
<td>PIC X(n)</td>
<td>n–length string of 1–byte characters</td>
</tr>
<tr>
<td>PIC X...X VARYING</td>
<td>variable–length string of 1–byte characters (1,2)</td>
</tr>
<tr>
<td>PIC X(n) VARYING</td>
<td>variable–length (n max.) string of 1–byte characters (2)</td>
</tr>
<tr>
<td>PIC N...N</td>
<td>fixed–length string of 2–byte NLS characters (1,3)</td>
</tr>
<tr>
<td>PIC N(n)</td>
<td>n–length string of 2–byte NLS characters (3)</td>
</tr>
<tr>
<td>PIC S9...9 BINARY</td>
<td>integer (4,5,7)</td>
</tr>
<tr>
<td>PIC S9(n) BINARY</td>
<td></td>
</tr>
<tr>
<td>PIC S9...9 COMP</td>
<td>byte–swapped integer (4,5,6,7)</td>
</tr>
<tr>
<td>PIC S9(n) COMP</td>
<td></td>
</tr>
<tr>
<td>COMP–1</td>
<td>floating–point number (4,5)</td>
</tr>
<tr>
<td>COMP–2</td>
<td></td>
</tr>
<tr>
<td>PIC S9...9V9...9 COMP–3</td>
<td>packed decimal (4,5)</td>
</tr>
<tr>
<td>PIC S9(n)V9(n) COMP–3</td>
<td>integer (if precision, which is optional, is omitted)</td>
</tr>
<tr>
<td>PIC S9...9V9...9 DISPLAY SIGN LEADING SEPARATE</td>
<td>display</td>
</tr>
<tr>
<td>PIC S9(n)V9(n) DISPLAY SIGN LEADING SEPARATE</td>
<td></td>
</tr>
<tr>
<td>SQL–CURSOR</td>
<td>cursor variable</td>
</tr>
</tbody>
</table>

Table 1 – 2 Host Variable Declarations

Notes:

1. X...X and 9...9 stand for a given number (n) of Xs or 9s. For variable–length strings, n is the maximum length.

2. The keyword VARYING assigns the VARCHAR external datatype to a character string. For more information, see “Declaring VARCHAR Variables” on page 1 – 21.

3. Before using the PIC N datatype in your Pro*COBOL source files, verify that it is supported by your COBOL compiler.

4. Only signed numbers (PIC S...) are allowed. For floating–point numbers, however, PIC strings are not accepted.

5. You must specify the appropriate COMPUTATIONAL system directive — COMP, COMP–1, COMP–2, COMP–3, or COMP–5 — for numeric variables.

6. With COMP or COMP–5, the number cannot have a fractional part; scaled binary numbers are not supported.

7. The maximum value of n ranges from 9 to 18, depending upon your system.

8. One–dimensional tables of COBOL types are also supported.
Table 1 – 3 shows the compatible Oracle internal datatypes.

<table>
<thead>
<tr>
<th>Internal Type</th>
<th>COBOL Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR(x)</td>
<td>PIC [X...X]PIC N...N</td>
<td>character string</td>
</tr>
<tr>
<td></td>
<td>PIC [X(n)]PIC N(n)</td>
<td>n-character string</td>
</tr>
<tr>
<td></td>
<td>PIC [X(n)]X(n) VARYING</td>
<td>variable-length string</td>
</tr>
<tr>
<td></td>
<td>PIC S9..9 COMP</td>
<td>integer</td>
</tr>
<tr>
<td></td>
<td>PIC S9(n) COMP</td>
<td>integer</td>
</tr>
<tr>
<td></td>
<td>PIC S9..9 BINARY</td>
<td>integer</td>
</tr>
<tr>
<td></td>
<td>PIC S9(n) BINARY</td>
<td>integer</td>
</tr>
<tr>
<td></td>
<td>PIC S9..9 COMP–5</td>
<td>integer</td>
</tr>
<tr>
<td></td>
<td>PIC S9(n) COMP–5</td>
<td>integer</td>
</tr>
<tr>
<td></td>
<td>COMP–1</td>
<td>floating point number</td>
</tr>
<tr>
<td></td>
<td>COMP–2</td>
<td>floating point number</td>
</tr>
<tr>
<td></td>
<td>PIC S9..9V9..9 COMP–3</td>
<td>packed decimal or integer</td>
</tr>
<tr>
<td></td>
<td>PIC S9(n)V9(n) COMP–3</td>
<td>packed decimal or integer</td>
</tr>
<tr>
<td>NUMBER</td>
<td>PIC S9..9 COMP</td>
<td>integer</td>
</tr>
<tr>
<td></td>
<td>PIC S9(n) COMP</td>
<td>integer</td>
</tr>
<tr>
<td></td>
<td>PIC S9..9 BINARY</td>
<td>integer</td>
</tr>
<tr>
<td></td>
<td>PIC S9(n) BINARY</td>
<td>integer</td>
</tr>
<tr>
<td></td>
<td>PIC S9..9 COMP–5</td>
<td>integer</td>
</tr>
<tr>
<td></td>
<td>PIC S9(n) COMP–5</td>
<td>integer</td>
</tr>
<tr>
<td></td>
<td>COMP–1</td>
<td>floating point number</td>
</tr>
<tr>
<td></td>
<td>COMP–2</td>
<td>floating point number</td>
</tr>
<tr>
<td></td>
<td>PIC S9..9V9..9 COMP–3</td>
<td>packed decimal</td>
</tr>
<tr>
<td></td>
<td>PIC S9(n)V9(n) COMP–3</td>
<td>packed decimal</td>
</tr>
<tr>
<td></td>
<td>PIC [X...X]PIC N...N</td>
<td>character string</td>
</tr>
<tr>
<td></td>
<td>PIC [X(n)]PIC N(n)</td>
<td>n-character string</td>
</tr>
<tr>
<td></td>
<td>PIC X...X VARYING</td>
<td>variable-length string</td>
</tr>
<tr>
<td></td>
<td>PICX(n) VARYING</td>
<td>n-byte variable-length string</td>
</tr>
<tr>
<td>DATE</td>
<td>PIC X(n)</td>
<td>n-byte character string</td>
</tr>
<tr>
<td></td>
<td>PIC X...X VARYING</td>
<td>n-byte variable-length string</td>
</tr>
<tr>
<td>LONG</td>
<td>PIC X(n)</td>
<td>n-byte character string</td>
</tr>
<tr>
<td></td>
<td>PIC X...X VARYING</td>
<td>n-byte variable-length string</td>
</tr>
<tr>
<td>RAW</td>
<td>PIC X(n)</td>
<td>n-byte character string</td>
</tr>
<tr>
<td></td>
<td>PIC X...X VARYING</td>
<td>n-byte variable-length string</td>
</tr>
<tr>
<td>ROWID</td>
<td>PIC S9..9V9..9 DISPLAY</td>
<td>display</td>
</tr>
<tr>
<td>MSLABEL</td>
<td>PIC S9(n)V9(n) DISPLAY</td>
<td>display</td>
</tr>
<tr>
<td>DISPLAY</td>
<td>SQL–CURSOR</td>
<td>cursor variable</td>
</tr>
</tbody>
</table>

Table 1 – 3 Compatible Oracle Internal Datatypes
Notes:

1. \( x \) ranges from 1 to 255, and 1 is the default. \( y \) ranges from 1 to 2000.
2. \( p \) ranges from 2 to 38. \( s \) ranges from –84 to 127.
3. Strings can be converted to NUMBERs only if they consist of convertible characters — 0 to 9, period (.), +, –, E, e. The NLS settings for your system might change the decimal point from a period (.) to a comma (,).
4. When converted to a string type, the default size of a DATE depends on the NLS settings in effect on your system. When converted to a binary value, the length is 7 bytes.
5. When converted to a string type, a ROWID requires from 18 to 256 bytes.
6. Trusted Oracle7 only.

Example Declarations

In the following example, you declare several host variables for use later in your Pro*COBOL program:

```cobol
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01  STR1  PIC X(3).
  01  STR2  PIC X(3) VARYING.
  01  NUM1  PIC S9(5) COMP.
  01  NUM2  COMP–1.
  01  NUM3  COMP–2.
EXEC SQL END DECLARE SECTION END-EXEC.
```

You can also declare one-dimensional tables of simple COBOL types, as the next example shows:

```cobol
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01  XMP–TABLES.
    05  TAB1  PIC XXX OCCURS 3 TIMES.
    05  TAB2  PIC XXX VARYING OCCURS 3 TIMES.
    05  TAB3  PIC S999 COMP–3 OCCURS 3 TIMES.
EXEC SQL END DECLARE SECTION END-EXEC.
```

Initialization

You can initialize host variables in the Declare Section using the VALUE clause, as shown in the following example:

```cobol
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01  USERNAME    PIC X(10) VALUE "SCOTT".
  01  MAX–SALARY  PIC S9(4) COMP VALUE 5000.
EXEC SQL END DECLARE SECTION END-EXEC.
```

If a string value assigned to a character variable is shorter than the declared length of the variable, the string is blank–padded on the right. If the string value assigned to a character variable is longer than the declared length, the string is truncated.
Restrictions

You cannot declare unsigned numeric character variables (PIC 9) or alphabetic character (PIC A) variables in the Declare Section. For example, the following declarations are invalid:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 ID-NUMBER   PIC 9(4).
01 FIRST-NAME PIC A(10).
EXEC SQL END DECLARE SECTION END-EXEC.
```

Nor can you define edited data items in the Declare Section. Therefore, the following declarations are invalid:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 AMOUNT-OF-CHECK PIC ****9V99.
EXEC SQL END DECLARE SECTION END-EXEC.
```

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 COBOL statements, as this example shows:

```
WORKING-STORAGE SECTION.
...
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01 EMP-NUMBER PIC S9(4) COMP VALUE ZERO.
  01 EMP-NAME PIC X(10) VALUE SPACE.
  01 SALARY     PIC S9(5)V99 COMP-3.
EXEC SQL END DECLARE SECTION END-EXEC.
...

PROCEDURE DIVISION.
...
DISPLAY "Employee number? " WITH NO ADVANCING.
ACCEPT EMP-NUMBER.
EXEC SQL SELECT ENAME, SAL
  INTO :EMP-NAME, :SALARY FROM EMP
  WHERE EMPNO = :EMP-NUMBER
END-EXEC.
COMPUTE BONUS = SALARY / 10.
...
```

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

```
WORKING-STORAGE SECTION.
...
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01 EMPNO   PIC S9(4) COMP VALUE ZERO.
  01 ENAME   PIC X(10) VALUE SPACE.
  01 COMM    PIC S9(5)V99 COMP-3.
EXEC SQL END DECLARE SECTION END-EXEC.
...
PROCEDURE DIVISION.
...
EXEC SQL SELECT ENAME, COMM
    INTO :ENAME, :COMM FROM EMP
    WHERE EMPNO = :EMPNO
END-EXEC.

Elementary Item versus Group Item

You cannot reference group items in SQL statements (see “Referencing VARCHAR Variables” on page 1–22 for an exception). However, you can reference elementary items wherever host variables are allowed. For example, given the declaration

EXEC SQL BEGIN DECLARE SECTION END-EXEC.
...
01  DEPARTURE.
   05  HOUR    PIC X(2).
   05  MINUTE  PIC X(2).
EXEC SQL END DECLARE SECTION END-EXEC.

the following statement is invalid:

EXEC SQL SELECT DHOUR, DMINUTE
    INTO :DEPARTURE FROM SCHEDULE
    WHERE ...
END-EXEC.

However, the following statement is valid:

EXEC SQL SELECT DHOUR, DMINUTE
    INTO :HOUR, :MINUTE FROM SCHEDULE
    WHERE ...
END-EXEC.

Elementary names need not be unique because you can qualify them using the following syntax:

<group_item>.<elementary_item>

This naming convention is allowed only in SQL statements. It is similar to the IN (or OF) clause in COBOL, examples of which follow:

MOVE MINUTE IN DEPARTURE TO MINUTE-OUT.
DISPLAY HOUR OF DEPARTURE.
The COBOL IN (or OF) clause is not allowed in SQL statements.
Qualify elementary names to avoid ambiguity. For example, given the following declarations:

```cobol
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01 DEPARTURE.
    05 HOUR PIC X(2).
    05 MINUTE PIC X(2).
  01 ARRIVAL.
    05 HOUR PIC X(2).
    05 MINUTE PIC X(2).
EXEC SQL END DECLARE SECTION END-EXEC.
```

you must qualify HOUR and MINUTE, as in

```sql
EXEC SQL SELECT AHOUR, AMINUTE
              INTO :ARRIVAL.HOUR, :ARRIVAL.MINUTE
              FROM SCHEDULE
              WHERE ...;
```

This works for items nested deeper than two levels, provided you fully qualify them starting at the highest level — even if that would be unnecessary in COBOL. For example, given the declarations

```cobol
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01 TIMETABLE.
    05 DEPARTURE.
      10 HOUR PIC X(2).
      10 MINUTE PIC X(2).
    05 ARRIVAL.
      10 HOUR PIC X(2).
      10 MINUTE PIC X(2).
EXEC SQL END DECLARE SECTION END-EXEC.
```

you must fully qualify HOUR and MINUTE, as in

```sql
EXEC SQL SELECT AHOUR, AMINUTE
              INTO :TIMETABLE.ARRIVAL.HOUR, :TIMETABLE.ARRIVAL.MINUTE
              FROM SCHEDULE
              WHERE ...;
END-EXEC.
```

even though in COBOL, the following would suffice:

```cobol
DISPLAY HOUR OF ARRIVAL, ":\n", MINUTE OF ARRIVAL.
```

Restrictions

A host variable 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 of the *Programmer’s Guide to the Oracle Precompilers* for a list of Oracle reserved words and keywords.
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.

Declaring Indicator Variables

An indicator variable must be explicitly declared in the Declare Section as a 2-byte integer (PIC S9(4) COMP) and must not be an Oracle reserved word. In the following example, you declare an indicator variable named COMM–IND (the name is arbitrary):

```cobol
01  EMP–NAME    PIC X(10) VALUE SPACE.
01  COMMISSION  PIC S9(5)V99 COMP–3.
01  COMM–IND    PIC S9(4) COMP.
```

You can define an indicator variable anywhere in the Declare Section. It need not follow its associated host variable.

Referencing Indicator Variables

In SQL statements, an indicator variable must be prefixed with a colon and appended to its associated host variable. In COBOL statements, an indicator variable must not be prefixed with a colon or appended to its associated host variable. An example follows:

```cobol
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 EMP–NAME PIC X(10) VALUE SPACE.
01 COMMISSION PIC S9(5)V99 COMP–3.
01 COMM–IND PIC S9(4) COMP.
EXEC SQL END DECLARE SECTION END-EXEC.
```

An indicator variable must be prefixed with a colon and appended to its associated host variable. In COBOL statements, an indicator variable must not be prefixed with a colon or appended to its associated host variable. An example follows:

```sql
EXEC SQL SELECT SAL, COMM
    INTO :SALARY, :COMMISSION:COMM–IND FROM EMP
    WHERE EMPNO = :EMP–NUMBER
END-EXEC.
```

IF COMM–IND = –1
   COMPUTE PAY = SALARY
ELSE
   COMPUTE PAY = SALARY + COMMISSION.

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

```sql
:<host_variable> INDICATOR :<indicator_variable>
```

and is equivalent to

```sql
:<host_variable>::<indicator_variable>
```

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

Indicator variables cannot be used in the WHERE clause to search for nulls. For example, the following DELETE statement triggers an Oracle error at run time:

*    Set indicator variable.
COMM–IND = -1
EXEC SQL
    DELETE FROM EMP WHERE COMM = :COMMISSION:COMM–IND
END–EXEC.

The correct syntax follows:

EXEC SQL
    DELETE FROM EMP WHERE COMM IS NULL
END–EXEC.

Oracle Restrictions

When DBMS=V6, Oracle does not issue an error message 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 specified, you can disable the ORA–01405 message by also specifying UNSAFE_NULL=YES on the command line. For more information, see the Programmer’s Guide to the Oracle Precompilers.

ANSI Requirements

When MODE=ORACLE, if you SELECT or FETCH a truncated column value into a host variable that is not associated with an indicator variable, Oracle issues the following error message:

ORA–01406: fetched column value was truncated

However, when MODE={ANSI|ANSI14|ANSI13}, no error is generated. Values for indicator variables are discussed in Chapter 3 of the Programmer’s Guide to the Oracle Precompilers.

Indicator Variables for Multi–Byte NLS Variables

Indicator variables for multi–byte NLS character variables can be used as with any other host variable. However, a positive value (the result of a SELECT or FETCH was truncated) represents the string length in 2–byte characters instead of 1–byte characters.
Host Tables

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

Declaring Host Tables

You declare and dimension host tables in the Declare Section. In the following example, three host tables are declared, each dimensioned with 50 elements:

```sql
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01 EMP-TABLES.
    05 EMP-NUMBER OCCURS 50 TIMES PIC S9(4) COMP.
    05 EMP-NAME OCCURS 50 TIMES PIC X(10).
    05 SALARY OCCURS 50 TIMES PIC S9(5)V99 COMP-3.
EXEC SQL END DECLARE SECTION END-EXEC.
```

You can use the INDEXED BY phrase in the OCCURS clause to specify an index, as the next example shows:

```sql
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01 EMP-TABLES.
    05 EMP-NUMBER PIC X(10) OCCURS 50 TIMES
        INDEXED BY EMP-INDX.
    ...
EXEC SQL END DECLARE SECTION END-EXEC.
```

The INDEXED BY phrase implicitly declares the index item EMP-INDX.

Restrictions

Multi-dimensional host tables are not allowed. Thus, the two-dimensional host table declared in the following example is invalid:

```sql
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01 NATION.
    05 STATE OCCURS 50 TIMES.
      10 STATE-NAME PIC X(25).
      10 COUNTY OCCURS 25 TIMES.
        15 COUNTY-NAME PIC X(25).
EXEC SQL END DECLARE SECTION END-EXEC.
```

Variable-length host tables are not allowed either. For example, the following declaration of EMP-REC is invalid:

```sql
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01 EMP-FILE.
    05 REC-COUNT PIC S9(3) COMP.
    05 EMP-REC OCCURS 0 TO 250 TIMES
        DEPENDING ON REC-COUNT.
EXEC SQL END DECLARE SECTION END-EXEC.
```
Referencing Host Tables

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

```
WORKING-STORAGE SECTION.
  EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01 EMP-TABLES.
   05 EMP-NUMBER PIC S9(4) COMP OCCURS 50 TIMES.
   05 EMP-NAME PIC X(10) OCCURS 50 TIMES.
   05 DEPT-NUMBER PIC S9(4) COMP OCCURS 25 TIMES.
  EXEC SQL END DECLARE SECTION END-EXEC.
...
PROCEDURE DIVISION.
...
* Populate host tables here.
...
EXEC SQL INSERT INTO EMP (EMPNO, ENAME, DEPTNO)
   VALUES (:EMP-NUMBER, :EMP-NAME, :DEPT-NUMBER)
END-EXEC.
```

Host tables must not be subscripted in SQL statements. For example, the following INSERT statement is invalid:

```
WORKING-STORAGE SECTION.
  EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01 EMP-TABLES.
   05 EMP-NUMBER PIC S9(4) COMP OCCURS 50 TIMES.
   05 EMP-NAME PIC X(10) OCCURS 50 TIMES.
   05 DEPT-NUMBER PIC S9(4) COMP OCCURS 50 TIMES.
  EXEC SQL END DECLARE SECTION END-EXEC.
...
PROCEDURE DIVISION.
...
PERFORM LOAD-EMP VARYING J FROM 1 BY 1 UNTIL J > 50.
...
LOAD-EMP.
  EXEC SQL INSERT INTO EMP (EMPNO, ENAME, DEPTNO)
   VALUES (:EMP-NUMBER(J), :EMP-NAME(J), :DEPT-NUMBER(J))
END-EXEC.
```

You need not process host tables in a PERFORM VARYING statement. Instead, use the unsubscripted table names in your SQL statement. Oracle treats a SQL statement containing host tables of dimension \( n \) like the same statement executed \( n \) times with \( n \) different scalar host variables. For more information, see Chapter 8 of the *Programmer’s Guide to the Oracle Precompilers*. 
Using Indicator Tables  You can use indicator tables to assign nulls to elements in input host tables and to detect nulls or truncated values in output host tables. The following example shows how to INSERT with indicator tables:

```
WORKING-STORAGE SECTION.
  EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01  EMP-TABLES.
    05  EMP-NUMBER   PIC S9(4) COMP OCCURS 50 TIMES.
    05  DEPT-NUMBER  PIC S9(4) COMP OCCURS 50 TIMES.
    05  COMMISSION   PIC S9(5)V99 COMP-3 OCCURS 50 TIMES.
    05  COMM-IND     PIC S9(4) COMP OCCURS 50 TIMES.
  EXEC SQL END DECLARE SECTION END-EXEC.
...
PROCEDURE DIVISION.
...
* Populate the host and indicator tables.
* Set indicator table to all zeros.
...
EXEC SQL INSERT INTO EMP (EMPNO, DEPTNO, COMM)
       VALUES (:EMP-NUMBER, :DEPT-NUMBER,
              :COMMISSION, :COMM-IND)
END-EXEC.
```

The dimension of the indicator table must be greater than, or equal to, the dimension of the host table.
VARCHAR Variables

COBOL string datatypes are fixed length. However, Pro*COBOL lets you declare a variable–length string pseudotype called VARCHAR.

Declaring VARCHAR Variables

You define a VARCHAR host variable by adding the keyword VARYING to its declaration, as shown in the following example:

```sql
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
...
01 ENAME PIC X(15) VARYING.
...
EXEC SQL END DECLARE SECTION END-EXEC.
```

The VARYING phrase in the preceding example is used in PERFORM and SEARCH statements to increment subscripts and indexes. Do not confuse this with the COBOL VARYING clause.

You can define a VARCHAR variable only in the Declare Section. Think of it as an extended COBOL datatype or predeclared group item. For example, the precompiler expands the VARCHAR declaration

```c
01 ENAME PIC X(15) VARYING.
```

into a group item with length and string fields, as follows:

```c
01 ENAME.
   05 ENAME-LEN PIC S9(4) COMP.
   05 ENAME-ARR PIC X(15).
```

The `length` field (suffixed with –LEN) holds the current length of the value stored in the `string` field (suffixed with –ARR). The maximum length in the VARCHAR host–variable declaration must be in the range of 1 to 65533 bytes.

**Note:** The keyword VARYING cannot be used when declaring multi–byte NLS character data.

The advantage of using VARCHAR variables is that you can explicitly set and reference the length field. With input host variables, Oracle reads the value of the length field and uses that many characters of the string field. With output host variables, Oracle sets the length value to the length of the character string stored in the string field.
The Pro*COBOL Precompiler implicitly recognizes two group–item formats as VARCHAR host variables with VARCHAR=YES specified on the command line. For variable–length single–byte character types, use the following structure (length expressed in single–byte characters):

```
01 DATA–NAME–1.
  49 DATA–NAME–2 PIC S9(4) COMP.
  49 DATA–NAME–3 PIC X(<length>).
```

For variable–length multi–byte NLS character types, use this structure (length expressed in double–byte characters):

```
01 DATA–NAME–1.
  49 DATA–NAME–2 PIC S9(4) COMP.
  49 DATA–NAME–3 PIC N(<length>).
```

The elementary items in these group–item structures must be declared as level 49 for the Pro*COBOL Precompiler to recognize them as VARCHAR host variables.

**Note:** Tables of multi–byte NLS VARCHAR variables are not supported.

For more information about the Pro*COBOL VARCHAR option, see Chapter 6 of the *Programmer’s Guide to the Oracle Precompilers*.

In SQL statements, you reference a VARCHAR variable using the group name prefixed with a colon, as the following example shows:

```
WORKING–STORAGE SECTION.
...
EXEC SQL BEGIN DECLARE SECTION END–EXEC.
...
  01 PART–NUMBER PIC X(5).
  01 PART–DESC PIC X(20) VARYING.
EXEC SQL END DECLARE SECTION END–EXEC.
...
PROCEDURE DIVISION.
...
EXEC SQL
   SELECT PDESC INTO :PART–DESC FROM PARTS
   WHERE PNUM = :PART–NUMBER
END–EXEC.
```

After the query executes, PART–DESC–LEN holds the actual length of the character string retrieved from the database and stored in PART–DESC–ARR.

Remember, except for VARCHAR variables, you cannot reference group items in SQL statements.
In COBOL statements, you can reference VARCHAR variables using the group name or the elementary items, as this example shows:

```
WORKING-STORAGE SECTION.

... 
EXEC SQL BEGIN DECLARE SECTION END-EXEC.

... 
01  EMP-TABLES.
   05  EMP-NAME OCCURS 50 TIMES PIC X(15) VARYING.
   ... 
EXEC SQL END DECLARE SECTION END-EXEC.
... 
PROCEDURE DIVISION.
...
PERFORM DISPLAY-NAME
   VARYING J FROM 1 BY 1 UNTIL J > NAME-COUNT.
... 
DISPLAY-NAME.
   DISPLAY EMP-NAME-ARR OF EMP-NAME(J).
```
Handling Character Data

This section explains how the Pro*COBOL Precompiler handles character host variables. There are two types of single-byte character host variables and one type of multi-byte NLS character host variables:

- PIC X(n) (or PIC X...X)
- PIC X(n) VARYING (or PIC X...X VARYING)
- PIC N(n) (or PIC N...N)

Attention: Before using multi-byte NLS datatypes, verify that the PIC N datatype is supported by your COBOL compiler.

Effects of the MODE Option

The MODE option determines how the Pro*COBOL Precompiler treats data in character strings. The MODE option allows your program to use ANSI fixed-length strings or to maintain compatibility with previous versions of the Oracle Server and the Pro*COBOL Precompiler.

With respect to character handling, MODE={ANSI14 | ANSI13} is equivalent to MODE=ORACLE. The MODE option affects character data on input (from host variables to Oracle) and on output (from Oracle to host variables).

Note: The MODE option does not affect how Pro*COBOL handles PIC X(n) VARYING variables.

Single-Byte Character Variables

Single-byte character variables are declared using the PIC X(n) and PIC X VARYING datatypes. These types of variables handle character data based on their roles as input or output variables.

On Input

When MODE=ORACLE, the program interface strips trailing blanks before sending the value to the database. If you insert into a fixed-length CHAR column, Oracle re-appends trailing blanks up to the length of the database column. However, if you insert into a variable-length VARCHAR2 column, Oracle never appends blanks.

When MODE=ANSI, trailing blanks are never stripped.

Make sure that the input value is not trailed by extraneous characters. For example, nulls are not stripped and are inserted into the database. Normally, this is not a problem because when a value is ACCEPTed or MOVEd into a PIC X(n) variable, COBOL appends blanks up to the length of the variable.
The following example illustrates the point:

```
WORKING-STORAGE SECTION.
...
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01 EMPLOYEES.
    05 EMP–NAME     PIC X(10).
    05 DEPT–NUMBER  PIC S9(4) VALUE 20 COMP.
    05 EMP–NUMBER   PIC S9(9) VALUE 9999 COMP.
    05 JOB–NAME     PIC X(8).
...
EXEC SQL END DECLARE SECTION END-EXEC.
...
PROCEDURE DIVISION.
...
DISPLAY "Employee name?" WITH NO ADVANCING.
ACCEPT EMP–NAME.
* Assume that the name MILLER was entered
* EMP–NAME contains "MILLER    " (4 trailing blanks)
  MOVE "SALES" TO JOB–NAME.
* JOB–NAME now contains "SALES   " (3 trailing blanks)
  EXEC SQL INSERT INTO EMP (EMPNO, ENAME, DEPTNO, JOB)
  END–EXEC.
...
```

If you precompile the last example with MODE=ORACLE and the target database columns are VARCHAR2, the program interface strips the trailing blanks on input and inserts just the 6-character string “MILLER” and the 5-character string “SALES” into the database. However, if the target database columns are CHAR, the strings are blank-padded to the width of the columns.

If you precompile the last example with MODE=ANSI and the JOB column is defined as CHAR(10), the value inserted into that column is “SALES####” (five trailing blanks). However, if the JOB column is defined as VARCHAR2(10), the value inserted is “SALES###” (three trailing blanks), because the host variable is declared as PIC X(8). This might not be what you want, so be careful.

On Output

The MODE option has no effect on output to character variables. When you use a PIC X(n) variable as an output host variable, Oracle blank-pads it. In our example, when your program fetches the string “MILLER” from the database, EMP–NAME contains the value “MILLER####” (with four trailing blanks). This character string can be used without change as input to another SQL statement.
Multi–byte NLS Character Variables

Specify MODE=ANSI when using fixed–length NLS host variables (declared with the PIC N clause) are assigned to external datatype 96 (ANSI fixed CHAR).

⚠️ Warning: Oracle7 SQL functions should not be used on columns or host variables that store multi–byte NLS data. Multi–byte NLS features are supported by the SQLLIB runtime library, but they are not supported by the Oracle7 Server.

On Input

Input host variables are stripped of any trailing double–byte spaces. If the multi–byte NLS input variable contains only spaces, one double–byte space is left in the buffer to act as a sentinel.

To declare a multi–byte NLS character type in Pro*COBOL, use the following PICTURE clause:

```
WORKING–STORAGE SECTION.
  EXEC SQL BEGIN DECLARE SECTION END–EXEC.
  ...
  01 <nls_variable> PIC N(<length>).
  ...
  EXEC SQL END DECLARE SECTION END–EXEC.
```

where `nls_variable` is a valid COBOL variable name and `length` is the maximum number of multi–byte NLS characters the string can hold.

The example in the “Single–Byte Character Variables” section (page 1 – 24) could be rewritten to accept double–byte NLS characters using the PIC N datatype shown in the following example:

```
EXEC SQL BEGIN DECLARE SECTION END–EXEC.
  01 EMPLOYEES.
    05 EMP–NAME      PIC N(10).
    05 DEPT–NUMBER   PIC S9(4) VALUE 20 COMP.
    05 EMP–NUMBER    PIC S9(9) VALUE 9999 COMP.
    05 JOB–NAME      PIC N(8).
  EXEC SQL END DECLARE SECTION END–EXEC.
```

The EMP–NAME variable now accepts up to ten double–byte characters, while JOB–NAME accepts eight double–byte characters.
On Output

When you use a multi-byte NLS character variable as an output host variable, Oracle blank-pads it with double-byte spaces. This character string can be used without change as input to another SQL statement, providing it is mapped to another multi-byte NLS variable.

In the example for single-byte character data on page 1–25, the program fetches the string “MILLER” from the database. If using multi-byte NLS characters, EMP–NAME contains the value “MILLER” with each character allocated two bytes. The string is blank-padded with four double-byte spaces.

Restrictions

**Tables Disallowed.** Host variables declared using the PIC N datatype must not be tables.

**No Odd-Byte Widths.** Oracle CHAR columns should not be used to store multi-byte NLS characters. A runtime error is generated if data with an odd number of bytes is FETCHed from a single-byte column into a multi-byte NLS (PIC N) host variable.

**No Host Variable Equivalencing.** Multi-byte NLS character variables cannot be equivalenced using an EXEC SQL VAR statement.

**No Dynamic SQL.** Dynamic SQL is not available for NLS multi-byte character string host variables in Pro*COBOL.
**VARCHAR Variables**

VARCHAR variables handle character data based on their roles as input or output variables.

**On Input**

When you use a VARCHAR variable as an input host variable, your program must assign values to the length and string fields of the expanded VARCHAR declaration, as shown in the following example:

```
IF ENAME-IND = -1
  MOVE "NOT AVAILABLE" TO ENAME-ARR
  MOVE 13 TO ENAME-LEN.
```

You need not blank-pad the string variable. In SQL operations, Oracle uses exactly the number of characters given by the length field, counting any spaces.

Host input variables for multi-byte NLS data are not stripped of trailing double-byte spaces. The length component is assumed to be the length of the data in characters, not bytes.

**On Output**

When you use a VARCHAR variable as an output host variable, Oracle sets the length field. An example follows:

```
WORKING-STORAGE SECTION.
...
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01 EMPNO PIC S9(4) COMP.
  01 ENAME PIC X(15) VARYING.
...
EXEC SQL END DECLARE SECTION END-EXEC.
...
PROCEDURE DIVISION.
...
EXEC SQL
  SELECT ENAME INTO :ENAME FROM EMP
  WHERE EMPNO = :EMPNO
END-EXEC.
IF ENAME-LEN = 0
  MOVE FALSE TO VALID-DATA.
```

An advantage of VARCHAR variables over fixed-length strings is that the length of the value returned by Oracle is available right away. With fixed-length strings, to get the length of the value, your program must count the number of characters.

Host output variables for multi-byte NLS data are not padded at all. The length of the buffer is set to the length in characters, not bytes.
The Oracle Datatypes

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. An external datatype specifies how data is stored in a host variable. For descriptions of the Oracle datatypes, see Chapter 3 of the *Programmer’s Guide to the Oracle Precompilers*.

### Internal Datatypes

For values stored in database columns, Oracle uses the following internal datatypes:

<table>
<thead>
<tr>
<th>Name</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR</td>
<td>96</td>
<td>≤ 255-byte, fixed–length string</td>
</tr>
<tr>
<td>DATE</td>
<td>12</td>
<td>7-byte, fixed–length date/time value</td>
</tr>
<tr>
<td>LONG</td>
<td>8</td>
<td>≤ 2147483647-byte, variable–length string</td>
</tr>
<tr>
<td>LONG RAW</td>
<td>24</td>
<td>≤ 2147483647-byte, variable–length binary data</td>
</tr>
<tr>
<td>MLSLABEL</td>
<td>105</td>
<td>≤ 5-byte, variable–length binary label</td>
</tr>
<tr>
<td>NUMBER</td>
<td>2</td>
<td>fixed or floating point number</td>
</tr>
<tr>
<td>RAW</td>
<td>23</td>
<td>≤ 255-byte, variable–length binary data</td>
</tr>
<tr>
<td>ROWID</td>
<td>11</td>
<td>fixed–length binary value</td>
</tr>
<tr>
<td>VARCHAR2</td>
<td>1</td>
<td>≤ 2000-byte, variable–length string</td>
</tr>
</tbody>
</table>

*Table 1 – 4 Internal Datatypes*

These internal datatypes can be quite different from COBOL datatypes. For example, COBOL has no equivalent to the NUMBER datatype, which was specially designed for portability and high precision.
External Datatypes

As the table below shows, the external datatypes include all the internal datatypes plus several datatypes found in other supported host languages. For example, the STRING external datatype refers to a C null–terminated string. You use the datatype names in datatype equivalencing, and you use the datatype codes in dynamic SQL Method 4.

<table>
<thead>
<tr>
<th>Name</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR</td>
<td>1</td>
<td>≤ 65535–byte, variable–length character string (1)</td>
</tr>
<tr>
<td></td>
<td>96</td>
<td>≤ 65535–byte, fixed–length character string (1)</td>
</tr>
<tr>
<td>CHARF</td>
<td>96</td>
<td>≤ 65535–byte, fixed–length character string</td>
</tr>
<tr>
<td>CHARZ</td>
<td>97</td>
<td>≤ 65535–byte, fixed–length, null–terminated string (2)</td>
</tr>
<tr>
<td>DATE</td>
<td>12</td>
<td>7–byte, fixed–length date/time value</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>7</td>
<td>COBOL packed decimal</td>
</tr>
<tr>
<td>DISPLAY</td>
<td>91</td>
<td>COBOL numeric character string</td>
</tr>
<tr>
<td>FLOAT</td>
<td>4</td>
<td>4–byte or 8–byte floating–point number</td>
</tr>
<tr>
<td>INTEGER</td>
<td>3</td>
<td>2–byte or 4–byte signed integer</td>
</tr>
<tr>
<td>LONG</td>
<td>8</td>
<td>≤ 2147483647–byte, fixed–length string</td>
</tr>
<tr>
<td>LONG RAW</td>
<td>24</td>
<td>≤ 2147483647–byte, fixed–length binary data</td>
</tr>
<tr>
<td>LONG VARCHAR</td>
<td>94</td>
<td>≤ 2147483643–byte, variable–length string</td>
</tr>
<tr>
<td>LONG VARCHAR</td>
<td>95</td>
<td>≤ 2147483643–byte, variable–length binary data</td>
</tr>
<tr>
<td>MLSLABEL</td>
<td>106</td>
<td>2.5–byte, variable–length binary data</td>
</tr>
<tr>
<td>NUMBER</td>
<td>2</td>
<td>integer or floating–point number</td>
</tr>
<tr>
<td>RAW</td>
<td>23</td>
<td>≤ 65535–byte, fixed–length binary data (2)</td>
</tr>
<tr>
<td>ROWID</td>
<td>11</td>
<td>(typically) 13–byte, fixed–length binary value</td>
</tr>
<tr>
<td>STRING</td>
<td>5</td>
<td>≤ 65535–byte, null–terminated character string (2)</td>
</tr>
<tr>
<td>UNSIGNED</td>
<td>68</td>
<td>2–byte or 4–byte unsigned integer</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>9</td>
<td>≤ 65533–byte, variable–length character string</td>
</tr>
<tr>
<td>VARCHAR2</td>
<td>1</td>
<td>≤ 65535–byte, variable–length character string (2)</td>
</tr>
<tr>
<td>VARNUM</td>
<td>6</td>
<td>variable–length binary number</td>
</tr>
<tr>
<td>VARRAW</td>
<td>15</td>
<td>≤ 6553–byte, variable–length binary data</td>
</tr>
</tbody>
</table>

Table 1 – 5 External Datatypes

Notes:

1. CHAR is datatype 1 when MODE={ORACLE | ANSI13 | ANSI14} and datatype 96 when MODE=ANSI.
2. Maximum size is 32767 (32K) on some platforms.
Datatype Conversion

At precompile time, an external datatype is assigned to each host variable in the Declare Section. For example, the precompiler assigns the INTEGER external datatype to host variables of type PIC S9(n) COMP. 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 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 PIC S9(4) COMP value. You cannot, however, convert a CHAR value of “65543” (number too large) or “10F” (number not decimal) to a PIC S9(4) COMP value. Likewise, you cannot convert a PIC X(n) value that contains alphabetic characters to a NUMBER value.

For more information about datatype conversion, see Chapter 3 of the *Programmer’s Guide to the Oracle Precompilers*. 
Datatype Equivalencing

Datatype equivalencing lets you control the way Oracle interprets input data and the way Oracle formats output data. You can equivalence supported COBOL datatypes to Oracle external datatypes on a variable–by–variable basis.

**Attention:** Multi–byte NLS character variables cannot be equivalenced using the EXEC SQL VAR statement.

Host Variable Equivalencing

By default, the Pro*COBOL Precompiler assigns a specific external datatype to every host variable. The default assignments are shown in Table 1 – 6. For more information about datatype equivalencing, see Chapter 3 in the *Programmer’s Guide to the Oracle Precompilers*.

<table>
<thead>
<tr>
<th>Host Type</th>
<th>External Type</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIC X...X</td>
<td>VARCHAR2</td>
<td></td>
</tr>
<tr>
<td>PIC X(n)</td>
<td>CHARF</td>
<td>96   (when MODE=ANSI)</td>
</tr>
<tr>
<td>PIC N...N</td>
<td>CHARF</td>
<td>96   (when MODE=ANSI)</td>
</tr>
<tr>
<td>PIC X...X VARYING</td>
<td>VARCHAR</td>
<td>9</td>
</tr>
<tr>
<td>PIC X(n) VARYING</td>
<td>CHARF</td>
<td>96 (when MODE=ANSI)</td>
</tr>
<tr>
<td>PIC S9...9 COMP</td>
<td>INTEGER</td>
<td>3</td>
</tr>
<tr>
<td>PIC S9(n) COMP</td>
<td>FLOAT</td>
<td>4</td>
</tr>
<tr>
<td>PIC S9...9 COMP–5</td>
<td>DECIMAL</td>
<td>7</td>
</tr>
<tr>
<td>PIC S9...9V9...9 DISPLAY SIGN LEADING SEPARATE</td>
<td>DISPLAY</td>
<td>91</td>
</tr>
</tbody>
</table>

Table 1 – 6 Host Variable Equivalencing

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

```sql
EXEC SQL
    VAR <host_variable>
    IS <ext_type_name> [{(<length> | <precision>,<scale>),}] 
END-EXEC.
```

where *host_variable* is an input or output host variable (or host table) declared earlier in the Declare Section, *ext_type_name* is the name of a valid external datatype, and *length* is an integer literal specifying a valid length in bytes.
When `ext_type_name` is FLOAT, use `length`; when `ext_type_name` is DECIMAL, you must specify `precision` and `scale` instead of `length`.

Host variable equivalencing is useful in several ways. For example, you can use it when you want Oracle to store but not interpret data. Suppose you want to store a host table of 4-byte integers in a RAW database column. Simply equivalence the host table to the RAW external datatype, as follows:

```
WORKING–STORAGE SECTION.
  EXEC SQL BEGIN DECLARE SECTION END–EXEC.
  01  EMP–TABLES.
      05  EMP–NUMBER  PIC S9(9) COMP OCCURS 50 TIMES.
      ...
  *    Reset default datatype (INTEGER) to RAW.
  EXEC SQL VAR EMP–NUMBER IS RAW (200) END–EXEC.
  EXEC SQL END DECLARE SECTION END–EXEC.
```

With host tables, the length you specify must match the buffer size required to hold the table. In the last example, you specified a length of 200, which is the buffer size needed to hold 50 4-byte integers.
Embedding PL/SQL

The Pro*COBOL Precompiler treats a PL/SQL block like a single embedded SQL statement. So, you can place a PL/SQL block anywhere in a host program that you can place a SQL statement.

To embed a PL/SQL block in your host program, declare the variables to be shared with PL/SQL and bracket the PL/SQL block with the EXEC SQL EXECUTE and END–EXEC keywords.

Host Variables

Inside a PL/SQL block, host variables are 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.

VARCHAR Variables

When entering a PL/SQL block, Oracle automatically checks the length fields of VARCHAR host variables, so you must set the length fields before the block is entered. For input variables, set the length field to the length of the value stored in the string field. For output variables, set the length field to the maximum length allowed by the string field.

Multi–Byte NLS Features

Multi–byte NLS features are not supported within a PL/SQL block. These features include N–quoted character literals (see Chapter 3 of the Programmer’s Guide to the Oracle Precompilers) and fixed–length character variables.

Indicator Variables

In a PL/SQL block, you cannot refer to an indicator variable by itself; it must be appended to its associated host variable. Also, if you refer to a host variable with its indicator variable, you must always refer to it that way in the same 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.

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.

SQLCHECK

You must specify SQLCHECK=SEMANTICS when precompiling a program with an embedded PL/SQL block. You must also use the USERID option. For more information, see Chapter 6 of the Programmer’s Guide to the Oracle Precompilers.
Cursor Variables

Starting with Release 1.7 of the Pro*COBOL Precompiler, you can use cursor variables in your Pro*COBOL programs to process multi-row queries using static embedded SQL. A cursor variable identifies a cursor reference that is defined and opened on the Oracle7 Server, Release 7.2 or later, 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.

- *Security*: the user of the application (the username when the Pro*COBOL application connected to the database) must have execute permission on the stored procedure that opens the cursor. This user, however, does not need to have read permission on the tables used in the query. This capability can be used to limit access to the columns in the table.

### Declaring a Cursor Variable

You declare a Pro*COBOL cursor variable using the SQL–CURSOR pseudotype. For example:

```
WORKING–STORAGE SECTION.
...
EXEC SQL BEGIN DECLARE SECTION END–EXEC.
...
01 CUR–VAR SQL–CURSOR.
...
EXEC SQL END DECLARE SECTION END–EXEC.
```

A SQL–CURSOR variable is implemented as a COBOL group item in the code that Pro*COBOL generates. A cursor variable is just like any other Pro*COBOL host variable.
Allocating a Cursor Variable

Before you can OPEN or FETCH from a cursor variable, you must initialize it using the Pro*COBOL ALLOCATE command. For example, to initialize the cursor variable CUR–VAR that was declared in the previous section, write the following statement:

```sql
EXEC SQL
ALLOCATE :CUR–VAR
END–EXEC.
```

Allocating a cursor variable does not require a call to the server, either at precompile time or at run time.

⚠️ **Warning:** Allocating a cursor variable does cause heap memory to be used. For this reason, avoid allocating a cursor variable in a program loop.

Opening a Cursor Variable

You must use an embedded anonymous PL/SQL block to open a cursor variable on the Oracle Server. The anonymous PL/SQL block may open the cursor either indirectly by calling a PL/SQL stored procedure that opens the cursor (and defines it in the same statement) or directly from the Pro*COBOL program.

Opening Indirectly through a Stored PL/SQL Procedure

Consider the following PL/SQL package stored in the database:

```sql
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
        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*COBOL program, and FETCH from the cursor variable EMP–CURSOR in the program. For example:

```cobol
WORKING-STORAGE SECTION.
    EXEC SQL BEGIN DECLARE SECTION END–EXEC.
    01  EMP–CURSOR SQL–CURSOR.
    01  DEPT–NUM PIC S9(4).
    01  EMP–NAME PIC X(10) VARYING.
    EXEC SQL END DECLARE SECTION END–EXEC.
    ...

PROCEDURE DIVISION.
    ...
    * Allocate the cursor variable.
    EXEC SQL
    ALLOCATE :EMP–CURSOR
    END–EXEC.
    ...
    MOVE 30 TO DEPT_NUM.
    * Open the cursor on the Oracle Server.
    EXEC SQL EXECUTE
    BEGIN
        demo_cur_pkg.open_emp_cur(:EMP–CURSOR, :DEPT–NUM);
    END;
    END–EXEC.
    EXEC SQL
    WHENEVER NOT FOUND DO PERFORM SIGN–OFF
    END–EXEC.
    FETCH–LOOP.
    EXEC SQL
    FETCH :EMP–CURSOR INTO :EMP–NAME
    END–EXEC.
    DISPLAY "Employee Name: ":EMP–NAME.
    GO TO FETCH–LOOP.
    ...
```
Opening Directly from Your Pro*COBOL Application

To open a cursor using a PL/SQL anonymous block in a Pro*COBOL program, define the cursor in the anonymous block. Consider the following example:

```cobol
PROCEDURE DIVISION.
...
EXEC SQL EXECUTE
BEGIN
   OPEN :EMP-CURSOR FOR SELECT ENAME FROM EMP
       WHERE DEPTNO = :DEPT-NUM;
   END;
END-EXEC.
...
```

Return Types

When you define a reference cursor (REF CURSOR) in a PL/SQL stored procedure, you must declare the type that the cursor returns. The return types allowed for reference cursors are described in the PL/SQL User’s Guide and Reference.

Fetching from a Cursor Variable

Use the embedded SQL FETCH .... INTO command to retrieve the rows SELECTed when you opened the cursor variable. For example:

```sql
EXEC SQL
   FETCH :EMP-CURSOR INTO :EMP-INFO:EMP-INFO-IND
END-EXEC.
```

Before you can FETCH from a cursor variable, the variable must be initialized and opened. You cannot FETCH from an unopened cursor variable.

Closing a Cursor Variable

Use the embedded SQL CLOSE command to close a cursor variable. For example:

```sql
EXEC SQL
   CLOSE :CUR-VAR
END-EXEC.
```
Restrictions

The following restrictions apply to the use of cursor variables:

- Cursor variables are not supported in 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 or unALLOCATEd 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.

Error Conditions

Do not perform any of the following operations:

- FETCH from a closed cursor variable
- use a cursor variable that is not ALLOCATEd
- CLOSE a cursor variable that is not open

These operations on cursor variables result in errors.

Sample Programs

The following sample programs — a SQL script (SAMPLE11.SQL) and a Pro*COBOL program (SAMPLE11.PCO) — demonstrate how you can use cursor variables in Pro*COBOL.

SAMPLE11.SQL

Following is the PL/SQL source code for a creating a package that declares and opens a cursor variable:

CONNECT SCOTT/TIGER
CREATE OR REPLACE PACKAGE emp_demo_pkg AS
    TYPE emp_cur_type IS REF CURSOR RETURN emp%ROWTYPE;
    PROCEDURE open_cur (cursor IN OUT emp_cur_type, dept_num IN number);
END emp_demo_pkg;
/
CREATE OR REPLACE PACKAGE BODY emp_demo_pkg AS
    PROCEDURE open_cur (cursor IN OUT emp_cur_type, dept_num IN number) IS
    BEGIN
        OPEN cursor FOR SELECT * FROM emp
         WHERE deptno = dept_num
         ORDER BY ename ASC;
    END;
END emp_demo_pkg;
/
Following is a Pro*COBOL sample program that uses the cursor declared in the SAMPLE11.SQL example to fetch employee names, salaries, and commissions from the EMP table.

IDENTIFICATION DIVISION.
PROGRAM-ID. CURSOR-VARIABLES.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
   EXEC ORACLE OPTION (SQLCHECK=FULL) END-EXEC.
   EXEC SQL BEGIN DECLARE SECTION END-EXEC.
      01 USERNAME PIC X(15) VARYING.
      01 PASSWD PIC X(15) VARYING.
      01 HOST PIC X(15) VARYING.
* Declare the cursor variable.
      01 EMP-CUR SQL-CURSOR.
      01 EMP-INFO.
         05 EMP-NUM PIC S9(4) COMP.
         05 EMP-NAM PIC X(10) VARYING.
         05 EMP-JOB PIC X(10) VARYING.
         05 EMP-MGR PIC S9(4) COMP.
         05 EMP-DAT PIC X(10) VARYING.
         05 EMP-SAL PIC S9(6)V99 DISPLAY SIGN LEADING SEPARATE.
         05 EMP-COM PIC S9(6)V99 DISPLAY SIGN LEADING SEPARATE.
      05 EMP-INFO-IND.
         05 EMP-NUM-IND PIC S9(2) COMP.
         05 EMP-NAM-IND PIC S9(2) COMP.
         05 EMP-JOB-IND PIC S9(2) COMP.
         05 EMP-MGR-IND PIC S9(2) COMP.
         05 EMP-DAT-IND PIC S9(2) COMP.
         05 EMP-SAL-IND PIC S9(2) COMP.
         05 EMP-COM-IND PIC S9(2) COMP.
         05 EMP-DEP-IND PIC S9(2) COMP.
   EXEC SQL END DECLARE SECTION END-EXEC.
   EXEC SQL INCLUDE SQLCA END-EXEC.
      01 DISPLAY-VARIABLES.
         05 D-DEP-NUM PIC Z(3)9.
         05 D-EMP-NAM PIC X(10).
         05 D-EMP-SAL PIC Z(4)9.99.
         05 D-EMP-COM PIC Z(4)9.99.
PROCEDURE DIVISION.
BEGIN-PGM.
EXEC SQL
   WHENEVER SQLERROR DO PERFORM SQL-ERROR
END-EXEC.
PERFORM LOGON.

* Initialize the cursor variable.
EXEC SQL
   ALLOCATE :EMP-CUR
END-EXEC.
DISPLAY "Enter department number (0 to exit):  
   WITH NO ADVANCING.
ACCEPT EMP-DEP.
IF EMP-DEP <= 0
   PERFORM SIGN-OFF
END-IF.
MOVE EMP-DEP TO D-DEP-NUM.

* Open the cursor by calling a PL/SQL stored procedure.
EXEC SQL EXECUTE
   BEGIN
      emp_demo_pkg.open_cur(:EMP-CUR, :EMP-DEP);
   END;
END-EXEC.
DISPLAY " 
DISPLAY "For department ", D-DEP-NUM, ":".
DISPLAY " 
DISPLAY "EMPLOYEE  SALARY    COMMISSION".
DISPLAY "-------------  ---------------  ".

FETCH-LOOP.
EXEC SQL
   WHENEVER NOT FOUND DO PERFORM SIGN-OFF
END-EXEC.
MOVE SPACES TO EMP-NAM-ARR.

* Fetch data from the cursor into the host variables.
EXEC SQL FETCH :EMP-CUR
   INTO :EMP-NUM:EMP-NUM-IND,
       :EMP-NAM:EMP-NAM-IND,
       :EMP-JOB:EMP-JOB-IND,
       :EMP-MGR:EMP-MGR-IND,
       :EMP-DAT:EMP-DAT-IND,
       :EMP-SAL:EMP-SAL-IND,
       :EMP-COM:EMP-COM-IND,
       :EMP-DEP:EMP-DEP-IND
END-EXEC.
MOVE EMP-SAL TO D-EMP-SAL.
MOVE EMP-COM TO D-EMP-COM.
* Check for commission and print results.
IF EMP-COM-IND = 0
   DISPLAY EMP-NAM-ARR, "   ", D-EMP-SAL,
   "   ", D-EMP-COM
ELSE
   DISPLAY EMP-NAM-ARR, "   ", D-EMP-SAL,
   "        N/A"
END-IF.
GO TO FETCH-LOOP.

LOGON.
MOVE "SCOTT" TO USERNAME-ARR.
MOVE 5 TO USERNAME-LEN.
MOVE "TIGER" TO PASSWD-ARR.
MOVE 5 TO PASSWD-LEN.
MOVE "INST1_ALIAS" TO HOST-ARR.
MOVE 11 TO HOST-LEN.
EXEC SQL
   CONNECT :USERNAME IDENTIFIED BY :PASSWD
END-EXEC.
DISPLAY "   ".
DISPLAY "CONNECTED TO ORACLE AS USER: ", USERNAME-ARR.

SIGN-OFF.
* Close the cursor variable.
EXEC SQL
   CLOSE :EMP-CUR
END-EXEC.
DISPLAY "   ".
DISPLAY "HAVE A GOOD DAY.".
DISPLAY "   ".
EXEC SQL
   COMMIT WORK RELEASE
END-EXEC.
STOP RUN.

SQL-ERROR.
EXEC SQL
   WHENEVER SQLERROR CONTINUE
END-EXEC.
DISPLAY "   ".
DISPLAY "ORACLE ERROR DETECTED:"
DISPLAY "   ".
DISPLAY SQLERRMC.
EXEC SQL
   ROLLBACK WORK RELEASE
END-EXEC.
STOP RUN.
Connecting to Oracle

Your Pro*COBOL program must log on to Oracle before querying or manipulating data. To log on, you use the CONNECT statement, as in

```
EXEC SQL
    CONNECT :USERNAME IDENTIFIED BY :PASSWD
END-EXEC.
```

where USERNAME and PASSWD are PIC X(n) or PIC X(n) VARYING host variables. Alternatively, you can use the statement

```
EXEC SQL
    CONNECT :USR–PWD
END-EXEC.
```

where the host variable USR–PWD contains your username and password separated by a slash (/).

The CONNECT statement must be the first SQL statement executed by the program. That is, other executable SQL statements can positionally, but not logically, precede the CONNECT statement.

To supply the Oracle username and password separately, you define two host variables in the Declare Section as character strings or VARCHAR variables. If you supply a userid 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:

```
WORKING STORAGE SECTION.

... EXEC SQL BEGIN DECLARE SECTION END-EXEC.
   01  USERNAME  PIC X(10) VARYING.
   01  PASSWD    PIC X(10) VARYING.
...
EXEC SQL END DECLARE SECTION END-EXEC.
...

PROCEDURE DIVISION.

LOGON.
   MOVE “SCOTT” TO USERNAME–ARR.
   MOVE 5 TO USERNAME–LEN.
   MOVE “TIGER” TO PASSWD–ARR.
   MOVE 5 TO PASSWD–LEN.
   EXEC SQL WHENEVER SQLERROR GOTO LOGON–ERROR END-EXEC.
   EXEC SQL
       CONNECT :USERNAME IDENTIFIED BY :PASSWD
   END-EXEC.
```
However, you cannot hardcode a username and password into the CONNECT statement or use quoted literals. For example, the following statements are invalid:

```cobol
EXEC SQL
  CONNECT SCOTT IDENTIFIED BY TIGER
END-EXEC.

EXEC SQL
  CONNECT "SCOTT" IDENTIFIED BY "TIGER"
END-EXEC.
```
**Automatic Logons**

You can automatically log on to the Oracle using the following userid:

\<prefix\>\<username\>

where *prefix* is the value of the Oracle initialization parameter `OS_AUTHENT_PREFIX` (the default value is `OPS$`) and *username* is your operating system user or task name. For example, if the prefix is `OPS$`, your user name is `TBARNES`, and `OPS$TBARNES` is a valid Oracle userid, you log on to Oracle as user `OPS$TBARNES`.

Consider the following example:

```
WORKING-STORAGE SECTION.
...
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
...
  01 ORACLE-ID PIC X(1) VALUE "/".
...
EXEC SQL END DECLARE SECTION END-EXEC.

PROCEDURE DIVISION.
...
EXEC SQL
  CONNECT :ORACLE-ID
END-EXEC.
...
```

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 a slash (`/`) automatically logs you on to Oracle as user `OPS$RHILL`.

You can also pass a character string to the precompiler. However, the string cannot contain trailing blanks. For example, the following CONNECT statement will fail:

```
WORKING-STORAGE SECTION.
...
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
...
  01 ORACLE-ID PIC X(5) VALUE "  ".
EXEC SQL END DECLARE SECTION END-EXEC.

PROCEDURE DIVISION.
EXEC SQL
  CONNECT :ORACLE-ID
END-EXEC.
...
```

For more information about operating system authentication, see the *Oracle7 Server Administrator’s Guide*. 
Concurrent Logons

Your application can use SQL*Net to access any combination of remote and local databases concurrently or make multiple connections to the same database. In the following example, you connect to two non–default databases concurrently:

```
WORKING-STORAGE SECTION.
...
* Declare needed host variables.
  EXEC SQL BEGIN DECLARE SECTION END-EXEC.
    01 USERNAME   PIC X(5)   VALUE "SCOTT".
    01 PASSWD    PIC X(5)   VALUE "TIGER".
    01 DB-STRING1 PIC X(12) VALUE "NEWYORK".
    01 DB-STRING2 PIC X(12) VALUE "BOSTON".
  EXEC SQL END DECLARE SECTION END-EXEC.
...
PROCEDURE DIVISION.
...
* Give each database connection a unique name.
  EXEC SQL DECLARE DB-NAME1 DATABASE END-EXEC.
  EXEC SQL DECLARE DB-NAME2 DATABASE END-EXEC.

* Connect to the two non-default databases.
  EXEC SQL
    CONNECT :USERNAME IDENTIFIED BY :PASSWD
    AT DB-NAME1 USING :DB-STRING1
  END-EXEC.
  EXEC SQL
    CONNECT :USERNAME IDENTIFIED BY :PASSWD
    AT DB-NAME2 USING :DB-STRING2
  END-EXEC.
```

The string syntax in DB–STRING1 and DB–STRING2 depends on your network driver and how it is configured. DB–NAME1 and DB–NAME2 name the non–default connections; they can be undeclared identifiers or host variables.

For step–by–step instructions on connecting to Oracle via SQL*Net, see Chapter 3 in the Programmer’s Guide to the Oracle Precompilers.
This chapter supplements Chapter 8 of the *Programmer’s Guide to the Oracle Precompilers*. It discusses error reporting and recovery as it applies to Pro*COBOL.

You learn how to declare and use the SQLSTATE status variable, the SQL Communications Area (SQLCA), and the SQLCODE status variable. You also learn how to declare and enable the Oracle Communications Area (ORACA).
Error Handling Alternatives

The Pro*COBOL Precompiler supports four status variables that serve as error handling mechanisms:

- SQLCODE
- SQLSTATE
- SQLCA (using the WHENEVER statement)
- ORACA

The precompiler MODE option governs ANSI/ISO compliance. The availability of the SQLCODE, SQLSTATE, and SQLCA variables depends on the MODE setting. You can declare and use the ORACA variable regardless of the MODE setting. For more information, see “Using the Oracle Communications Area” on page 2 – 17.

When MODE={ORACLE | ANSI13}, you must declare the SQLCA status variable. SQLCODE and SQLSTATE declarations are accepted (not recommended) but are not recognized as status variables. For more information, see “Using the SQL Communications Area” on page 2 – 9.

When MODE={ANSI | ANSI14}, you can use any one, two, or all three of the SQLCODE, SQLSTATE, and SQLCA variables. To determine which variable (or variable combination) is best for your application, see “Using Status Variables when MODE={ANSI | ANSI14}” on page 2 – 3.

SQLCODE and SQLSTATE

With Pro*COBOL, Release 1.5, the SQLCODE status variable was introduced as the SQL89 standard ANSI/ISO error reporting mechanism. The SQL92 standard listed SQLCODE as a deprecated feature and defined a new status variable, SQLSTATE (introduced with Pro*COBOL, Release 1.6), as the preferred ANSI/ISO error reporting mechanism.

SQLCODE stores error codes and the “not found” condition. It is retained only for compatibility with SQL89 and is likely to be removed from future versions of the standard.

Unlike SQLCODE, SQLSTATE stores error and warning codes and uses a standardized coding scheme. After executing a SQL statement, the Oracle server returns a status code to the SQLSTATE variable currently in scope. The status code indicates whether a 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.
SQLCA

The SQLCA is a record–like, host–language data structure. Oracle updates the SQLCA after *every executable* SQL statement. (SQLCA values are undefined 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 two ways:

- implicit checking with the WHENEVER statement
- explicit checking of SQLCA variables

You can use WHENEVER statements, code explicit checks on SQLCA variables, or do both. Generally, using WHENEVER statements is preferable because it is easier, more portable, and ANSI–compliant.

ORACA

When more information is needed about runtime errors than the SQLCA provides, you can use the ORACA, which contains cursor statistics, SQL statement data, option settings, and system statistics.

The ORACA is optional and can be declared regardless of the MODE setting. For more information about the ORACA status variable, see “Using the Oracle Communications Area” on page 2 – 17.

Using Status Variables when MODE={ANSI | ANSI14}

When MODE={ANSI | ANSI14}, you must declare at least one — you may declare two or all three — of the following status variables:

- SQLCODE
- SQLSTATE
- SQLCA

You cannot declare SQLCODE if SQLCA is declared. Likewise, you cannot declare SQLCA if SQLCODE is declared. The field in the SQLCA data structure that stores the error code for is also called SQLCODE, so errors will occur if both status variables are declared.

Your program can get the outcome of the most recent executable SQL statement by checking SQLCODE and/or SQLSTATE explicitly with your own code after executable SQL and PL/SQL statements. Your program can also check SQLCA implicitly (with the WHENEVER SQLERROR and WHENEVER SQLWARNING statements) or it can check the SQLCA variables explicitly.

**Note:** When MODE={ORACLE | ANSI13 | ANSI14}, you must declare the SQLCA status variable. For more information, see “Using the SQL Communications Area” on page 2 – 9.
Some Historical Information

The treatment of status variables and variable combinations by the Oracle Pro*COBOL Precompiler has evolved beginning with Release 1.5.

Release 1.5

Pro*COBOL, Release 1.5, presumed there was a status variable SQLCODE whether or not it was declared in a Declare Section; in fact, the precompiler never noted whether SQLCODE was declared or not — it just presumed it was. SQLCA would be used as a status variable if and only if there was an INCLUDE of the SQLCA.

Release 1.6

Beginning with Pro*COBOL, Release 1.6, the precompiler no longer presumes that there is a SQLCODE status variable and it is not required. The precompiler requires that at least one of SQLCA, SQLCODE, or SQLSTATE be declared.

SQLCODE is recognized as a status variable if and only if at least one of the following criteria is satisfied:

- It is declared in a Declare Section with exactly the right datatype.
- The precompiler finds no other status variable.

If the precompiler finds a SQLSTATE declaration (of exactly the right type of course) in a Declare Section or finds an INCLUDE of the SQLCA, it will not presume SQLCODE is declared.

Release 1.7

Because Pro*COBOL, Release 1.5, allowed the SQLCODE variable to be declared outside of a Declare Section while also declaring SQLCA, Pro*COBOL, Release 1.6 and greater, is presented with a compatibility problem. A new option, ASSUME_SQLCODE={YES | NO} (default NO), was added to fix this in Release 1.6.7 and is documented as a new feature in Release 1.7.

When ASSUME_SQLCODE=YES, and when SQLSTATE is declared as a status variable, the precompiler presumes SQLCODE is declared whether or not it is declared in a Declare Section or of the proper type. This causes Releases 1.6.7 and later to act like Release 1.5 in this regard. For information about the precompiler option ASSUME_SQLCODE, see Chapter 6 in the Programmer’s Guide to the Oracle Precompilers.
Declaring Status Variables

This section describes how to declare SQLCODE and SQLSTATE. For information about declaring the SQLCA status variable, see “Declaring the SQLCA” on page 2–11.

Declaring SQLCODE

SQLCODE must be declared as a 4-byte integer variable either inside or outside the Declare Section, as shown in the following example:

*    Declare host and indicator variables.
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
...
EXEC SQL END DECLARE SECTION END-EXEC.
*    Declare the SQLCODE status variable.
01 SQLCODE PIC S9(9) COMP.

If declared outside the Declare Section, SQLCODE is recognized as a status variable if and only if ASSUME_SQLCODE=YES. When MODE={ORACLE | ANSI13 | ANSI14}, declarations of the SQLCODE variable are ignored.

⚠️ Warning: Do not declare SQLCODE if SQLCA is declared. Likewise, do not declare SQLCA if SQLCODE is declared. The status variable declared by the SQLCA structure is also called SQLCODE, so errors will occur if both error-reporting mechanisms are used.

After every SQL operation, Oracle returns a status code to the SQLCODE variable. 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.

Declaring SQLSTATE

SQLSTATE must be declared as a five-character alphanumeric string inside the Declare Section, as shown in the following example:

* Declare the SQLSTATE status variable.
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
...
01 SQLSTATE PIC X(5).
...
EXEC SQL END DECLARE SECTION END-EXEC.

When MODE={ORACLE | ANSI13 | ANSI14}, SQLSTATE declarations are ignored. Declaring the SQLCA is optional.
When MODE={ANSI | ANSI14}, the behavior of the status variables depends on the following:

- which variables are declared
- declaration placement (inside or outside the Declare Section)
- ASSUME_SQLCODE setting

Table 2–1 and Table 2–2 describe the resulting behavior of each status variable combination when ASSUME_SQLCODE=NO and when ASSUME_SQLCODE=YES, respectively.
<table>
<thead>
<tr>
<th>Declare Section (IN/OUT/—)</th>
<th>SQLCODE</th>
<th>SQLSTATE</th>
<th>SQLCA</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>OUT — — —</td>
<td>SQLCODE is declared and is presumed to be a status variable.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUT — OUT —</td>
<td>This status variable configuration is not supported.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUT — IN —</td>
<td>This status variable configuration is not supported.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUT OUT — —</td>
<td>SQLCODE is declared and is presumed to be a status variable, and SQLSTATE is declared but is not recognized as a status variable.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUT OUT OUT —</td>
<td>This status variable configuration is not supported.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUT OUT IN —</td>
<td>This status variable configuration is not supported.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUT IN — —</td>
<td>SQLSTATE is declared as a status variable, and SQLCODE is declared but is not recognized as a status variable.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUT IN OUT —</td>
<td>This status variable configuration is not supported.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUT IN IN —</td>
<td>This status variable configuration is not supported.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IN — IN —</td>
<td>SQLCODE is declared as a status variable.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IN — OUT IN</td>
<td>This status variable configuration is not supported.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IN — IN IN</td>
<td>This status variable configuration is not supported.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IN OUT — —</td>
<td>SQLCODE is declared as a status variable, and SQLSTATE is declared but is not recognized as a status variable.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IN OUT OUT IN</td>
<td>This status variable configuration is not supported.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IN OUT IN —</td>
<td>This status variable configuration is not supported.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IN IN — —</td>
<td>SQLCODE and SQLSTATE are declared as a status variables.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IN IN OUT IN</td>
<td>This status variable configuration is not supported.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IN IN IN —</td>
<td>This status variable configuration is not supported.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>— — — —</td>
<td>This status variable configuration is not supported.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>— — OUT —</td>
<td>SQLCA is declared as a status variable.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>— — IN —</td>
<td>SQLCA is declared as a status host variable.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>— OUT — —</td>
<td>This status variable configuration is not supported.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>— OUT OUT —</td>
<td>SQLCA is declared as a status variable, and SQLSTATE is declared but is not recognized as a status variable.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>— OUT IN —</td>
<td>SQLCA is declared as a status host variable, and SQLSTATE is declared but is not recognized as a status variable.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>— IN — —</td>
<td>SQLSTATE is declared as a status variable.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>— IN OUT —</td>
<td>SQLSTATE and SQLCA are declared as status variables.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>— IN IN —</td>
<td>SQLSTATE and SQLCA are declared as status host variables.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2 – 1 Status Variable Behavior with ASSUME_SQLCODE=NO and MODE=ANSI | ANSI14
<table>
<thead>
<tr>
<th>Declare Section (IN/OUT/—)</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLCODE</td>
<td>SQLSTATE</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>OUT</td>
<td>—</td>
</tr>
<tr>
<td>OUT</td>
<td>—</td>
</tr>
<tr>
<td>OUT</td>
<td>—</td>
</tr>
<tr>
<td>OUT</td>
<td>OUT</td>
</tr>
<tr>
<td>OUT</td>
<td>OUT</td>
</tr>
<tr>
<td>OUT</td>
<td>OUT</td>
</tr>
<tr>
<td>OUT</td>
<td>IN</td>
</tr>
<tr>
<td>OUT</td>
<td>IN</td>
</tr>
<tr>
<td>OUT</td>
<td>IN</td>
</tr>
<tr>
<td>IN</td>
<td>—</td>
</tr>
<tr>
<td>IN</td>
<td>—</td>
</tr>
<tr>
<td>IN</td>
<td>—</td>
</tr>
<tr>
<td>IN</td>
<td>OUT</td>
</tr>
<tr>
<td>IN</td>
<td>OUT</td>
</tr>
<tr>
<td>IN</td>
<td>OUT</td>
</tr>
<tr>
<td>IN</td>
<td>IN</td>
</tr>
<tr>
<td>IN</td>
<td>IN</td>
</tr>
<tr>
<td>IN</td>
<td>IN</td>
</tr>
<tr>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>—</td>
<td>OUT</td>
</tr>
<tr>
<td>—</td>
<td>IN</td>
</tr>
<tr>
<td>—</td>
<td>IN</td>
</tr>
<tr>
<td>—</td>
<td>IN</td>
</tr>
</tbody>
</table>

Table 2-2 Status Variable Behavior with ASSUME_SQLCODE= YES and MODE=ANSI | ANSI14
Using the SQL Communications Area

Oracle uses the SQL Communications Area (SQLCA) to store status information passed to your program at run time. The SQLCA is a record–like, COBOL data structure that is updated after each executable SQL statement, so it always reflects the outcome of the most recent SQL operation. To determine that outcome, you can check variables in the SQLCA explicitly with your own COBOL code or implicitly with the WHENEVER statement.

When MODE={ORACLE|ANSI13}, the SQLCA is required; if the SQLCA is not declared, compile–time errors will occur. The SQLCA is optional when MODE={ANSI|ANSI14}, 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.

When MODE={ANSI|ANSI14}, you must declare either SQLSTATE (see “Declaring SQLSTATE” on page 2 – 5) or SQLCODE (see “Declaring SQLCODE” on page 2 – 5) or both. The SQLSTATE status variable supports the SQLSTATE status variable specified by the SQL92 standard. You can use the SQLSTATE status variable with or without SQLCODE. For more information see Chapter 8 of the Programmer’s Guide to the Oracle Precompilers.
What’s in the SQLCA? The SQLCA contains runtime information about the execution of SQL statements, such as Oracle error codes, warning flags, event information, rows–processed count, and diagnostics.

Figure 2–1 shows all the variables in the SQLCA. However, SQLWARN2, SQLWARN5, SQLWARN6, SQLWARN7, and SQLEXT are not currently in use.

```
01 SQLCA.
   05 SQLCAID               PIC X(8).
   05 SQLCABC               PIC S9(9) COMPUTATIONAL.
   05 SQLCODE               PIC S9(9) COMPUTATIONAL.
   05 SQLERRM.
      49 SQLERRML           PIC S9(4) COMPUTATIONAL.
      49 SQLERRMC           PIC X(70)
   05 SQLERRF               PIC X(8).
   05 SQLERRD OCCURS 6 TIMES
                              PIC S9(9) COMPUTATIONAL.
   05 SQLWARN.
      10 SQLWARNO           PIC X(1).
      10 SQLWARN1           PIC X(1).
      10 SQLWARN2           PIC X(1).
      10 SQLWARN3           PIC X(1).
      10 SQLWARN4           PIC X(1).
      10 SQLWARN5           PIC X(1).
      10 SQLWARN6           PIC X(1).
      10 SQLWARN7           PIC X(1).
   05 SQLEXT                PIC X(8).
```

Figure 2–1  SQLCA Variable Declarations for Pro*COBOL

For a full description of the SQLCA, its fields, and the values its fields can store, see Chapter 8 of the *Programmer’s Guide to the Oracle Precompilers*. 
Declaring the SQLCA

To declare the SQLCA, simply include it (using an EXEC SQL INCLUDE statement) in your Pro*COBOL source file outside the Declare Section as follows:

```cobol
EXEC SQL INCLUDE SQLCA END-EXEC.
```

The SQLCA must be declared outside the Declare Section.

⚠️ **Warning:** Do not declare SQLCODE if SQLCA is declared. Likewise, do not declare SQLCA if SQLCODE is declared. The status variable declared by the SQLCA structure is also called SQLCODE, so errors will occur if both error-reporting mechanisms are used.

When you precompile your program, the INCLUDE SQLCA statement is replaced by several variable declarations that allow Oracle to communicate with the program.

⚠️ **Attention:** When using multi-byte NLS host variables, the SQLCA must be included.

Key Components of Error Reporting

The key components of Pro*COBOL error reporting depend on several fields in the SQLCA.

### Status Codes

Every executable SQL statement returns a status code in the SQLCA variable SQLCODE, which you can check implicitly with WHENEVER SQLERROR or explicitly with your own COBOL code.

### Warning Flags

Warning flags are returned in the SQLCA variables SQLWARN0 through SQLWARN7, which you can check with WHENEVER SQLWARNING or with your own COBOL code. These warning flags are useful for detecting runtime conditions that are not considered errors by Oracle.

### Rows-Processed Count

The number of rows processed by the most recently executed SQL statement is returned in the SQLCA variable SQLERRD(3). For repeated FETCHes on an OPEN cursor, SQLERRD(3) keeps a running total of the number of rows fetched.

### 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 SQLERRD(5), which you can check explicitly. The offset specifies the character position in the SQL statement at which the parse error begins. The first character occupies position zero. For example, if the offset is 9, the parse error begins at the tenth character.
If your SQL statement does not cause a parse error, Oracle sets SQLERRD(5) to zero. Oracle also sets SQLERRD(5) to zero if a parse error begins at the first character (which occupies position zero). So, check SQLERRD(5) only if SQLCODE is negative, which means that an error has occurred.

The error code and message for Oracle errors are available in the SQLCA variable SQLERRMC. For example, you might place the following statements in an error-handling routine:

*    Handle SQL execution errors.
    MOVE SQLERRMC TO ERROR-MESSAGE.
    DISPLAY ERROR-MESSAGE.

At most, the first 70 characters of message text are stored. For messages longer than 70 characters, you must call the SQLGLM subroutine, which is discussed next.

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 subroutine.

If connected to Oracle, you can call SQLGLM using the syntax

```
CALL "SQLGLM" USING MSG-TEXT, MAX-SIZE, MSG-LENGTH
```

where:

- MSG-TEXT is the field in which to store the error message. (Oracle blank-pads to the end of this field.)
- MAX-SIZE is an integer that specifies the maximum size of the MSG-TEXT field in bytes.
- MSG-LENGTH is an integer variable in which Oracle stores the actual length of the error message.

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 specified for MAX-SIZE.
The following example uses SQLGLM to get an error message of up to 200 characters in length:

```
WORKING–STORAGE SECTION.
...
* Declare variables for the SQL–ERROR subroutine call.
  01  MSG–TEXT    PIC X(200).
  01  MAX–SIZE    PIC S9(9) COMP VALUE 200.
  01  MSG–LENGTH  PIC S9(9) COMP.
...
PROCEDURE DIVISION.
MAIN.
  EXEC SQL WHENEVER SQLERROR GOTO SQL–ERROR END–EXEC.
...
SQL–ERROR.
* Clear the previous message text.
  MOVE SPACES TO MSG–TEXT.
* Get the full text of the error message.
  CALL "SQLGLM" USING MSG–TEXT, MAX–SIZE, MSG–LENGTH.
  DISPLAY MSG–TEXT.
```

In the example, SQLGLM is called only when a SQL error has occurred. Always make sure SQLCODE is negative before calling SQLGLM. If you call SQLGLM when SQLCODE is zero, you get the message text associated with a prior SQL statement.

**Note:** If your application calls SQLGLM to get message text or your Oracle*Forms user exit calls SQLIEM to display a failure message, the message length must be passed. Do not use the SQLCA variable SQLERRML; SQLERRML is a PIC S9(4) COMP integer while SQLGLM and SQLIEM expect a PIC S9(9) COMP integer. Instead, use another variable declared as PIC S9(9) COMP.

**Using the WHENEVER Statement**

By default, the Pro*COBOL Precompiler ignores Oracle error and warning conditions and continues 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, PERFORMing a paragraph, branching to a paragraph, or stopping.

Code the WHENEVER statement using the following syntax:

```
EXEC SQL
   WHENEVER <condition> <action>
END–EXEC.
```
You can have Oracle automatically check the SQLCA for any of the following conditions, which are described in the *Programmer’s Guide to the Oracle Precompilers*.

- SQLWARNING
- SQLERROR
- NOT FOUND

When Oracle detects one of the preceding conditions, you can have your program take any of the following actions:

- CONTINUE
- DO PERFORM `paragraph_name`
- GOTO `paragraph_name`
- STOP

When using the WHENEVER ... DO statement, the usual rules for PERFORMing a paragraph apply. However, you cannot use the THRU, TIMES, UNTIL, or VARYING clauses.

For example, the following WHENEVER ... DO statement is invalid:

```cobol
PROCEDURE DIVISION.
  * Invalid statement
  EXEC SQL WHENEVER SQLERROR DO
        PERFORM DISPLAY-ERROR THRU LOG-OFF
  END-EXEC.
  ...
  DISPLAY-ERROR.
  ...
  LOG-OFF.
  ...
```
In the following example, WHENEVER SQLERROR DO statements are used to handle specific errors:

PROCEDURE DIVISION.
MAIN.
...
EXEC SQL
WHENEVER SQLERROR DO PERFORM INS-ERROR
END-EXEC.
EXEC SQL
INSERT INTO EMP (EMPNO, ENAME, DEPTNO)
VALUES (:EMP-NUMBER, :EMP-NAME, :DEPT-NUMBER)
END-EXEC.
EXEC SQL
WHENEVER SQLERROR DO PERFORM DEL-ERROR
END-EXEC.
EXEC SQL
DELETE FROM DEPT
WHERE DEPTNO = :DEPT-NUMBER
END-EXEC.
...
* Error-handling paragraphs.
INS-ERROR.
* Check for “duplicate key value” Oracle error
  IF SQLCODE IN SQLCA = -1
    ...
* Check for “value too large” Oracle error
  ELSE IF SQLCODE IN SQLCA = -1401
    ...
  ELSE
    ...
  END-IF.
...
DEL-ERROR.
* Check for the number of rows processed.
  IF SQLERRORD(3) IN SQLCA = 0
    ...
  ELSE
    ...
  END-IF.
...

Notice how the paragraphs check variables in the SQLCA to determine a course of action. For more information about the WHENEVER conditions and actions, see Chapter 8 of the Programmer’s Guide to the Oracle Precompilers.
Because WHENEVER is a declarative statement, its scope is positional, not logical. It tests all executable SQL statements that 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.

**Suggestion:** You might want to place WHENEVER statements at the beginning of each program unit that contains SQL statements. That way, SQL statements in one program unit will not reference WHENEVER actions in another program unit, causing errors at compile or run time.

Careless use of the WHENEVER statement can cause problems. For example, the following code enters an infinite loop if the DELETE statement sets the NOT FOUND condition, because no rows meet the search condition:

```
* Improper use of WHENEVER.
EXEC SQL WHENEVER NOT FOUND GOTO NO–MORE END–EXEC.
PERFORM GET–ROWS UNTIL DONE = "YES".
...
GET–ROWS.
EXEC SQL FETCH EMP–CURSOR INTO :EMP–NAME, :SALARY END–EXEC.
...
NO–MORE.
MOVE "YES" TO DONE.
EXEC SQL DELETE FROM EMP WHERE EMPNO = :EMP–NUMBER END–EXEC.
...
```

In the next example, the NOT FOUND condition is properly handled by resetting the GOTO target:

```
* Proper use of WHENEVER.
EXEC SQL WHENEVER NOT FOUND GOTO NO–MORE END–EXEC.
PERFORM GET–ROWS UNTIL DONE = "YES".
...
GET–ROWS.
EXEC SQL FETCH EMP–CURSOR INTO :EMP–NAME, :SALARY END–EXEC.
...
NO–MORE.
MOVE "YES" TO DONE.
EXEC SQL WHENEVER NOT FOUND GOTO NONE–FOUND END–EXEC.
EXEC SQL DELETE FROM EMP WHERE EMPNO = :EMP–NUMBER END–EXEC.
...
NONE–FOUND.
...
```
Using the Oracle Communications Area

The SQLCA handles standard SQL communications. The Oracle Communications Area (ORACA) is a similar structure that you can include in your program to handle Oracle–specific communications. When you need more runtime information than the SQLCA provides, use the ORACA.

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, an area of memory reserved for cursor management.

What’s in the ORACA? The ORACA contains option settings, system statistics, and extended diagnostics. Figure 2–2 shows all the variables in the ORACA.

```
01 ORACA
  05 ORACAID PIC X(8).
  05 ORACABC PIC S9(9) COMPUTATIONAL.
  05 ORACCHF PIC S9(9) COMPUTATIONAL.
  05 ORADBF PIC S9(9) COMPUTATIONAL.
  05 ORACHCF PIC S9(9) COMPUTATIONAL.
  05 ORASTXTF PIC S9(9) COMPUTATIONAL.
  05 ORASTXT.
    49 ORASTXTL PIC S9(4) COMPUTATIONAL.
    49 ORASTXTL PIC X(70).
  05 ORASFMN.
    49 ORASFNML PIC S9(4) COMPUTATIONAL.
    49 ORASFNMC PIC X(70).
  05 ORASLNR PIC S9(9) COMPUTATIONAL.
  05 ORAHCOC PIC S9(9) COMPUTATIONAL.
  05 ORAMOC PIC S9(9) COMPUTATIONAL.
  05 ORACC PIC S9(9) COMPUTATIONAL.
  05 ORANOR PIC S9(9) COMPUTATIONAL.
  05 ORANFR PIC S9(9) COMPUTATIONAL.
  05 ORANEX PIC S9(9) COMPUTATIONAL.
```

Figure 2–2 ORACA Variable Declarations for Pro*COBOL
Declaring the ORACA  To declare the ORACA, simply include it (using an EXEC SQL INCLUDE statement) in your Pro*COBOL source file outside the Declare Section as follows:

*   Include the Oracle Communications Area (ORACA).
   EXEC SQL INCLUDE ORACA END–EXEC.

Enabling the ORACA  To enable the ORACA, you must set the ORACA precompiler option to YES on the command line or in a configuration file with

ORACA=YES

or inline with

   EXEC Oracle OPTION (ORACA=YES) END–EXEC.

Then, you must choose appropriate runtime options by setting flags in the ORACA. Enabling the ORACA is optional because it adds to runtime overhead. The default setting is ORACA=NO.
This chapter provides several embedded SQL programs to guide you in writing your own. These programs illustrate the key concepts and features of Pro*COBOL programming and demonstrate techniques that let you take full advantage of SQL's power and flexibility.

Each sample program in this chapter is available online. Table 3 – 1 shows the usual filenames of the sample programs. However, the exact filenames are system-dependent. For specific filenames, see your Oracle system–specific documentation.

<table>
<thead>
<tr>
<th>Filename</th>
<th>Demonstrates</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAMPLE1.PCO</td>
<td>a simple query</td>
</tr>
<tr>
<td>SAMPLE2.PCO</td>
<td>cursor operations</td>
</tr>
<tr>
<td>SAMPLE3.PCO</td>
<td>array fetches</td>
</tr>
<tr>
<td>SAMPLE4.PCO</td>
<td>datatype equivalencing</td>
</tr>
<tr>
<td>SAMPLE5.PCO</td>
<td>an Oracle Forms user exit</td>
</tr>
<tr>
<td>SAMPLE6.PCO</td>
<td>dynamic SQL Method 1</td>
</tr>
<tr>
<td>SAMPLE7.PCO</td>
<td>dynamic SQL Method 2</td>
</tr>
<tr>
<td>SAMPLE8.PCO</td>
<td>dynamic SQL Method 3</td>
</tr>
<tr>
<td>SAMPLE9.PCO</td>
<td>calling a stored procedure</td>
</tr>
</tbody>
</table>

Table 3 – 1  Pro*COBOL Sample Programs
Sample Program 1: Simple Query

This program logs on to Oracle, prompts the user for an employee number, queries the database for the employee’s name, salary, and commission, then displays the result. The program ends when the user enters a zero employee number.

IDENTIFICATION DIVISION.
PROGRAM-ID. QUERY.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.

EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01 USERNAME PIC X(10) VARYING.
  01 PASSWD PIC X(10) VARYING.
  01 EMP-REC-VARS.
      05 EMP-NAME PIC X(10) VARYING.
      05 EMP-NUMBER PIC S9(4) COMP VALUE ZERO.
      05 SALARY PIC S9(5)V99 COMP-3 VALUE ZERO.
      05 COMMISSION PIC S9(5)V99 COMP-3 VALUE ZERO.
      05 COMM-IND PIC S9(4) COMP VALUE ZERO.
EXEC SQL END DECLARE SECTION END-EXEC.

EXEC SQL INCLUDE SQLCA END-EXEC.

01 DISPLAY-VARIABLES.
  05 D-EMP-NAME PIC X(10).
  05 D-SALARY PIC Z(4)9.99.
  05 D-COMMISSION PIC Z(4)9.99.

  01 D-TOTAL-QUERIED PIC 9(4) VALUE ZERO.

PROCEDURE DIVISION.
BEGIN-PGM.
 EXEC SQL
  WHENEVER SQLERROR DO PERFORM SQL-ERROR
END-EXEC.
PERFORM LOGON.

QUERY-LOOP.
DISPLAY " ".
DISPLAY "ENTER EMP NUMBER (0 TO QUIT): " WITH NO ADVANCING.
ACCEPT EMP-NUMBER
IF (EMP-NUMBER = 0) PERFORM SIGN-OFF.
MOVE SPACES TO EMP-NAME-ARR.
EXEC SQL
  WHENEVER NOT FOUND GOTO NO-EMP
END-EXEC.
EXEC SQL
  SELECT ENAME, SAL, COMM
  FROM EMP
  WHERE EMPNO = :EMP–NUMBER
END-EXEC.
PERFORM DISPLAY–INFO.
ADD 1 TO D–TOTAL–QUERIED.
GO TO QUERY–LOOP.

NO-EMP.
DISPLAY "NOT A VALID EMPLOYEE NUMBER – TRY AGAIN.".
GO TO QUERY–LOOP.

LOGON.
MOVE "SCOTT" TO USERNAME–ARR.
MOVE 5 TO USERNAME–LEN.
MOVE "TIGER" TO PASSWD–ARR.
MOVE 5 TO PASSWD–LEN.
EXEC SQL
  CONNECT :USERNAME IDENTIFIED BY :PASSWD
END-EXEC.
DISPLAY "".
DISPLAY "CONNECTED TO ORACLE AS USER: ", USERNAME–ARR.
DISPLAY–INFO.
DISPLAY "".
DISPLAY "EMPLOYEE    SALARY    COMMISSION".
DISPLAY "––––––––    ––––––    ––––––––––".
MOVE EMP–NAME–ARR TO D–EMP–NAME.
MOVE SALARY TO D–SALARY.
IF COMM–IND = –1
  DISPLAY D–EMP–NAME, D–SALARY, "          NULL"
ELSE
  MOVE COMMISSION TO D–COMMISSION
  DISPLAY D–EMP–NAME, D–SALARY, "      ", D–COMMISSION
END-IF.

SIGN–OFF.
DISPLAY "".
DISPLAY "TOTAL NUMBER QUERIED WAS ", D–TOTAL–QUERIED, ".".
DISPLAY "".
DISPLAY "HAVE A GOOD DAY.".
DISPLAY "".
EXEC SQL
  COMMIT WORK RELEASE
END-EXEC.
STOP RUN.
SQL-ERROR.
EXEC SQL
   WHENEVER SQLERROR CONTINUE
END-EXEC.
DISPLAY " ".
DISPLAY "ORACLE ERROR DETECTED:".
DISPLAY " ".
DISPLAY SQLERRMC.
EXEC SQL
   ROLLBACK WORK RELEASE
END-EXEC.
STOP RUN.
Sample Program 2: Cursor Operations

This program logs on to Oracle, declares and opens a cursor, fetches the names, salaries, and commissions of all salespeople, displays the results, then closes the cursor.

IDENTIFICATION DIVISION.
PROGRAM-ID. CURSOR-OPS.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01  USERNAME          PIC X(10) VARYING.
  01  PASSWD            PIC X(10) VARYING.
  01  EMP-REC-VARS.
     05  EMP-NAME      PIC X(10) VARYING.
     05  SALARY        PIC S9(6)V99
        DISPLAY SIGN LEADING SEPARATE.
     05  COMMISSION    PIC S9(6)V99
        DISPLAY SIGN LEADING SEPARATE.
EXEC SQL VAR SALARY IS DISPLAY(8,2) END-EXEC.
EXEC SQL VAR COMMISSION IS DISPLAY(8,2) END-EXEC.
EXEC SQL END DECLARE SECTION END-EXEC.

EXEC SQL INCLUDE SQLCA END-EXEC.

  01  DISPLAY-VARIABLES.
     05  D-EMP-NAME    PIC X(10).
     05  D-SALARY      PIC Z(4)9.99.
     05  D-COMMISSION  PIC Z(4)9.99.

PROCEDURE DIVISION.
BEGIN-PGM.
EXEC SQL
  WHENEVER SQLERROR DO PERFORM SQL-ERROR
END-EXEC.
PERFORM LOGON.
EXEC SQL
  DECLARE SALESPEOPLE CURSOR FOR
  SELECT ENAME, SAL, COMM FROM EMP
  WHERE JOB LIKE 'SALES%'
END-EXEC.
EXEC SQL
  OPEN SALESPEOPLE
END-EXEC.
DISPLAY "SALESPERSON   SALARY   COMMISSION".
DISPLAY "–––––––––––   ––––––   ––––––––––".

FETCH-LOOP.
  EXEC SQL
    WHENEVER NOT FOUND DO PERFORM SIGN-OFF
  END-EXEC.
  EXEC SQL
    FETCH SALESPEOPLE
    INTO :EMP-NAME, :SALARY, :COMMISSION
  END-EXEC.
  MOVE EMP-NAME-ARR TO D-EMP-NAME.
  MOVE SALARY TO D-SALARY.
  MOVE COMMISSION TO D-COMMISSION.
  DISPLAY D-EMP-NAME, " ", D-SALARY, " ", D-COMMISSION.
  MOVE SPACES TO EMP-NAME-ARR.
  GO TO FETCH-LOOP.

LOGON.
  MOVE "SCOTT" TO USERNAME-ARR.
  MOVE 5 TO USERNAME-LEN.
  MOVE "TIGER" TO PASSWD-ARR.
  MOVE 5 TO PASSWD-LEN.
  EXEC SQL
    CONNECT :USERNAME IDENTIFIED BY :PASSWD
  END-EXEC.
  DISPLAY " ".
  DISPLAY "CONNECTED TO ORACLE AS USER: ", USERNAME-ARR.
  DISPLAY " ".

SIGN-OFF.
EXEC SQL
CLOSE SALESPEOPLE
END-EXEC.
DISPLAY " ".
DISPLAY "HAVE A GOOD DAY.".
DISPLAY " ".
EXEC SQL
COMMIT WORK RELEASE
END-EXEC.
STOP RUN.

SQL-ERROR.
EXEC SQL
WHENEVER SQLERROR CONTINUE
END-EXEC.
DISPLAY " ".
DISPLAY "ORACLE ERROR DETECTED:".
DISPLAY " ".
DISPLAY SQLERRMC.
EXEC SQL
ROLLBACK WORK RELEASE
END-EXEC.
STOP RUN.
Sample Program 3: Fetching in Batches

This program logs on to Oracle, declares and opens a cursor, fetches in batches using host tables, and prints the results using the PRINT-IT paragraph.

IDENTIFICATION DIVISION.
PROGRAM-ID. HOST-TABLES.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.

EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01  USERNAME          PIC X(15) VARYING.
  01  PASSWD            PIC X(15) VARYING.
  01  EMP-REC-TABLES.
      05  EMP-NUMBER OCCURS 5 TIMES PIC S9(4) COMP.
      05  EMP-NAME      OCCURS 5 TIMES PIC X(10) VARYING.
      05  SALARY        OCCURS 5 TIMES PIC S9(6)V99
                        DISPLAY SIGN LEADING SEPARATE.
  EXEC SQL VAR SALARY IS DISPLAY(8,2) END-EXEC.
EXEC SQL END DECLARE SECTION END-EXEC.

EXEC SQL INCLUDE SQLCA END-EXEC.
  01  NUM-RET           PIC S9(9) COMP VALUE ZERO.
  01  PRINT-NUM         PIC S9(9) COMP VALUE ZERO.
  01  COUNTER           PIC S9(9) COMP.
  01  DISPLAY-VARIABLES.
      05  D-EMP-NAME    PIC X(10).
      05  D-EMP-NUMBER  PIC 9(4).
      05  D-SALARY      PIC Z(4)9.99.
PROCEDURE DIVISION.

BEGIN-PGM.
  EXEC SQL
      WHENEVER SQLERROR DO PERFORM SQL-ERROR
END-EXEC.
  PERFORM LOGON.
  EXEC SQL
      DECLARE C1 CURSOR FOR
      SELECT EMPNO, SAL, ENAME FROM EMP
END-EXEC.
  EXEC SQL
      OPEN C1
END-EXEC.
FETCH-LOOP.
  EXEC SQL
   WHENEVER NOT FOUND DO PERFORM SIGN-OFF
END-EXEC.
EXEC SQL
   FETCH C1 INTO :EMP-NUMBER, :SALARY, :EMP-NAME
END-EXEC.
SUBTRACT NUM-RET FROM SQLERRD(3) GIVING PRINT-NUM.
PERFORM PRINT-IT.
MOVE SQLERRD(3) TO NUM-RET.
GO TO FETCH-LOOP.

LOGON.
MOVE "SCOTT" TO USERNAME-ARR.
MOVE 5 TO USERNAME-LEN.
MOVE "TIGER" TO PASSWD-ARR.
MOVE 5 TO PASSWD-LEN.
EXEC SQL
   CONNECT :USERNAME IDENTIFIED BY :PASSWD
END-EXEC.
DISPLAY " ".
DISPLAY "CONNECTED TO ORACLE AS USER: ", USERNAME-ARR.

PRINT-IT.
DISPLAY " ".
DISPLAY "EMPLOYEE NUMBER   SALARY   EMPLOYEE NAME".
DISPLAY "-----------------   ------   ----------------".
PERFORM PRINT-ROWS
   VARYING COUNTER FROM 1 BY 1 UNTIL COUNTER > PRINT-NUM.

PRINT-ROWS.
MOVE EMP-NUMBER(COUNTER) TO D-EMP-NUMBER.
MOVE SALARY(COUNTER) TO D-SALARY.
MOVE SPACES TO EMP-NAME-ARR IN EMP-NAME(COUNTER).

SIGN-OFF.
SUBTRACT NUM-RET FROM SQLERRD(3) GIVING PRINT-NUM.
IF (PRINT-NUM > 0) PERFORM PRINT-IT.
EXEC SQL CLOSE C1 END-EXEC.
EXEC SQL COMMIT WORK RELEASE END-EXEC.
DISPLAY " ".
DISPLAY "HAVE A GOOD DAY.".
DISPLAY " ".
STOP RUN.
SQL-ERROR.
   EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
   DISPLAY ".
   DISPLAY "ORACLE ERROR DETECTED:"
   DISPLAY " ".
   DISPLAY SQLERRMC.
   EXEC SQL
      ROLLBACK WORK RELEASE
END-EXEC.
STOP RUN.
Sample Program 4: Datatype Equivalencing

After connecting to Oracle, this program creates a database table named IMAGE in the SCOTT account, then simulates the insertion of bitmap images of employee numbers into the table. Datatype equivalencing lets the program use the Oracle external datatype LONG RAW to represent the images. Later, when the user enters an employee number, the number’s “bitmap” is selected from the IMAGE table and pseudo-displayed on the terminal screen.

IDENTIFICATION DIVISION.
PROGRAM-ID. DTY-EQUIV.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
  EXEC SQL BEGIN DECLARE SECTION END-EXEC.
    01 USERNAME PIC X(10) VARYING.
    01 PASSWD PIC X(10) VARYING.
    01 EMP-REC-VARS.
      05 EMP-NUMBER PIC S9(4) COMP.
      05 EMP-NAME PIC X(10) VARYING.
      05 SALARY PIC S9(6)V99 DISPLAY SIGN LEADING SEPARATE.
      05 COMMISSION PIC S9(6)V99 DISPLAY SIGN LEADING SEPARATE.
      05 COMM-IND PIC S9(4) COMP.
  EXEC SQL VAR SALARY IS DISPLAY(8,2) END-EXEC.
  EXEC SQL VAR COMMISSION IS DISPLAY(8,2) END-EXEC.
    01 BUFFER-VAR.
      05 BUFFER PIC X(8192).
  EXEC SQL VAR BUFFER IS LONG RAW END-EXEC.
    01 SELECTION PIC S9(4) COMP.
  EXEC SQL END DECLARE SECTION END-EXEC.
  EXEC SQL INCLUDE SQLCA END-EXEC.

  01 DISPLAY-VARIABLES.
    05 D-EMP-NAME PIC X(10).
    05 D-SALARY PIC Z(4)9.99.
    05 D-COMMISSION PIC Z(4)9.99.
    01 REPLY PIC X(10).
    01 INDX PIC S9(9) COMP.
    01 PRT-QUOT PIC S9(9) COMP.
    01 PRT-MOD PIC S9(9) COMP.
PROCEDURE DIVISION.

BEGIN-PGM.
  EXEC SQL
    WHENEVER SQLERROR DO PERFORM SQL-ERROR
  END-EXEC.
  PERFORM LOGON.
  DISPLAY "OK TO DROP THE IMAGE TABLE? (Y/N) "
    WITH NO ADVANCING.
  ACCEPT REPLY.
  IF (REPLY NOT = "Y") AND (REPLY NOT = "Y")
    PERFORM SIGN-OFF.
  EXEC SQL
    WHENEVER SQLERROR CONTINUE
  END-EXEC.
  EXEC SQL
    DROP TABLE IMAGE
  END-EXEC.
  IF (SQLCODE = 0) DISPLAY
    "TABLE IMAGE DROPPED - CREATING NEW TABLE."
  ELSE IF (SQLCODE = -942) DISPLAY
    "TABLE IMAGE DOES NOT EXIST - CREATING NEW TABLE."
  ELSE PERFORM SQL-ERROR.
  EXEC SQL
    WHENEVER SQLERROR DO PERFORM SQL-ERROR
  END-EXEC.
  EXEC SQL
    CREATE TABLE IMAGE
      (EMPNO NUMBER(4) NOT NULL, BITMAP LONG RAW)
  END-EXEC.
  EXEC SQL
    DECLARE EMPCUR CURSOR FOR
      SELECT EMPNO, ENAME FROM EMP
  END-EXEC.
  EXEC SQL
    OPEN EMPCUR
  END-EXEC.
  DISPLAY " ".
  DISPLAY "INSERTING BITMAPS INTO IMAGE FOR ALL EMPLOYEES...".
  DISPLAY " ".

INSERT-LOOP.
EXEC SQL
  WHENEVER NOT FOUND GOTO NOT-FOUND
END-EXEC.
EXEC SQL
  FETCH EMPCUR INTO :EMP-NUMBER, :EMP-NAME
END-EXEC.
MOVE EMP-NAME-ARR TO D-EMP-NAME.
DISPLAY "EMPLOYEE ", D-EMP-NAME WITH NO ADVANCING.
PERFORM GET-IMAGE.
EXEC SQL
  INSERT INTO IMAGE VALUES (:EMP-NUMBER, :BUFFER)
END-EXEC.
DISPLAY ", IS DONE!".
MOVE SPACES TO EMP-NAME-ARR.
GO TO INSERT-LOOP.

NOT-FOUND.
EXEC SQL
  CLOSE EMPCUR
END-EXEC.
EXEC SQL
  COMMIT WORK
END-EXEC.
DISPLAY ""
DISPLAY "DONE INSERTING BITMAPS. NEXT, DISPLAY SOME.".

DISP-LOOP.
MOVE 0 TO SELECTION.
DISPLAY ""
DISPLAY "ENTER EMP NUMBER (0 TO QUIT): " WITH NO ADVANCING.
ACCEPT SELECTION.
IF (SELECTION = 0) PERFORM SIGN-OFF.
EXEC SQL
  WHENEVER NOT FOUND GOTO NO-EMP
END-EXEC.
EXEC SQL
  SELECT EMP.EMPNO, ENAME, SALARY, COMM, BITMAP
  INTO :EMP-NUMBER, :EMP-NAME, :SALARY,
  :COMMISSION:COMM-IND, :BUFFER
  FROM EMP, IMAGE
  WHERE EMP.EMPNO = :SELECTION AND EMP.EMPNO = IMAGE.EMPNO
END-EXEC.
DISPLAY " ".
PERFORM SHOW-IMAGE.
MOVE EMP-NAME-ARR TO D-EMP-NAME.
MOVE SALARY TO D-SALARY.
MOVE COMMISSION TO D-COMMISSION.
DISPLAY "EMPLOYEE ", D-EMP-NAME, " HAS SALARY ", D-SALARY
WITH NO ADVANCING.
IF COMM-IND = -1
   DISPLAY " AND NO COMMISSION."
ELSE
   DISPLAY " AND COMMISSION ", D-COMMISSION
END-IF.
MOVE SPACES TO EMP-NAME-ARR.
GO TO DISP-LOOP.

NO-EMP.
DISPLAY "NOT A VALID EMPLOYEE NUMBER – TRY AGAIN.".
GO TO DISP-LOOP.

LOGON.
MOVE "SCOTT" TO USERNAME-ARR.
MOVE 5 TO USERNAME-LEN.
MOVE "TIGER" TO PASSWD-ARR.
MOVE 5 TO PASSWD-LEN.
EXEC SQL
   CONNECT :USERNAME IDENTIFIED BY :PASSWD
END-EXEC.
DISPLAY " ".
DISPLAY "CONNECTED TO ORACLE AS USER: ", USERNAME-ARR.
DISPLAY " ".

GET-IMAGE.
PERFORM MOVE-IMAGE
   VARYING INDX FROM 1 BY 1 UNTIL INDX > 8192.

MOVE-IMAGE.
STRING "**" DELIMITED BY SIZE INTO BUFFER WITH POINTER INDX.
DIVIDE 256 INTO INDX GIVING PRT-QUOT REMAINDER PRT-MOD.
IF (PRT-MOD = 0) DISPLAY "." WITH NO ADVANCING.

SHOW-IMAGE.
PERFORM VARYING INDX FROM 1 BY 1 UNTIL INDX > 10
DISPLAY " ............................."
END-PERFORM.
DISPLAY " ".
SIGN-OFF.
   DISPLAY " ".
   DISPLAY "HAVE A GOOD DAY.".
   DISPLAY " ".
EXEC SQL
   COMMIT WORK RELEASE
END-EXEC.
STOP RUN.

SQL-ERROR.
EXEC SQL
   WHENEVER SQLERROR CONTINUE
END-EXEC.
   DISPLAY " ".
   DISPLAY "ORACLE ERROR DETECTED:".
   DISPLAY " ".
   DISPLAY SQLERRMC.
   EXEC SQL
      ROLLBACK WORK RELEASE
END-EXEC.
STOP RUN.
Sample Program 5: Oracle Forms User Exit

This user exit concatenates form fields. To call the user exit from an Oracle Forms trigger, use the syntax

```cobol
'user_exit>('CONCAT <field1>, <field2>, ..., <result_field>');
```

where `user_exit` is a packaged procedure supplied with Oracle Forms and CONCAT is the name of the user exit. A sample CONCAT form invokes the user exit. For more information about Oracle Forms user exits, see Chapter 11 of the Programmer’s Guide to the Oracle Precompilers.

```cobol
IDENTIFICATION DIVISION.
PROGRAM-ID. CONCAT.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.

EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01 FIELD-NAME             PIC X(80)  VARYING.
  01 FIELD-VALUE            PIC X(80)  VARYING.
  01 RESULT                 PIC X(800) VARYING.
EXEC SQL END DECLARE SECTION END-EXEC.
EXEC SQL INCLUDE SQLCA END-EXEC.

01 EXIT-MESSAGE           PIC X(80).
01 EXIT-MESSAGE-LEN       PIC S9(9) COMP.
01 RTN-CODE               PIC S9(9) COMP.
  ?7 INDEX                PIC S9(4) COMP.
01 DONE-FLAG              PIC X.
  88 DONE                VALUE 'Y'.
01 PTR                    PIC S9(4) COMP.
01 WS-CMD-LINE.
  05 WS-CMD-LINE-Y        PIC X(80).
  05 WS-CMD-LINE-X        REDEFINES WS-CMD-LINE-Y
01 WS-FIELD-NAME-AREA.
  05 WS-FIELD-NAME        PIC X(80).
  05 WS-FIELD-NAME-X      REDEFINES WS-FIELD-NAME
                         PIC X OCCURS 80.
  05 WS-FIELD-NAME-LEN    PIC S9(4) COMP.

LINKAGE SECTION.
  01 CMD-LINE             PIC X(80).
  01 CMD-LINE-LEN         PIC S9(9) COMP.
  01 ERR-MSG              PIC X(80).
  01 ERR-MSG-LEN          PIC S9(9) COMP.
  01 IN-QUERY             PIC S9(9) COMP.
  01 RETURN-VALUE         PIC S9(9) COMP.
```
PROCEDURE DIVISION USING CMD-LINE, CMD-LINE-LEN,
   ERR-MSG, ERR-MSG-LEN,
   IN-QUERY, RETURN-VALUE.

MAIN.
   MOVE 1 TO PTR.
   MOVE SPACE TO RESULT-ARR.
   MOVE ZERO TO RESULT-LEN.
   MOVE SPACE TO DONE-FLAG.
   MOVE 7 TO INDX.
   MOVE CMD-LINE TO WS-CMD-LINE-Y.
   PERFORM CMD-LINE-PARSE UNTIL DONE.
   EXEC SQL
       WHENEVER SQLERROR GOTO SQL-ERROR
   END-EXEC.
   MOVE WS-FIELD-NAME TO FIELD-NAME-ARR.
   MOVE WS-FIELD-NAME-LEN TO FIELD-NAME-LEN.
   EXEC IAF
       PUT :FIELD-NAME VALUES(:RESULT)
   END-EXEC.
   MOVE SQL-IAPXIT-SUCCESS TO RTN-CODE.
   EXIT PROGRAM GIVING RTN-CODE.

CMD-LINE-PARSE.
   MOVE ZERO TO WS-FIELD-NAME-LEN.
   MOVE SPACES TO WS-FIELD-NAME.
   MOVE SPACES TO FIELD-NAME-ARR.
   MOVE ZERO TO FIELD-NAME-LEN.
   PERFORM GET-FIELD-NAME
       UNTIL WS-CMD-LINE-X(INDX) = ',' OR DONE.
   IF WS-CMD-LINE-X(INDX) = ','
      MOVE SPACES TO FIELD-NAME-ARR
      MOVE WS-FIELD-NAME TO FIELD-NAME-ARR
      MOVE WS-FIELD-NAME-LEN TO FIELD-NAME-LEN
      MOVE SPACES TO FIELD-VALUE-ARR
      EXEC IAF
          GET :FIELD-NAME INTO :FIELD-VALUE
      END-EXEC
      STRING FIELD-VALUE-ARR
          DELIMITED BY SPACE
          INTO RESULT-ARR
          WITH POINTER PTR
      ADD FIELD-VALUE-LEN TO RESULT-LEN
      ADD 1 TO INDX.
GET-FIELD-NAME.
  IF WS-CMD-LINE-X(INDX) NOT EQUAL SPACE
    ADD 1 TO WS-FIELD-NAME-LEN
    MOVE WS-CMD-LINE-X(INDX) TO
  END-IF
  ADD 1 TO INDX.
  IF INDX > CMD-LINE-LEN MOVE 'Y' TO DONE-FLAG.

SQL-ERROR.
  EXEC SQL
    WHENEVER SQLERROR CONTINUE
  END-EXEC.
  MOVE SQLERRMC TO EXIT-MESSAGE.
  MOVE SQLERRML TO EXIT-MESSAGE-LEN.
  CALL "SQLIEM" USING EXIT-MESSAGE EXIT-MESSAGE-LEN.
  MOVE SQL-IAPXIT-FAILURE TO RTN-CODE.
  EXIT PROGRAM.
Sample Program 6: 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.

IDENTIFICATION DIVISION.
PROGRAM-ID. DYNSQL1.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.

* INCLUDE THE ORACLE COMMUNICATIONS AREA, A STRUCTURE
* THROUGH WHICH ORACLE MAKES ADDITIONAL RUNTIME STATUS
* INFORMATION AVAILABLE TO THE PROGRAM.
EXEC SQL INCLUDE SQLCA END-EXEC.

* INCLUDE THE ORACLE COMMUNICATIONS AREA, A STRUCTURE
* THROUGH WHICH ORACLE MAKES ADDITIONAL RUNTIME STATUS
* INFORMATION AVAILABLE TO THE PROGRAM.
EXEC SQL INCLUDE ORACA END-EXEC.

* THE OPTION ORACA=YES MUST BE SPECIFIED TO ENABLE USE OF
* THE ORACA.
EXEC ORACLE OPTION (ORACA=YES) END-EXEC.

* THE RELEASE_CURSOR=YES OPTION INSTRUCTS PRO*C OBOL TO
* RELEASE IMPLICIT CURSORS ASSOCIATED WITH EMBEDDED SQL
* STATEMENTS. THIS ENSURES THAT ORACLE DOES NOT KEEP
* PARSE LOCKS ON TABLES, SO THAT SUBSEQUENT DATA
* MANIPULATION OPERATIONS ON THOSE TABLES DO NOT RESULT
* IN PARSE-LOCK ERRORS.
EXEC ORACLE OPTION (RELEASE_CURSOR=YES) END-EXEC.

* ALL HOST VARIABLES USED IN EMBEDDED SQL MUST APPEAR IN
* THE DECLARE SECTION.
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01 USERNAME PIC X(10) VALUE "SCOTT".
  01 PASSWD PIC X(10) VALUE "TIGER".
  01 DYNSTMT PIC X(80) VARYING.
EXEC SQL END DECLARE SECTION END-EXEC.

* DECLARE VARIABLES NEEDED TO DISPLAY COMPUTATIONALS.
  01 ORASLNRD PIC 9(9).
PROCEDURE DIVISION.

MAIN.

* BRANCH TO PARAGRAPH SQLERROR IF AN ORACLE ERROR OCCURS.
  EXEC SQL
    WHENEVER SQLERROR GOTO SQLERROR
  END-EXEC.

* SAVE TEXT OF CURRENT SQL STATEMENT IN THE ORACA IF AN
  ERROR OCCURS.
  MOVE 1 TO ORASTXTF.

* CONNECT TO ORACLE.
  EXEC SQL
    CONNECT :USERNAME IDENTIFIED BY :PASSWD
  END-EXEC.
  DISPLAY " ".
  DISPLAY "CONNECTED TO ORACLE.".
  DISPLAY " ".

* EXECUTE A STRING LITERAL TO CREATE THE TABLE. HERE,
  YOU GENERALLY USE A STRING VARIABLE INSTEAD OF A
  LITERAL, AS IS DONE LATER IN THIS PROGRAM. BUT, YOU
  CAN USE A LITERAL IF YOU WISH.
  DISPLAY "CREATE TABLE DYN1 (COL1 CHAR(4))".
  EXEC SQL
    EXECUTE IMMEDIATE "CREATE TABLE DYN1 (COL1 CHAR(4))"
  END-EXEC.

* ASSIGN A SQL STATEMENT TO THE VARYING STRING DYNSTMT.
  SET THE -LEN PART TO THE LENGTH OF THE -ARR PART.
  MOVE "INSERT INTO DYN1 VALUES ('TEST')"
  TO DYNSTMT-ARR.
  MOVE 36 TO DYNSTMT-LEN.
  DISPLAY DYNSTMT-ARR.

* EXECUTE DYNSTMT TO INSERT A ROW. THE SQL STATEMENT IS
  A STRING VARIABLE WHOSE CONTENTS THE PROGRAM MAY
  DETERMINE AT RUN TIME.
  EXEC SQL
    EXECUTE IMMEDIATE :DYNSTMT
  END-EXEC.
* COMMIT THE INSERT.
  EXEC SQL
    COMMIT WORK
END-EXEC.

* CHANGE DYNSTMT AND EXECUTE IT TO DROP THE TABLE.
  MOVE "DROP TABLE DYN1" TO DYNSTMT-ARR.
  MOVE 19 TO DYNSTMT-LEN.
  DISPLAY DYNSTMT-ARR.
  EXEC SQL
    EXECUTE IMMEDIATE :DYNSTMT
END-EXEC.

* COMMIT ANY PENDING CHANGES AND DISCONNECT FROM ORACLE.
  EXEC SQL
    COMMIT RELEASE
END-EXEC.
  DISPLAY "".
  DISPLAY "HAVE A GOOD DAY!".
  DISPLAY "".
  STOP RUN.

SQLERROR.
* ORACLE ERROR HANDLER. PRINT DIAGNOSTIC TEXT CONTAINING
* ERROR MESSAGE, CURRENT SQL STATEMENT, AND LOCATION OF
* ERROR.
  DISPLAY SQLERRMC.
  DISPLAY "IN ", ORASTXTC.
  MOVE ORASLNR TO ORASLNRD.
  DISPLAY "ON LINE ", ORASLNRD, " OF ", ORASFNMC.

* DISABLE ORACLE ERROR CHECKING TO AVOID AN INFINITE LOOP
* SHOULD ANOTHER ERROR OCCUR WITHIN THIS PARAGRAPH.
  EXEC SQL
    WHENEVER SQLERROR CONTINUE
END-EXEC.

* ROLL BACK ANY PENDING CHANGES AND DISCONNECT FROM
* ORACLE.
  EXEC SQL
    ROLLBACK RELEASE
END-EXEC.
  STOP RUN.
Sample Program 7: Dynamic SQL Method 2

This program uses dynamic SQL Method 2 to insert two rows into the EMP table, then delete them.

IDENTIFICATION DIVISION.
PROGRAM-ID. DYNSQL2.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
*   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 END–EXEC.

*   INCLUDE THE ORACLE COMMUNICATIONS AREA, A STRUCTURE THROUGH
*   WHICH ORACLE MAKES ADDITIONAL RUNTIME STATUS INFORMATION
*   AVAILABLE TO THE PROGRAM.
   EXEC SQL INCLUDE ORACA END–EXEC.

*   THE OPTION ORACA=YES MUST BE SPECIFIED TO ENABLE USE OF
*   THE ORACA.
   EXEC ORACLE OPTION (ORACA=YES) END–EXEC.

*   ALL HOST VARIABLES USED IN EMBEDDED SQL MUST APPEAR IN THE
*   DECLARE SECTION.
   EXEC SQL BEGIN DECLARE SECTION END–EXEC.
      01 USERNAME PIC X(10) VALUE "SCOTT".
      01 PASSWD PIC X(10) VALUE "TIGER".
      01 DYNSTMT PIC X(80) VARYING.
      01 EMPNO PIC S9(4) COMPUTATIONAL VALUE 1234.
      01 DEPTNO1 PIC S9(4) COMPUTATIONAL VALUE 97.
      01 DEPTNO2 PIC S9(4) COMPUTATIONAL VALUE 99.
   EXEC SQL END DECLARE SECTION END–EXEC.

*   DECLARE VARIABLES NEEDED TO DISPLAY COMPUTATIONALS.
      01 EMPNOD PIC 9(4).
      01 DEPTNO1D PIC 9(2).
      01 DEPTNO2D PIC 9(2).
      01 ORASLNRD PIC 9(9).
PROCEDURE DIVISION.

MAIN.
* BRANCH TO PARAGRAPH SQLERROR IF AN ORACLE ERROR OCCURS.
  EXEC SQL
    WHENEVER SQLERROR GOTO SQLERROR
END-EXEC.
* SAVE TEXT OF CURRENT SQL STATEMENT IN THE ORACA IF AN ERROR
  OCCURS.
  MOVE 1 TO ORASTXTF.
* CONNECT TO ORACLE.
  EXEC SQL
    CONNECT :USERNAME IDENTIFIED BY :PASSWD
END-EXEC.
  DISPLAY " ".
  DISPLAY "CONNECTED TO ORACLE.".
  DISPLAY " ".
* ASSIGN A SQL STATEMENT TO THE VARYING STRING 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.
  MOVE "INSERT INTO EMP (EMPNO, DEPTNO) VALUES (:V1, :V2)"
    TO DYNSTMT–ARR.
  MOVE 49 TO DYNSTMT–LEN.
* DISPLAY THE SQL STATEMENT AND ITS CURRENT INPUT HOST
  VARIABLES.
  DISPLAY DYNSTMT–ARR.
  MOVE EMPNO TO EMPNOD.
  MOVE DEPTNO1 TO DEPTNO1D.
  DISPLAY " V1 = ", EMPNOD, " V2 = ", DEPTNO1D.
* 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 MAY BE PREPARED MORE THAN ONCE,
  Optionally FROM A DIFFERENT STRING VARIABLE.
  EXEC SQL
    PREPARE S FROM :DYNSTMT
END-EXEC.
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 may be omitted only if the statement contains no placeholders.

A single prepared statement may be executed more than once, optionally using different input host variables.

EXEC SQL
   EXECUTE S USING :EMPNO, :DEPTNO1
END-EXEC.

Increment EMPNO and display new input host variables.
ADD 1 TO EMPNO.
MOVE EMPNO TO EMPNOD.
MOVE DEPTNO2 TO DEPTNO2D.
DISPLAY "    V1 =", EMPNOD, "    V2 =", DEPTNO2D.

REEXECUTE S to insert the new value of EMPNO and a different input host variable, DEPTNO2. A reprepare is not necessary.
EXEC SQL
   EXECUTE S USING :EMPNO, :DEPTNO2
END-EXEC.
* ASSIGN A NEW VALUE TO DYNSTMT.
  MOVE
    "DELETE FROM EMP WHERE DEPTNO = :V1 OR DEPTNO = :V2"
  TO DYNSTMT-ARR.
  MOVE 50 TO DYNSTMT-LEN.

* DISPLAY THE NEW SQL STATEMENT AND ITS CURRENT INPUT HOST
  VARIABLES.
  DISPLAY DYNSTMT-ARR.
  DISPLAY "    V1 = ", DEPTNO1D, "      V2 = ", DEPTNO2D.
  REPREPARE S FROM THE NEW DYNSTMT.
  EXEC SQL
    PREPARE S FROM :DYNSTMT
  END-EXEC.

* EXECUTE THE NEW S TO DELETE THE TWO ROWS PREVIOUSLY INSERTED.
  EXEC SQL
    EXECUTE S USING :DEPTNO1, :DEPTNO2
  END-EXEC.

* COMMIT ANY PENDING CHANGES AND DISCONNECT FROM ORACLE.
  EXEC SQL
    COMMIT RELEASE
  END-EXEC.
  DISPLAY " ".
  DISPLAY "HAVE A GOOD DAY!".
  DISPLAY " ".
  STOP RUN.

SQLERROR.
* ORACLE ERROR HANDLER. PRINT DIAGNOSTIC TEXT CONTAINING ERROR
  MESSAGE, CURRENT SQL STATEMENT, AND LOCATION OF ERROR.
  DISPLAY SQLERRMC.
  DISPLAY "IN ", ORASTXTC.
  MOVE ORASLNR TO ORASLNRD.
  DISPLAY "ON LINE ", ORASLNRD, " OF ", ORASFNMC.

* DISABLE ORACLE ERROR CHECKING TO AVOID AN INFINITE LOOP
  SHOULD ANOTHER ERROR OCCUR WITHIN THIS PARAGRAPH.
  EXEC SQL
    WHENEVER SQLERROR CONTINUE
  END-EXEC.

* ROLL BACK ANY PENDING CHANGES AND DISCONNECT FROM ORACLE.
  EXEC SQL
    ROLLBACK RELEASE
  END-EXEC.
  STOP RUN.
Sample Program 8: 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.

IDENTIFICATION DIVISION.
PROGRAM-ID. DYNSQL3.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.

* 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 END-EXEC.

* INCLUDE THE ORACLE COMMUNICATIONS AREA, A STRUCTURE
* THROUGH WHICH ORACLE MAKES ADDITIONAL RUNTIME STATUS
* INFORMATION AVAILABLE TO THE PROGRAM.
EXEC SQL INCLUDE ORACA END-EXEC.

* THE ORACA=YES OPTION MUST BE SPECIFIED TO ENABLE USE OF
* THE ORACA.
EXEC ORACLE OPTION (ORACA=YES) END-EXEC.

* ALL HOST VARIABLES USED IN EMBEDDED SQL MUST APPEAR IN
* THE DECLARE SECTION.
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
   01 USERNAME PIC X(10) VALUE "SCOTT".
   01 PASSWD PIC X(10) VALUE "TIGER".
   01 DYNSTMT PIC X(80) VARYING.
   01 ENAME PIC X(10).
   01 DEPTNO PIC 9(2).
EXEC SQL END DECLARE SECTION END-EXEC.

* DECLARE VARIABLES NEEDED TO DISPLAY COMPUTATIONALS.
   01 DEPTNOD PIC 9(2).
   01 ENAMED PIC X(10).
   01 SQLERRD3 PIC 9(2).
   01 ORASLRD PIC 9(4).
PROCEDURE DIVISION.

MAIN.
* BRANCH TO PARAGRAPH SQLERROR IF AN ORACLE ERROR OCCURS.
  EXEC SQL
  WHENEVER SQLERROR GO TO SQLERROR
END-EXEC.

* SAVE TEXT OF CURRENT SQL STATEMENT IN THE ORACA IF AN
* ERROR OCCURS.
  MOVE 1 TO ORASTXTF.

* CONNECT TO ORACLE.
  EXEC SQL
    CONNECT :USERNAME IDENTIFIED BY :PASSWD
END-EXEC.
  DISPLAY " ".
  DISPLAY "CONNECTED TO ORACLE.".
  DISPLAY " ".

* ASSIGN A SQL QUERY TO THE VARYING STRING DYNSTMT. BOTH
* THE ARRAY AND THE LENGTH PARTS MUST BE SET PROPERLY.
* NOTE THAT THE STATEMENT CONTAINS ONE HOST VARIABLE
* PLACEHOLDER, V1, FOR WHICH AN ACTUAL INPUT HOST
* VARIABLE MUST BE SUPPLIED AT OPEN TIME.
  MOVE "SELECT ENAME FROM EMP WHERE DEPTNO = :V1"
  TO DYNSTMT–ARR.
  MOVE 40 TO DYNSTMT–LEN.

* DISPLAY THE SQL STATEMENT AND ITS CURRENT INPUT HOST
* VARIABLE.
  DISPLAY DYNSTMT–ARR.
  MOVE DEPTNO TO DEPTNOD.
  DISPLAY " V1 = ", DEPTNOD.
  DISPLAY ".
  DISPLAY "EMPLOYEE".
  DISPLAY "--------".

* THE PREPARE STATEMENT ASSOCIATES A STATEMENT NAME WITH
* A STRING CONTAINING A SELECT STATEMENT. THE STATEMENT
* NAME, WHICH MUST BE UNIQUE, IS A SQL IDENTIFIER, NOT A
* HOST VARIABLE, AND SO DOES NOT APPEAR IN THE DECLARE
* SECTION.
  EXEC SQL
    PREPARE S FROM :DYNSTMT
END-EXEC.
THE DECLARE STATEMENT ASSOCIATES A CURSOR WITH A PREPARED STATEMENT. THE CURSOR NAME, LIKE THE STATEMENT NAME, DOES NOT APPEAR IN THE DECLARE SECTION.

EXEC SQL
  DECLARE C CURSOR FOR S
END-EXEC.


A SINGLE DECLARED CURSOR MAY BE OPENED MORE THAN ONCE, Optionally using different input host variables.

EXEC SQL
  OPEN C USING :DEPTNO
END-EXEC.

BRANCH TO PARAGRAPH NOTFOUND WHEN ALL ROWS HAVE BEEN RETRIEVED.
EXEC SQL
  WHENEVER NOT FOUND GO TO NOTFOUND
END-EXEC.

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 ARE NOT RETURNED. SPECIFYING MORE OUTPUT HOST VARIABLES THAN SELECT-LIST FIELDS RESULTS IN AN ORACLE ERROR.

EXEC SQL
  FETCH C INTO :ENAME
END-EXEC.

MOVE ENAME TO ENAMED.
DISPLAY ENAMED.

LOOP UNTIL NOT FOUND CONDITION IS DETECTED.
GO TO GETROWS.
NOTFOUND.
MOVE SQLERRD(3) TO SQLERRD3.
DISPLAY " ".
DISPLAY "QUERY RETURNED ", SQLERRD3, " ROW(S).".

* THE CLOSE STATEMENT RELEASES RESOURCES ASSOCIATED WITH
* THE CURSOR.
EXEC SQL
  CLOSE C
END-EXEC.

* COMMIT ANY PENDING CHANGES AND DISCONNECT FROM ORACLE.
EXEC SQL
  COMMIT RELEASE
END-EXEC.
DISPLAY " ".
DISPLAY "HAVE A GOOD DAY!".
DISPLAY " ".
STOP RUN.

SQLERROR.
* ORACLE ERROR HANDLER. PRINT DIAGNOSTIC TEXT CONTAINING
* ERROR MESSAGE, CURRENT SQL STATEMENT, AND LOCATION OF
* ERROR.
DISPLAY SQLERRMC.
DISPLAY "IN ", ORASTXTC.
MOVE ORASLNR TO ORASLNRD.
DISPLAY "ON LINE ", ORASLNRD, " OF ", ORASFNC.

* DISABLE ORACLE ERROR CHECKING TO AVOID AN INFINITE LOOP
* SHOULD ANOTHER ERROR OCCUR WITHIN THIS PARAGRAPH.
EXEC SQL
  WHENEVER SQLERROR CONTINUE
END-EXEC.

* RELEASE RESOURCES ASSOCIATED WITH THE CURSOR.
EXEC SQL
  CLOSE C
END-EXEC.

* ROLL BACK ANY PENDING CHANGES AND DISCONNECT FROM
* ORACLE.
EXEC SQL
  ROLLBACK RELEASE
END-EXEC.
STOP RUN.
Sample Program 9: Calling a Stored Procedure

Before trying the sample program, you must create a PL/SQL package named calldemo, by running a script named CALLDEMO.SQL, which is supplied with Pro*COBOL and shown below. The script can be found in the Pro*COBOL demo library. Check your system–specific Oracle documentation for exact spelling of the script.

```sql
CREATE OR REPLACE PACKAGE calldemo AS

    TYPE name_array IS TABLE OF emp.ename%type
      INDEX BY BINARY_INTEGER;
    TYPE job_array IS TABLE OF emp.job%type
      INDEX BY BINARY_INTEGER;
    TYPE sal_array IS TABLE OF emp.sal%type
      INDEX BY BINARY_INTEGER;

    PROCEDURE get_employees(
      dept_number IN     number,    -- department to query
      batch_size  IN     INTEGER,   -- rows at a time
      found       IN OUT INTEGER,   -- rows actually returned
      done_fetch  OUT    INTEGER,   -- all done flag
      emp_name    OUT    name_array,
      job         OUT    job_array,
      sal         OUT    sal_array);

END calldemo;
/

CREATE OR REPLACE PACKAGE BODY calldemo AS

    CURSOR get_emp (dept_number IN number) IS
      SELECT ename, job, sal FROM emp
      WHERE deptno = dept_number;
```
-- Procedure "get_employees" fetches a batch of employee
-- rows (batch size is determined by the client/caller
-- of the procedure). It can be called from other
-- stored procedures or client application programs.
-- The procedure opens the cursor if it is not
-- already open, fetches a batch of rows, and
-- returns the number of rows actually retrieved. At
-- end of fetch, the procedure closes the cursor.

PROCEDURE get_employees(
    dept_number IN     number,
    batch_size  IN     INTEGER,
    found       IN OUT INTEGER,
    done_fetch  OUT    INTEGER,
    emp_name    OUT    name_array,
    job         OUT    job_array,
    sal         OUT    sal_array) IS

BEGIN
    IF NOT get_emp%ISOPEN THEN      -- open the cursor if
        OPEN get_emp(dept_number);  -- not already open
    END IF;

    -- Fetch up to “batch_size” rows into PL/SQL table,
    -- tallying rows found as they are retrieved. When all
    -- rows have been fetched, close the cursor and exit
    -- the loop, returning only the last set of rows found.

    done_fetch := 0;  -- set the done flag FALSE
    found := 0;

    FOR i IN 1..batch_size LOOP
        FETCH get_emp INTO emp_name(i), job(i), sal(i);
        IF get_emp%NOTFOUND THEN    -- if no row was found
            CLOSE get_emp;
            done_fetch := 1;   -- indicate all done
            EXIT;
        ELSE
            found := found + 1;  -- count row
        END IF;
    END LOOP;

END;
/
The following sample program connects to Oracle, prompts the user for
a department number, then calls a PL/SQL procedure named

get_employees, which is stored in package calldemo. 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 tables. When the procedure finishes, row values in
the PL/SQL tables are automatically assigned to the corresponding
elements in the host tables. The program calls the procedure repeatedly,
displaying each batch of employee data, until no more data is found.

IDENTIFICATION DIVISION.
PROGRAM-ID. CALL-STORED-PROC.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.

EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01  USERNAME   PIC X(15) VARYING.
  01  PASSWD    PIC X(15) VARYING.
  01  DEPT–NUM  PIC S9(9) COMP.
  01  EMP–TABLES.
     05  EMP–NAME  OCCURS 10 TIMES PIC X(10).
     05  JOB–TITLE OCCURS 10 TIMES PIC X(10).
     05  SALARY    OCCURS 10 TIMES COMP–2.
  01  DONE–FLAG  PIC S9(9) COMP.
  01  TABLE–SIZE PIC S9(9) COMP VALUE 10.
  01  NUM–RET   PIC S9(9) COMP.
  01  SQLCODE   PIC S9(9) COMP.
EXEC SQL END DECLARE SECTION END-EXEC.

EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01  COUNTER   PIC S9(9) COMP.
  01  DISPLAY–VARIABLES.
     05  D–EMP–NAME PIC X(10).
     05  D–JOB–TITLE PIC X(10).
     05  D–SALARY  PIC Z(5)9.
EXEC SQL INCLUDE SQLCA END-EXEC.
PROCEDURE DIVISION.

BEGIN-PGM.
EXEC SQL
WHENEVER SQLERROR DO PERFORM SQL-ERROR
END-EXEC.
PERFORM LOGON.
PERFORM INIT-TABLES VARYING COUNTER FROM 1 BY 1
UNTIL COUNTER > 10.
PERFORM GET-DEPT-NUM.
PERFORM DISPLAY-HEADER.
MOVE ZERO TO DONE-FLAG.
MOVE ZERO TO NUM-RET.
PERFORM FETCH-BATCH UNTIL DONE-FLAG = 1.
PERFORM LOGOFF.

INIT-TABLES.
MOVE SPACE TO EMP-NAME(COUNTER).
MOVE SPACE TO JOB-TITLE(COUNTER).
MOVE ZERO TO SALARY(COUNTER).

GET-DEPT-NUM.
MOVE ZERO TO DEPT-NUM.
DISPLAY " ".
DISPLAY "ENTER DEPARTMENT NUMBER: " WITH NO ADVANCING.
ACCEPT DEPT-NUM.

DISPLAY-HEADER.
DISPLAY " ".
DISPLAY "EMPLOYEE    JOB TITLE    SALARY".
DISPLAY "––––––––    –––––––––    ––––––".

FETCH-BATCH.
EXEC SQL EXECUTE
BEGIN
CALLDEMO.GET_EMPLOYEES
 (:DEPT-NUM, :TABLE-SIZE,
 :NUM-RET, :DONE-FLAG,
 :EMP-NAME, :JOB-TITLE, :SALARY);
END;
END-EXEC.
PERFORM PRINT-ROWS VARYING COUNTER FROM 1 BY 1
UNTIL COUNTER > NUM-RET.
PRINT-ROWS.
  MOVE EMP-NAME(COUNTER) TO D-EMP-NAME.
  MOVE JOB-TITLE(COUNTER) TO D-JOB-TITLE.
  MOVE SALARY(COUNTER) TO D-SALARY.
  DISPLAY D-EMP-NAME, "   ",
       D-JOB-TITLE, "   ",
       D-SALARY.

LOGON.
  MOVE "SCOTT" TO USERNAME-ARR.
  MOVE 5 TO USERNAME-LEN.
  MOVE "TIGER" TO PASSWD-ARR.
  MOVE 5 TO PASSWD-LEN.
  EXEC SQL
    CONNECT :USERNAME IDENTIFIED BY :PASSWD
  END-EXEC.
  DISPLAY " ".
  DISPLAY "CONNECTED TO ORACLE AS USER: ", USERNAME-ARR.

LOGOFF.
  DISPLAY " ".
  DISPLAY "HAVE A GOOD DAY.".
  DISPLAY " ".
  EXEC SQL
    COMMIT WORK RELEASE
  END-EXEC.
  STOP RUN.

SQL-ERROR.
  EXEC SQL
    WHENEVER SQLERROR CONTINUE
  END-EXEC.
  DISPLAY " ".
  DISPLAY "ORACLE ERROR DETECTED: ".
  DISPLAY " ".
  DISPLAY SQLERRMC.
  EXEC SQL
    ROLLBACK WORK RELEASE
  END-EXEC.
  STOP RUN.
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 10 of the *Programmer’s Guide to the Oracle Precompilers.*
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 (input) 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 = :DNUM AND LOC = :DLOC
```

Placeholders cannot reference table or column names.

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
- take explicit control over datatype conversion between Oracle and COBOL types

To add this flexibility to your program, you must give the Oracle runtime library additional information.

What Information Does Oracle Need?

The Pro*COBOL 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:

```
* Dynamic SQL statement...
STMT = ‘DELETE FROM EMP WHERE DEPTNO = 30’
```

However, most dynamic SQL statements contain select–list items or placeholders for bind variables, as shown in the following UPDATE statement:

```
* Dynamic SQL statement with placeholders...
STMT = ‘UPDATE EMP SET COMM = :C WHERE EMPNO = :E’
```
To execute a dynamic SQL statement that contains select–list items and/or placeholders for bind variables, Oracle needs information about the program variables that will hold output or input values. Specifically, Oracle needs the following information:

- the number of select–list items and the number of bind variables
- the length of each select–list item and bind variable
- the datatype of each select–list item and bind variable
- the memory address of each output variable that will store the value of a select–list item, and the address of each bind variable

For example, to write the value of a select–list item, Oracle needs the address of the corresponding output variable.

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).

Descriptions of select–list items are stored in a select SQLDA, and descriptions of placeholders for bind variables are stored in a bind SQLDA.

The values of select–list items are stored in output buffers; the values of bind variables are stored in input buffers. You use the library routine SQLADR to store the addresses of these data buffers in a 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 filled in by your program, typically from information entered interactively by the user.

How Is the Information Obtained?

You use the DESCRIBE statement to help get the information Oracle needs. The DESCRIBE SELECT LIST statement examines each select–list item to determine its name, datatype, constraints, length, scale, and precision, 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. DESCRIBE also stores the total number of select–list items in the SQLDA.

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 SQL Descriptor Area (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 corresponds to a group item in your program.

A select descriptor stores descriptions of select–list items and the addresses of output buffers that hold the names and values of select–list items.

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 stores 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.

Remember, some descriptor variables contain addresses, not values. So, you must declare data buffers to hold the values. You decide the sizes of the required input and output buffers. Because COBOL does not support pointers, you must use the library subroutine SQLADR to get the addresses of input and output buffers. You learn how to call SQLADR in the section “Using SQLADR” on page 4 – 13.

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 SELDSC1, SELDSC2, and SELDSC3, so that you can FETCH from three concurrently open cursors. However, non–concurrent cursors can reuse SQLDAs.

Declaring a SQLDA

To declare select and bind SQLDAs, you can hardcode them into your program using the sample select and bind SQLDAs shown in Figure 4 – 1. You can modify the table dimensions to suit your needs.

Note: For byte–swapped platforms, use COMP5 instead of COMP when declaring a SQLDA.
Figure 4 – 1 Sample Pro*COBOL SQLDA Descriptors and Data Buffers
You might want to store the SQLDAs in files (named SELDSC and BNDDSC, for example), then copy the files into your program with the INCLUDE statement as follows:

```
EXEC SQL INCLUDE SELDSC END-EXEC.
EXEC SQL INCLUDE BNDDSC END-EXEC.
```

Figure 4 – 2 shows whether variables are set by SQLADR calls, DESCRIBE commands, FETCH commands, or program assignments.

---

**Figure 4 – 2 How Variables Are Set**

<table>
<thead>
<tr>
<th>Dynamic SQL Statement</th>
<th>'SELECT ENAME FROM EMP WHERE EMPNO=:NUM'</th>
</tr>
</thead>
<tbody>
<tr>
<td>select-list Item (SLI)</td>
<td>^</td>
</tr>
<tr>
<td>Select SQLDA</td>
<td>^</td>
</tr>
<tr>
<td>Set by:</td>
<td></td>
</tr>
<tr>
<td>SQLADR</td>
<td>Address of SLI name buffer</td>
</tr>
<tr>
<td>DESCRIBE</td>
<td>Address of SLI value buffer</td>
</tr>
<tr>
<td>DESCRIBE</td>
<td>Length of SLI name</td>
</tr>
<tr>
<td>DESCRIBE</td>
<td>Datatype of select-list item</td>
</tr>
<tr>
<td>Program</td>
<td>Length of SLI name buffer</td>
</tr>
<tr>
<td>Program</td>
<td>Length of SLI value buffer</td>
</tr>
<tr>
<td>Program</td>
<td>Datatype of SLI value buffer</td>
</tr>
<tr>
<td>Bind SQLDA</td>
<td>^</td>
</tr>
<tr>
<td>Address of P name buffer</td>
<td></td>
</tr>
<tr>
<td>Address of BV value buffer</td>
<td></td>
</tr>
<tr>
<td>Length of P name</td>
<td>^</td>
</tr>
<tr>
<td>Length of BV value buffer</td>
<td></td>
</tr>
<tr>
<td>Datatype of BV value buffer</td>
<td></td>
</tr>
<tr>
<td>Output Buffers</td>
<td></td>
</tr>
<tr>
<td>DESCRIBE</td>
<td>Name of select-list item</td>
</tr>
<tr>
<td>FETCH</td>
<td>Value of select-list item</td>
</tr>
<tr>
<td>Input Buffers</td>
<td></td>
</tr>
<tr>
<td>Name of placeholder</td>
<td></td>
</tr>
<tr>
<td>Value of bind variable</td>
<td></td>
</tr>
</tbody>
</table>
The SQLDA Variables

This section explains the purpose and use of each variable in the SQLDA.

**SQLDNUM**

This variable specifies the maximum number of select–list items or placeholders that can be DESCRIBEd. Thus, SQLDNUM determines the number of elements in the descriptor tables.

Before issuing a DESCRIBE command, you must set this variable to the dimension of the descriptor tables. After the DESCRIBE, you must reset it to the actual number of variables DESCRIBEd, which is stored in SQLDFND.

**SQLDFND**

This is the actual number of select–list items or placeholders found by the DESCRIBE command.

SQLDFND is set by DESCRIBE. If SQLDFND is negative, the DESCRIBE command found too many select–list items or placeholders for the size of the descriptor. For example, if you set SQLDNUM to 10 but DESCRIBE finds 11 select–list items or placeholders, SQLDFND is set to –11. If this happens, you cannot process the SQL statement without reallocating the descriptor.

After the DESCRIBE, you must set SQLDNUM equal to SQLDFND.

**SELDV | BNDDV**

This is a table containing the addresses of data buffers that store select–list or bind–variable values.

You must set the elements of SELDV or BNDDV using SQLADR.

**Select Descriptors**

The following statement

```
EXEC SQL FETCH ... USING DESCRIPTOR ...
```

directs Oracle to store FETCHed select–list values in the data buffers addressed by SELDV(1) through SELDV(SQLDNUM). Thus, Oracle stores the Jth select–list value in SEL–DV(J).

**Bind Descriptors**

You must set this table before issuing the OPEN command. The following statement

```
EXEC SQL OPEN ... USING DESCRIPTOR ...
```

directs Oracle to execute the dynamic SQL statement using the bind–variable values addressed by BNDDV(1) through BNDDV(SQLDNUM). (Typically, the values are entered by the user.) Oracle finds the Jth bind–variable value in BND–DV(J).
This is a table containing the addresses of data buffers that store select-list or bind-variable conversion format strings. Currently, you can use it only for COBOL packed decimals. The format for the conversion string is PP.+SS or PP.–SS where PP is the precision and SS is the scale. For definitions of precision and scale, see the section “Extracting Precision and Scale” on page 4–18.

The use of format strings is optional. If you want a conversion format for the Jth select-list item or bind variable, set SELDFMT(J) or BNDDFMT(J) using SQLADR, then store the packed-decimal format (“07.+02” for example) in SEL–DFMT or BND–DFMT. Otherwise, set SELDFMT(J) or BNDDFMT(J) to zero.

This is a table containing the lengths of select-list or bind-variable values stored in the data buffers.

DESCRIBE SELECT LIST sets the table 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 SELDVLN(J) before the FETCH command.

The format of the length differs among Oracle datatypes. For CHAR select-list items, DESCRIBE SELECT LIST sets SELDVLN(J) to the maximum length in bytes 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 routine SQLPRC to extract precision and scale values from SELDVLN. See the section “Extracting Precision and Scale” on page 4–18.

You must reset SELDVLN(J) to the required length of the data buffer before the FETCH. For example, when coercing a NUMBER to a COBOL character string, set SELDVLN(J) to the precision of the number plus two for the sign and decimal point. When coercing a NUMBER to a COBOL floating point number, set SELDVLN(J) to the length of the appropriate floating point type on your system. For more information about the lengths of coerced datatypes, see the section “Converting Data” on page 4–14.
Bind Descriptors

You must set the table of lengths before issuing the OPEN command. For example, you can use the following statements to set the lengths of bind–variable character strings entered by the user:

```
PROCEDURE DIVISION.
...
PERFORM GET–INPUT–VAR
   VARYING J FROM 1 BY 1 UNTIL J > SQLDNUM IN BNDDSC.
...
GET–INPUT–VAR.
   DISPLAY "Enter value of ", BND–DH–VNAME(J).
   ACCEPT INPUT–STRING.
   UNSTRING INPUT–STRING DELIMITED BY "  "
      INTO BND–DV(J) COUNT IN BNDDVLN(J).
```

Because Oracle accesses a data buffer indirectly, using the address in SELDV(J) or BNDDV(J), it does not know the length of the value in that buffer. If you want to change the length Oracle uses for the Jth select–list or bind–variable value, reset SELDVLN(J) or BNDDVLN(J) to the length you need. Each input or output buffer can have a different length.

SELDFMTL | BNDFMTL

This is a table containing the lengths of select–list or bind–variable conversion format strings. Currently, you can use it only for COBOL packed decimals.

The use of format strings is optional. If you want a conversion format for the Jth select–list item or bind variable, set SELDFMTL(J) before the FETCH or BNDFMTL(J) before the OPEN to the length of the packed–decimal format stored in SEL–DFMT or BND–DFMT. Otherwise, set SELDFMTL(J) or BNDFMTL(J) to zero.

If the value of SELDFMTL(J) or BNDFMTL(J) is zero, SELDFMT(J) or BNDFMT(J) is not used.
SELDVTYP | BNDDVTYP

This is a table containing the 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 SELDV. This topic is covered in “Converting Data” on page 4–14.

Select Descriptors

DESCRIBE SELECT LIST sets the table of datatype codes to the internal datatype (for example, VARCHAR2, CHAR, NUMBER, or DATE) 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. For calculations, you might want to coerce numbers from Oracle to COBOL format. See “Coercing Datatypes” on page 4–17.

The high bit of SELDVTYP(J) is set to indicate the null/not null status of the Jth select-list column. You must always clear this bit before issuing an OPEN or FETCH command. You use the library routine SQLNUL to retrieve the datatype code and clear the null/not null bit. For more information, see “Handling Null/Not Null Datatypes” on page 4–20.

You should change the Oracle NUMBER internal datatype to an external datatype compatible with that of the COBOL data buffer addressed by SELDV(J).

Bind Descriptors

DESCRIBE BIND VARIABLES sets the table of datatype codes to zeros. You must reset the table of datatypes before issuing the OPEN command. The code represents the external (COBOL) datatype of the buffer addressed by BNDDV(J). Often, bind-variable values are stored in character strings, so the datatype table elements are set to 1 (the VARCHAR2 datatype code).

To change the datatype of the Jth select-list or bind-variable value, reset SELDVTYP(J) or BNDDVTYP(J) to the datatype you want.
This is a table containing the addresses of data buffers that store indicator–variable values. You must set the elements of SELDI or BNDDI using SQLADR.

**Select Descriptors**

You must set this table before issuing the FETCH command. When Oracle executes the statement

```sql
EXEC SQL FETCH ... USING DESCRIPTOR ... 
```

if the Jth returned select–list value is null, the buffer addressed by SELDI(J) is set to −1. Otherwise, it is set to zero (the value is not null) or a positive integer (the value was truncated).

**Bind Descriptors**

You must initialize this table and set the associated indicator variables before issuing the OPEN command. When Oracle executes the statement

```sql
EXEC SQL OPEN ... USING DESCRIPTOR ... 
```

the buffer addressed by BNDDI(J) determines whether the Jth bind variable is null. If the value of an indicator variable is −1, its associated bind variable is null.

This is a table containing the addresses of data buffers that store select–list or placeholder names as they appear in dynamic SQL statements. You must set the elements of SELDH–VNAME or BNDDH–VNAME using SQLADR before issuing the DESCRIBE command.

DESCRIBE directs Oracle to store the name of the Jth select–list item or placeholder in the data buffer addressed by SELDH–VNAME(J) or BNDDH–VNAME(J). Thus, Oracle stores the Jth select–list or placeholder name in SEL–DH–VNAME(J) or BND–DH–VNAME(J).

This is a table containing the maximum lengths of the data buffers that store select–list or placeholder names. The buffers are addressed by the elements of SELDH–VNAME or BNDDH–VNAME.

You must set the elements of SELDH–MAX–VNAMEL or BNDDH–MAX–VNAMEL before issuing the DESCRIBE command. Each select–list or placeholder name buffer can have a different length.

This is a table containing the actual lengths of the names of the select–list or placeholder. DESCRIBE sets the table of actual lengths to the number of characters in each select–list or placeholder name.
<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELDI-VNAME</td>
<td>This is a table containing the 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, you can use this table only with bind descriptors. You must set the elements of BNDDI-VNAME using SQLADR before issuing the DESCRIBE command. DESCRIBE BIND VARIABLES directs Oracle to store any indicator-variable names in the data buffers addressed by BNDDI-VNAME(1) through BNDDI-VNAME(SQLDNUM). Thus, Oracle stores the Jth indicator-variable name in BND-DI-VNAME(J).</td>
</tr>
<tr>
<td>BNDDI-VNAME</td>
<td>This is a table containing the addresses of data buffers that store indicator-variable names. The buffers are addressed by the elements of SELDI-VNAME or BNDDI-VNAME. You can associate indicator-variable names only with bind variables. So, you can use this table only with bind descriptors. You must set the elements BNDDI-MAX-VNAMEL(1) through BNDDI-MAX-VNAMEL(SQLDNUM) before issuing the DESCRIBE command. Each indicator-variable name buffer can have a different length.</td>
</tr>
<tr>
<td>SELDI-MAX-VNAMEL</td>
<td>This is a table containing the maximum lengths of the data buffers that store indicator-variable names. The buffers are addressed by the elements of SELDI-VNAME or BNDDI-VNAME. You can associate indicator-variable names only with bind variables. So, you can use this table only with bind descriptors.</td>
</tr>
<tr>
<td>BNDDI-MAX-VNAMEL</td>
<td>This is a table containing the maximum lengths of the data buffers that store indicator-variable names. The buffers are addressed by the elements of SELDI-VNAME or BNDDI-VNAME. You can associate indicator-variable names only with bind variables. So, you can use this table only with bind descriptors. You must set the elements BNDDI-MAX-VNAMEL(1) through BNDDI-MAX-VNAMEL(SQLDNUM) before issuing the DESCRIBE command. Each indicator-variable name buffer can have a different length.</td>
</tr>
<tr>
<td>SELDI-CUR-VNAMEL</td>
<td>This is a table containing the actual lengths of the names of the indicator variables. You can associate indicator-variable names only with bind variables. So, you can use this table only with bind descriptors. DESCRIBE BIND VARIABLES sets the table of actual lengths to the number of characters in each indicator-variable name.</td>
</tr>
<tr>
<td>BNDDI-CUR-VNAMEL</td>
<td>This is a table containing the actual lengths of the names of the indicator variables. You can associate indicator-variable names only with bind variables. So, you can use this table only with bind descriptors. DESCRIBE BIND VARIABLES sets the table of actual lengths to the number of characters in each indicator-variable name.</td>
</tr>
<tr>
<td>SELDFCLP</td>
<td>This is a table reserved for future use. It must be present because Oracle expects the group item SELDSC or BNDDSC to be a certain size. You must set the elements of SELDFCLP or BNDDFCLP to zero.</td>
</tr>
<tr>
<td>BNDDFCLP</td>
<td>This is a table reserved for future use. It must be present because Oracle expects the group item SELDSC or BNDDSC to be a certain size. You must set the elements of SELDFCRCP or BNDDFCRCP to zero.</td>
</tr>
<tr>
<td>SELDFCRCP</td>
<td>This is a table reserved for future use. It must be present because Oracle expects the group item SELDSC or BNDDSC to be a certain size. You must set the elements of SELDFCRCP or BNDDFCRCP to zero.</td>
</tr>
<tr>
<td>BNDDFCRCP</td>
<td>This is a table reserved for future use. It must be present because Oracle expects the group item SELDSC or BNDDSC to be a certain size. You must set the elements of SELDFCRCP or BNDDFCRCP to zero.</td>
</tr>
</tbody>
</table>
Some Preliminaries

You need a working knowledge of the following subjects to implement dynamic SQL Method 4:

- using the library routine SQLADR
- converting data
- coercing datatypes
- handling null/not null datatypes

Using SQLADR

You must call the library subroutine SQLADR to get the addresses of data buffers that store input and output values. You store the addresses in a bind or select SQLDA so that Oracle knows where to read bind–variable values or write select–list values.

Call SQLADR using the syntax

```
CALL "SQLADR" USING BUFFER, ADDRESS.
```

where:

BUFFER Is a data buffer that stores the value or name of a select–list item, bind variable, or indicator variable.

ADDRESS Is an integer variable that returns the address of the data buffer.

A call to SQLADR stores the address of BUFFER in ADDRESS. In the next example, you use SQLADR to initialize the select descriptor tables SELDV, SELDH–VNAME, and SELDI. Their elements address data buffers for select–list values, select–list names, and indicator values.

```
PROCEDURE DIVISION.
...
PERFORM INIT–SELDSC
    VARYING J FROM 1 BY 1 UNTIL J > SQLDNUM IN SELDSC.
...
INIT–SELDSC.
    CALL "SQLADR" USING SEL–DV(J), SELDV(J).
    CALL "SQLADR" USING SEL–DH–VNAME(J), SELDH–VNAME(J).
    CALL "SQLADR" USING SEL–DI(J), SELDI(J).
...
```
Converting Data

This section provides more detail about the datatype descriptor table. 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 PIC S9(n) COMP. However, Method 4 lets you control data conversion and formatting. You specify conversions by setting datatype codes in the datatype descriptor table.

Internal Datatypes

Internal datatypes specify the formats used by Oracle to store column values in database tables and the formats 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 SELDVTYP (datatype) descriptor table. For example, the datatype code for the Jth select–list item is returned to SELDVTYP(J).

Table 4 – 1 shows the Oracle internal datatypes and their codes.

<table>
<thead>
<tr>
<th>Oracle Internal Datatype</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>VARCHAR2</td>
<td>1</td>
</tr>
<tr>
<td>NUMBER</td>
<td>2</td>
</tr>
<tr>
<td>LONG</td>
<td>8</td>
</tr>
<tr>
<td>ROWID</td>
<td>11</td>
</tr>
<tr>
<td>DATE</td>
<td>12</td>
</tr>
<tr>
<td>RAW</td>
<td>23</td>
</tr>
<tr>
<td>LONG RAW</td>
<td>24</td>
</tr>
<tr>
<td>CHAR</td>
<td>96</td>
</tr>
<tr>
<td>MLSLABEL</td>
<td>105</td>
</tr>
</tbody>
</table>

Table 4 – 1  Oracle Internal Datatypes and Related Codes

External Datatypes

External datatypes specify the formats used to store values in input and output host variables.

The DESCRIBE BIND VARIABLES command sets the BNDDVTYP table 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 Jth bind variable, reset BNDDVTYP(J) to the external datatype you want.
The following table shows the Oracle external datatypes and their codes, as well as the corresponding COBOL datatypes:

<table>
<thead>
<tr>
<th>Name</th>
<th>Code</th>
<th>COBOL Datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>VARCHAR2</td>
<td>1</td>
<td>PIC X(n) when MODE != ANSI</td>
</tr>
<tr>
<td>NUMBER</td>
<td>2</td>
<td>PIC X(n)</td>
</tr>
<tr>
<td>INTEGER</td>
<td>3</td>
<td>PIC S9(n) COMP</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PIC S9(n) COMP5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(COMP5 for byte-swapped platforms)</td>
</tr>
<tr>
<td>FLOAT</td>
<td>4</td>
<td>COMP–1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>COMP–2</td>
</tr>
<tr>
<td>STRING (1)</td>
<td>5</td>
<td>PIC X(n)</td>
</tr>
<tr>
<td>VARNUM</td>
<td>6</td>
<td>PIC X(n)</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>7</td>
<td>PIC S9(n)V9(n) COMP–3</td>
</tr>
<tr>
<td>LONG</td>
<td>8</td>
<td>PIC X(n)</td>
</tr>
<tr>
<td>VARCHAR (2)</td>
<td>9</td>
<td>PIC X(n) VARYING</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PIC N(n) VARYING</td>
</tr>
<tr>
<td>ROWID</td>
<td>11</td>
<td>PIC X(n)</td>
</tr>
<tr>
<td>DATE</td>
<td>12</td>
<td>PIC X(n)</td>
</tr>
<tr>
<td>VARRAW (2)</td>
<td>15</td>
<td>PIC X(n)</td>
</tr>
<tr>
<td>RAW</td>
<td>23</td>
<td>PIC X(n)</td>
</tr>
<tr>
<td>LONG RAW</td>
<td>24</td>
<td>PIC X(n)</td>
</tr>
<tr>
<td>UNSIGNED</td>
<td>68</td>
<td>(not supported)</td>
</tr>
<tr>
<td>DISPLAY</td>
<td>91</td>
<td>PIC S9...9V9...9 DISPLAY SIGN</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LEADING SEPARATE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PIC S9(n)V9(n) DISPLAY SIGN</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LEADING SEPARATE</td>
</tr>
<tr>
<td>LONG VARCHAR (2)</td>
<td>94</td>
<td>PIC X(n)</td>
</tr>
<tr>
<td>LONG VARRAW (2)</td>
<td>95</td>
<td>PIC X(n)</td>
</tr>
<tr>
<td>CHARF</td>
<td>96</td>
<td>PIC X(n) when MODE = ANSI</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PIC N(n) when MODE = ANSI</td>
</tr>
<tr>
<td>CHARZ (1)</td>
<td>97</td>
<td>PIC X(n)</td>
</tr>
<tr>
<td>CURSOR</td>
<td>102</td>
<td>SQL–CURSOR</td>
</tr>
<tr>
<td>MLSLABEL</td>
<td>106</td>
<td>PIC X(n)</td>
</tr>
</tbody>
</table>

Table 4 – 2 Oracle External and Related COBOL Datatypes

Notes:
1. For use in an EXEC SQL VAR statement only.
2. Include the n-byte length field.

For more information about the Oracle datatypes and their formats, see Chapter 3 of the Programmer’s Guide to the Oracle Precompilers.
PL/SQL provides a variety of predefined scalar and composite datatypes. A scalar type has no internal components. A composite type has internal components that can be manipulated individually. Table 4 – 3 shows the predefined PL/SQL scalar datatypes and their Oracle internal datatype equivalences.

<table>
<thead>
<tr>
<th>PL/SQL Datatype</th>
<th>Oracle Internal Datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>VARCHAR</td>
<td>VARCHAR2</td>
</tr>
<tr>
<td>BINARY_INTEGER</td>
<td>NUMBER</td>
</tr>
<tr>
<td>DEC</td>
<td></td>
</tr>
<tr>
<td>DECIMAL</td>
<td></td>
</tr>
<tr>
<td>DOUBLE PRECISION</td>
<td></td>
</tr>
<tr>
<td>FLOAT</td>
<td></td>
</tr>
<tr>
<td>INT</td>
<td></td>
</tr>
<tr>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td>NATURAL</td>
<td></td>
</tr>
<tr>
<td>NUMBER</td>
<td></td>
</tr>
<tr>
<td>NUMERIC</td>
<td></td>
</tr>
<tr>
<td>POSITIVE</td>
<td></td>
</tr>
<tr>
<td>REAL</td>
<td></td>
</tr>
<tr>
<td>SMALLINT</td>
<td></td>
</tr>
<tr>
<td>LONG</td>
<td>LONG</td>
</tr>
<tr>
<td>ROWID</td>
<td>ROWID</td>
</tr>
<tr>
<td>DATE</td>
<td>DATE</td>
</tr>
<tr>
<td>RAW</td>
<td>RAW</td>
</tr>
<tr>
<td>LONG RAW</td>
<td>LONG RAW</td>
</tr>
<tr>
<td>CHAR</td>
<td>CHAR</td>
</tr>
<tr>
<td>CHARACTER</td>
<td></td>
</tr>
<tr>
<td>STRING</td>
<td></td>
</tr>
<tr>
<td>MLSLABEL</td>
<td>MLSLABEL</td>
</tr>
</tbody>
</table>

Table 4 – 3 PL/SQL Datatype Equivalences with Oracle Internal Datatypes
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 SELDVTYP descriptor table.

For example, you might want to reset NUMBER values to FLOAT values, which correspond to PIC S9(n)V9(n) COMP–1 values in COBOL. 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 BNDDVTYP table 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 SELDVTYP or BNDDVTYP descriptor table, you are “coercing datatypes.” For example, to coerce the Jth select–list value to VARCHAR2, use the following statement:

```
* Coerce select–list value to VARCHAR2.
MOVE 1 TO SELDVTYP(J).
```

When coercing a NUMBER select–list value to VARCHAR2 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 SELDVLN (length) descriptor table to tell Oracle the buffer length to use. To specify the length of the Jth select–list value, set SELDVLN(J) to the length you need.

For example, if DESCRIBE SELECT LIST finds that the Jth select–list item is of type NUMBER, and you want to store the returned value in a COBOL variable declared as PIC S9(n)V9(n) COMP–1, simply set SELDVTYP(J) to 4 and SELDVLN(J) to the length of COMP–1 numbers on your system.
Exceptions

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 SELDVTYP table. Unless you reset the code before the FETCH, the date value is returned in its 7-byte internal format. To get the date in its default character format, you must change the datatype code from 12 to 1 (VARCHAR2), and increase the SELDVLN value from 7 to 9.

Similarly, when you DESCRIBE a NUMBER select–list item, Oracle returns the datatype code 2 to the SELDVTYP table. 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), or some other appropriate datatype.

Extracting Precision and Scale

The library subroutine SQLPRC extracts precision and scale. Normally, it is used after the DESCRIBE SELECT LIST, and its first parameter is SELDVLN(J). To call SQLPRC, use the following syntax

CALL "SQLPRC" USING LENGTH, PRECISION, SCALE.

where:

LENGTH
Is an integer variable that stores the length of an Oracle NUMBER value. The scale and precision of the value are stored in the low and next–higher bytes, respectively.

PRECISION
Is 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 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).
The following example shows how SQLPRC is used to compute maximum display lengths for NUMBER values that will be coerced to VARCHAR2:

```
WORKING-STORAGE SECTION.
  01  PRECISION       PIC S9(9) COMP.
  01  SCALE           PIC S9(9) COMP.
  01  DISPLAY-LENGTH  PIC S9(9) COMP.
  01  MAX-LENGTH      PIC S9(9) COMP VALUE 80.
...
PROCEDURE DIVISION.
...
  PERFORM ADJUST-LENGTH
    VARYING J FROM 1 BY 1 UNTIL J > SQLDNUM IN SELDSC.

ADJUST-LENGTH.
* If datatype is NUMBER, extract precision and scale.
  IF SELDVTYP(J) = 2
    CALL "SQLPRC" USING SELDVLN(J), PRECISION, SCALE.
    MOVE 0 TO DISPLAY-LENGTH.
* Precision is set to zero if the select-list item
* refers to a NUMBER of unspecified size. We allow for
* a maximum precision of 10.
  IF SELDVTYP(J) = 2 AND PRECISION = 0
    MOVE 10 TO DISPLAY-LENGTH.
* Allow for possible decimal point and sign.
  IF SELDVTYP(J) = 2 AND PRECISION > 0
    ADD 2 TO PRECISION
    MOVE PRECISION TO DISPLAY-LENGTH.
...
```

Notice that the first parameter in the subroutine call is the Jth element in the table of select-list lengths.

The SQLPRC procedure, defined in the SQLLIB runtime library, returns zero as the precision and scale values for certain SQL datatypes. The SQLPR2 procedure is similar to SQLPRC in that it has the same syntax and returns the same binary values, except for the datatypes shown in Table 4 – 4.

<table>
<thead>
<tr>
<th>SQL Datatype</th>
<th>Binary Precision</th>
<th>Binary Scale</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLOAT</td>
<td>126</td>
<td>-127</td>
</tr>
<tr>
<td>FLOAT(n)</td>
<td>n (range is 1..126)</td>
<td>-127</td>
</tr>
<tr>
<td>REAL</td>
<td>63</td>
<td>-127</td>
</tr>
<tr>
<td>DOUBLE PRECISION</td>
<td>126</td>
<td>-127</td>
</tr>
</tbody>
</table>

Table 4 – 4 Datatype Exceptions to the SQLPR2 Procedure
Handling Null/Not Null Datatypes

For every select–list column (not expression), DESCRIBE SELECT LIST returns a null/not null indication in the datatype table of the select descriptor. If the Jth select–list column is constrained to be not null, the high–order bit of SELDVTYP(J) datatype variable is clear; otherwise, it is set.

Before using the datatype in an OPEN or FETCH statement, if the null status bit is set, you must clear it. Never set the bit.

You can use the library routine SQLNUL to find out if a column allows nulls, and to clear the datatype’s null status bit. You call SQLNUL using the syntax

```
CALL "SQLNUL" USING VALUE–TYPE, TYPE–CODE, NULL–STATUS.
```

where:

- **VALUE–TYPE** is a 2–byte integer variable that stores the datatype code of a select–list column.
- **TYPE–CODE** is a 2–byte integer variable that returns the datatype code of the select–list column with the high–order bit cleared.
- **NULL–STATUS** is 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:

```
WORKING–STORAGE SECTION.
...
* Declare variable for subroutine call.
  01 NULL–STATUS PIC S9(9) COMP.
...
PROCEDURE DIVISION.
MAIN.
  EXEC SQL WHENEVER SQLERROR GOTO SQL–ERROR END–EXEC.
  ...
  PERFORM HANDLE–NULLS
      VARYING J FROM 1 BY 1 UNTIL J > SQLDNUM IN SELDSC.
  ...
  HANDLE–NULLS.
* Find out if column is NOT NULL, and clear high–order bit.
  CALL "SQLNUL" USING SELDVTYP(J), SELDVTYP(J), NULL–STATUS.
* If NULL–STATUS = 1, nulls are allowed.
```

Notice that the first and second parameters in the subroutine call are the same. Respectively, they are the datatype variable before and after its null status bit is cleared.
The Basic Steps

Method 4 can be used to process any dynamic SQL statement. In the example on page 4 – 37, a query is processed so that 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 descriptors.
3. Set the maximum number of select-list items and placeholders that can be DESCRIBEd.
4. Initialize the select and bind descriptors.
5. Store 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 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 data buffers using the select descriptor.
16. Process the select-list values returned by FETCH.
17. CLOSE the cursor when there are no more rows to FETCH.

Note: If the dynamic SQL statement is not a query or contains a known number of select-list items or placeholders, then some of the above steps are unnecessary.
A Closer Look at Each Step

This section discusses each step in more detail. Also, at the end of this chapter is a full-length program illustrating Method 4. With Method 4, you use the following sequence of embedded SQL statements:

```sql
EXEC SQL
  PREPARE <statement_name>
  FROM {:<host_string>|<string_literal>}
END-EXEC.
EXEC SQL
  DECLARE <cursor_name> CURSOR FOR <statement_name>
END-EXEC.
EXEC SQL
  DESCRIBE BIND VARIABLES FOR <statement_name>
  INTO <bind_descriptor_name>
END-EXEC.
EXEC SQL
  OPEN <cursor_name> 
  [USING DESCRIPTOR <bind_descriptor_name>]
END-EXEC.
EXEC SQL
  DESCRIBE [SELECT LIST FOR] <statement_name>
  INTO <select_descriptor_name>
END-EXEC.
EXEC SQL
  FETCH <cursor_name> USING DESCRIPTOR <select_descriptor_name> 
END-EXEC.
EXEC SQL
  CLOSE <cursor_name>
END-EXEC.
```

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:

```sql
EXEC SQL FETCH <cursor_name> INTO <host_variable_list> END-EXEC.
```

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:

```sql
EXEC SQL OPEN <cursor_name> [USING <host_variable_list>] END-EXEC.
```

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 confine descriptor tables to 3 elements and to limit the maximum length of names and values to 5 and 10 characters, respectively.
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:

```sql
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
...
01 SELECT–STMT PIC X(120).
EXEC SQL END DECLARE SECTION END-EXEC.
```

Declare the SQLDAs

Because the query in our example might contain an unknown number of select–list items or placeholders, you must declare select and bind descriptors. Instead of hardcoding the SQLDAs, you use INCLUDE to copy them into your program, as follows:

```sql
EXEC SQL INCLUDE SELDSC END-EXEC.
EXEC SQL INCLUDE BNDDSC END-EXEC.
```

For reference, the INCLUDEd declaration of SELDSC follows:

```sql
WORKING-STORAGE SECTION.
...
01 SELDSC.
  05 SQLDNUM PIC S9(9) COMP.
  05 SQLDFND PIC S9(9) COMP.
  05 SELDVAR OCCURS 3 TIMES.
    10 SELDV PIC S9(9) COMP.
    10 SELEDFMT PIC S9(9) COMP.
    10 SELDVLEN PIC S9(9) COMP.
    10 SELDFMTL PIC S9(4) COMP.
    10 SELDFMET PIC S9(4) COMP.
    10 SELDI PIC S9(9) COMP.
    10 SELDH–VNAME PIC S9(9) COMP.
    10 SELDH–MAX–VNAMEL PIC S9(4) COMP.
    10 SELDH–CUR–VNAMEL PIC S9(4) COMP.
    10 SELDI–VNAME PIC S9(9) COMP.
    10 SELDI–MAX–VNAMEL PIC S9(4) COMP.
    10 SELDI–CUR–VNAMEL PIC S9(4) COMP.
    10 SELDFCLP PIC S9(9) COMP.
    10 SELDFCRCP PIC S9(9) COMP.
01 XSELDI.
  05 SEL–DI OCCURS 3 TIMES PIC S9(9) COMP.
01 XSELDIVNAME.
  05 SEL–DI–VNAME OCCURS 3 TIMES PIC X(5).
01 XSELDV.
  05 SEL–DV OCCURS 3 TIMES PIC X(10).
01 XSELDHVNAME.
  05 SEL–DH–VNAME OCCURS 3 TIMES PIC X(5).
```
Next, you set the maximum number of select-list items or placeholders that can be DESCRIBEd, as follows:

```
MOVE 3 TO SQLDNUM IN SELDSC.
MOVE 3 TO SQLDNUM IN BNDDSC.
```

You must initialize several descriptor variables. Some require the library subroutine SQLADR.

In our example, you store the maximum lengths of name buffers in the SELDH–MAX–VNAMEL, BNDDH–MAX–VNAMEL, and BNDDI–MAX–VNAMEL tables, and use SQLADR to store the addresses of value and name buffers in the SELDV, SELDI, BNDDV, BNDDI, SELDH–VNAME, BNDDH–VNAME, and BNDDI–VNAME tables.

```
PROCEDURE DIVISION.
...
PERFORM INIT–SELDSC
    VARYING J FROM 1 BY 1 UNTIL J > SQLDNUM IN SELDSC.
PERFORM INIT–BNDDSC
    VARYING J FROM 1 BY 1 UNTIL J > SQLDNUM IN BNDDSC.
...
INIT–SELDSC.
    MOVE SPACES TO SEL–DV(J).
    MOVE SPACES TO SEL–DH–VNAME(J).
    MOVE 5 TO SELDH–MAX–VNAMEL(J).
    CALL "SQLADR" USING SEL–DV(J), SELDV(J).
    CALL "SQLADR" USING SEL–DH–VNAME(J), SELDH–VNAME(J).
    CALL "SQLADR" USING SEL–DI(J), SELDI(J).
...
INIT–BNDDSC.
    MOVE SPACES TO BND–DV(J).
    MOVE SPACES TO BND–DH–VNAME(J).
    MOVE 5 TO BNDDH–MAX–VNAMEL(J).
    MOVE 5 TO BNDDI–MAX–VNAMEL(J).
    CALL "SQLADR" USING BND–DV(J), BNDDV(J).
    CALL "SQLADR" USING BND–DH–VNAME(J), BNDDH–VNAME(J).
    CALL "SQLADR" USING BND–DI(J), BNDDI(J).
    CALL "SQLADR" USING BND–DI–VNAME(J), BNDDI–VNAME(J).
...
```

Figure 4 – 3 and Figure 4 – 4 represent the resulting descriptors.
Figure 4–3 Initialized Select Descriptor
| SQLDNUM | 3 |
| SQLDFND |   |
| BNDDV   | 1 | address of BND–DV(1) |
|         | 2 | address of BND–DV(2) |
|         | 3 | address of BND–DV(3) |
| BNDDVLN | 2 |   |
|         | 3 |   |
| BNDDVTYP| 2 |   |
|         | 3 |   |
| BNDDI   | 1 | address of BND–DI(1) |
|         | 2 | address of BND–DI(2) |
|         | 3 | address of BND–DI(3) |
| BNDDH–VNAME | 1 | address of BND–DI–VNAME(1) |
|           | 2 | address of BND–DI–VNAME(2) |
|           | 3 | address of BND–DI–VNAME(3) |
| BNDDH–MAX–VNAME | 2 | 5 |
|             | 3 | 5 |
| BNDDH–CUR–VNAME | 2 | 1 |
|             | 3 | 3 |

**Data Buffers**

For values of bind variables:

For values of indicators:

For names of placeholders:

![Diagram of Initialized Bind Descriptor](image)

Figure 4–4 Initialized Bind Descriptor
Store 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:

DISPLAY "Enter a SELECT statement: " WITH NO ADVANCING.
ACCEPT SELECT–STMT.

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 END–EXEC.

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 END–EXEC.

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 bind variables into a bind descriptor. In our example, DESCRIBE readsies BNDDSC as follows:

EXEC SQL
    DESCRIBE BIND VARIABLES FOR SQL–STMT
    INTO BNDDSC
END–EXEC.

Note that BNDDSC must not be prefixed with a colon.

The DESCRIBE BIND VARIABLES statement must follow the PREPARE statement but precede the OPEN statement.

Figure 4–5 shows the bind descriptor in our example after the DESCRIBE. Notice that DESCRIBE has set SQLDFND to the actual number of placeholders found in the processed SQL statement.
<table>
<thead>
<tr>
<th>SQLDNUM</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLDFND</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BNDDV</th>
<th>1</th>
<th>address of BND–DV(1)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>address of BND–DV(2)</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>address of BND–DV(3)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BNDDV</th>
<th>1</th>
<th>set by DESCRIBE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BNDDVTYP</th>
<th>2</th>
<th>set by DESCRIBE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BNDDI</th>
<th>1</th>
<th>address of BND–DI(1)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>address of BND–DI(2)</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>address of BND–DI(3)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BNDDH–VNAME</th>
<th>1</th>
<th>address of BND–DH–VNAME(1)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>address of BND–DH–VNAME(2)</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>address of BND–DH–VNAME(3)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BNDDH–MAX–VNAME</th>
<th>2</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BNDDH–CUR–VNAME</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BNDDH–VNAME</th>
<th>1</th>
<th>address of BND–DI–VNAME(1)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>address of BND–DI–VNAME(2)</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>address of BND–DI–VNAME(3)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BNDDH–MAX–VNAME</th>
<th>2</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BNDDH–CUR–VNAME</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

**Data Buffers**

For values of bind variables:

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For values of placeholders:

<table>
<thead>
<tr>
<th>B</th>
<th>O</th>
<th>N</th>
<th>U</th>
<th>S</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For names of placeholders:

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For names of indicators:

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4–5 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:

```
IF SQLDFND IN BNDDSC < 0
  DISPLAY "Too many bind variables".
  GOTO ROLL-BACK
ELSE
  MOVE SQLDFND IN BNDDSC TO SQLDNUM IN BNDDSC.
```

Get Values for Bind Variables

Your program must get values for the bind variables in the SQL statement. 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. Prompt the user for the value, then process it, as follows:

```
PROCEDURE DIVISION.
...
PERFORM GET–INPUT–VAR
  VARYING J FROM 1 BY 1 UNTIL J > SQLDNUM IN BNDDSC.
...
GET–INPUT–VAR.
...
  * Replace the 0 DESCRIBed into the datatype table
  * with a 1 to avoid an "invalid datatype" Oracle error.
    MOVE 1 TO BNDDVTYP(J).

  * Get value of bind variable.
    DISPLAY "Enter value of ", BND–DH–VNAME(J).
    ACCEPT INPUT–STRING.
    UNSTRING INPUT–STRING DELIMITED BY "  "
    INTO BND–DV(J) COUNT IN BNDDVLN(J).
```

Assuming that the user supplied a value of 625 for BONUS, Figure 4 – 6 shows the resulting bind descriptor.
### Data Buffers

<table>
<thead>
<tr>
<th>SQLDNUM</th>
<th>1</th>
<th>reset by program</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLDFND</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>BNDDV</td>
<td>1</td>
<td>address of BND–DV(1)</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>address of BND–DV(2)</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>address of BND–DV(3)</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>set by program</td>
</tr>
<tr>
<td>BNDDVLN</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>reset by program</td>
</tr>
<tr>
<td>BNDDVTYP</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>BNDDI</td>
<td>1</td>
<td>address of BND–DI(1)</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>address of BND–DI(2)</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>address of BND–DI(3)</td>
</tr>
<tr>
<td>BNDDH–VNAME</td>
<td>1</td>
<td>address of BND–DH–VNAME(1)</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>address of BND–DH–VNAME(2)</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>address of BND–DH–VNAME(3)</td>
</tr>
<tr>
<td>BNDDH–MAX–VNAME</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>BNDDH–CUR–VNAME</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>BNDDH–VNAME</td>
<td>1</td>
<td>address of BND–DI–VNAME(1)</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>address of BND–DI–VNAME(2)</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>address of BND–DI–VNAME(3)</td>
</tr>
<tr>
<td>BNDDH–MAX–VNAME</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>BNDDH–CUR–VNAME</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

For values of bind variables:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6</td>
<td>2</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

For values of indicators:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For names of placeholders:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>B</td>
<td>O</td>
<td>N</td>
<td>U</td>
<td>S</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For names of indicators:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4–6 Bind Descriptor after Assigning Values
OPEN the Cursor

The OPEN statement for dynamic queries is similar to the one for static queries, except the cursor is associated with a bind descriptor. Values determined at run time and stored in buffers addressed by elements of the bind descriptor tables 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 BNDDSC as follows:

```
EXEC SQL
OPEN EMP–CUR USING DESCRIPTOR BNDDSC
END–EXEC.
```

Remember, BNDDSC 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 into a select descriptor. In our example, DESCRIBE readies SELDSC as follows:

```
EXEC SQL
DESCRIBE SELECT LIST FOR SQL–STMT INTO SELDSC
END–EXEC.
```

Accessing the Oracle data dictionary, DESCRIBE sets the length and datatype of each select–list value.

Figure 4 – 7 shows the select descriptor in our example after the DESCRIBE. Notice that DESCRIBE has set SQLDFND to the actual number of items found in the query select list. If the SQL statement is not a query, SQLDFND is set to zero.

Also notice that the NUMBER lengths are not usable yet. For columns defined as NUMBER, you must use the library subroutine SQLPRC to extract precision and scale. See the section “Coercing Datatypes” on page 4 – 17.
### Data Buffers

#### For values of select-list items:

<p>| | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

#### For values of indicators:

- 1
- 2
- 3

#### For names of select-list items:

- **ENAME**
- **EMP**
- **NO**
- **COM**

Figure 4 – 7  Select Descriptor after the DESCRIBE
Next, you must reset the maximum number of select–list items to the number actually found by DESCRIBE, as follows:

```
MOVE SQLDPND IN SELDSC TO SQLDNUM IN SELDSC.
```

In our example, before fetching the select–list values, you reset some elements in the length and datatype tables for display purposes.

```
PROCEDURE DIVISION.
...
PERFORM COERCE–COLUMN–TYPE
    VARYING J FROM 1 BY 1 UNTIL J > SQLDNUM IN SELDSC.
...
COERCE–COLUMN–TYPE.
*    Clear NULL bit.
    CALL "SQLNUL" USING SELDVTyp(J), SELDVTyp(J), NULL–STATUS.

*    If datatype is DATE, lengthen to 9 characters.
   IF SELDVTyp(J) = 12
      MOVE 9 TO SELDVLEN(J).

*    If datatype is NUMBER, extract precision and scale.
   MOVE 0 TO DISPLAY–LENGTH.
   IF SELDVTyp(J) = 2 AND PRECISION = 0
      MOVE 10 TO DISPLAY–LENGTH.
   IF SELDVTyp(J) = 2 AND PRECISION > 0
      ADD 2 TO PRECISION
         MOVE PRECISION TO DISPLAY–LENGTH.
   IF SELDVTyp(J) = 2
      IF DISPLAY–LENGTH > MAX–LENGTH
         DISPLAY "Column value too large for data buffer."
         GO TO END–PROGRAM
      ELSE
         MOVE DISPLAY–LENGTH TO SELDVLEN(J).

*    Coerce datatypes to VARCHAR2.
   MOVE 1 TO SELDVTyp(J).
```

Figure 4 – 8 shows the resulting select descriptor. Notice that the NUMBER lengths are now usable and that all the datatypes are VARCHAR2. The lengths in SELDVLEN(2) and SELDVLEN(3) 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.
<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLDNUM</td>
<td>3</td>
<td>reset by program</td>
</tr>
<tr>
<td>SQLDFND</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>SELDV</td>
<td>1</td>
<td>address of SEL–DV(1)</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>address of SEL–DV(2)</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>address of SEL–DV(3)</td>
</tr>
<tr>
<td>SELDVLN</td>
<td>1</td>
<td>reset by program</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td># = binary number</td>
</tr>
<tr>
<td>SELDTYP</td>
<td>1</td>
<td>reset by program</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>SELDI</td>
<td>1</td>
<td>address of SEL–DI(1)</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>address of SEL–DI(2)</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>address of SEL–DI(3)</td>
</tr>
<tr>
<td>SELDH_VNAME</td>
<td>1</td>
<td>address of SEL–DH–VNAME(1)</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>address of SEL–DH–VNAME(2)</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>address of SEL–DH–VNAME(3)</td>
</tr>
<tr>
<td>SELDH_MAX_VNAME</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>SELDH_CUR_VNAME</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

**Data Buffers**

- **For values of select–list items:**
  - 12345
  - 6789 1 0
- **For values of indicators:**
  - 1
  - 2
  - 3
- **For names of select–list items:**
  - EMPLOYEE NAME
  - EMPLOYEE NO
  - COMPANY

*Figure 4–8 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 SQLCODE in the SQLCA, the SQLCODE variable, or the SQLSTATE variable to the “no data found” Oracle error code. In the following example, FETCH returns the values of columns ENAME, EMPNO, and COMM to SELDSC:

```sql
EXEC SQL
  FETCH EMP-CURSOR USING DESCRIPTOR SELDSC
END-EXEC.
```

Figure 4–9 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 SELDV and SELDI.

For output buffers of datatype 1, Oracle, using the lengths stored in SELDVLEN, left-justifies CHAR or VARCHAR2 data, and right-justifies NUMBER data.

The value “MARTIN” was retrieved from a VARCHAR2(10) column in the EMP table. Using the length in SELDVLEN(1), 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 SELDVLEN(2) was increased by two to allow for a possible sign and decimal point, so Oracle right-justifies 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 SELDVLEN(3) was increased by two, so Oracle right-justifies the value in a 9-byte field.

Get and Process Select–List Values

After the FETCH, your program can process the select–list values returned by FETCH. In our example, values for columns ENAME, EMPNO, and COMM are processed.

CLOSE the Cursor

CLOSE disables the cursor. In our example, CLOSE disables EMP-CURSOR as follows:

```sql
EXEC SQL
  CLOSE EMP-CURSOR
END-EXEC.
```
<table>
<thead>
<tr>
<th>Function</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLDNUM</td>
<td>3</td>
</tr>
<tr>
<td>SQLDFND</td>
<td>3</td>
</tr>
<tr>
<td>SELDV</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>3</td>
</tr>
<tr>
<td>SELDVLEN</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>3</td>
</tr>
<tr>
<td>SELDTYP</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>3</td>
</tr>
<tr>
<td>SELDI</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>3</td>
</tr>
<tr>
<td>SELDH_VNAME</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>3</td>
</tr>
<tr>
<td>SELDH_MAX_VNAME</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>3</td>
</tr>
<tr>
<td>SELDH_CUR_VNAME</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>3</td>
</tr>
</tbody>
</table>

**Data Buffers**

For values of select–list items:

<table>
<thead>
<tr>
<th>M</th>
<th>A</th>
<th>R</th>
<th>T</th>
<th>I</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| 4 | 8 | 2 | . | 5 | 0 |

For names of select–list items:

<table>
<thead>
<tr>
<th>E</th>
<th>N</th>
<th>A</th>
<th>M</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>M</td>
<td>P</td>
<td>N</td>
<td>O</td>
</tr>
</tbody>
</table>

| C | O | M | M |

Set by FETCH

Figure 4 – 9 Select Descriptor after the FETCH
Using Host Tables with Method 4

To use input or output host tables with Method 4, you must use the optional FOR clause to tell Oracle the size of your host table. For more information about the FOR clause, see Chapter 9 of the *Programmer’s Guide to the Oracle Precompilers*.

Set descriptor entries for the Jth select-list item or bind variable, but instead of addressing a single data buffer, SELDVLN(J) or BNDDVLN(J) addresses a table of data buffers. Then use a FOR clause in the EXECUTE or FETCH statement, as appropriate, to tell Oracle the number of table elements you want to process.

This procedure is necessary, because Oracle has no other way of knowing the size of your host table.

In the example below, two input host tables are used to INSERT data into the EMP table. Note that EXECUTE can be used for non-queries with Method 4.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. DYN4INS.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 BNDDSC.
   02 SQLDNUM PIC S9(9) COMP VALUE 2.
   02 SQLDFND PIC S9(9) COMP.
   02 BNDDVAR OCCURS 2 TIMES.
      03 BNDDR PIC S9(9) COMP.
      03 BNDDFMT PIC S9(9) COMP.
      03 BNDDVLN PIC S9(9) COMP.
      03 BNDDFMTL PIC S9(4) COMP.
      03 BNDDVTYP PIC S9(4) COMP.
      03 BNDDI PIC S9(9) COMP.
      03 BNDDH-VNAME PIC S9(9) COMP.
      03 BNDDH-MAX-VNAMEL PIC S9(4) COMP.
      03 BNDDH-CUR-VNAMEL PIC S9(4) COMP.
      03 BNDDI-VNAME PIC S9(9) COMP.
      03 BNDDI-MAX-VNAMEL PIC S9(4) COMP.
      03 BNDDI-CUR-VNAMEL PIC S9(4) COMP.
      03 BNDDFCLP PIC S9(9) COMP.
      03 BNDDFCRCP PIC S9(9) COMP.
```
Since you know what the SQL statement will be, you can set up a two-dimensional table with a maximum of 2 columns and 8 rows. Each element can be up to 10 characters long. (You can alter these values according to your needs.)

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.

EXEC SQL INCLUDE SQLCA END-EXEC.

PROCEDURE DIVISION.

START-MAIN.

EXEC SQL WHENEVER SQLERROR GOTO SQL-ERROR END-EXEC.

MOVE "SCOTT" TO USERNAME.
MOVE "TIGER" TO PASSWD.
EXEC SQL
    CONNECT :USERNAME IDENTIFIED BY :PASSWD
END-EXEC.
DISPLAY "Connected to Oracle".

* Initialize bind and select descriptors.
PERFORM INIT-BNDDSC THRU INIT-BNDDSC-EXIT
    VARYING COLUMN-INDEX FROM 1 BY 1
    UNTIL COLUMN-INDEX > 2.

* Set up the SQL statement.
MOVE SPACES TO DYN-STATEMENT.
MOVE "INSERT INTO EMP(EMPNO, DEPTNO) VALUES(:E,:D)" TO DYN-STATEMENT.
```
* Prepare the SQL statement.
  EXEC SQL
  
  PREPARE S1 FROM :DYN-STATEMENT
  END-EXEC.

* Describe the bind variables.
  EXEC SQL
  
  DESCRIBE BIND VARIABLES FOR S1 INTO BNDDSC
  END-EXEC.

  PERFORM Z-BIND-TYPE THRU Z-BIND-TYPE-EXIT
  
  VARYING COLUMN-INDEX FROM 1 BY 1
  UNTIL COLUMN-INDEX > 2.

  IF SQLDFND IN BNDDSC < 0
  
  DISPLAY "TOO MANY BIND VARIABLES."
  GO TO SQL-ERROR
  ELSE
  
  DISPLAY "BIND VARS = " WITH NO ADVANCING
  MOVE SQLDFND IN BNDDSC TO DUMMY-INTEGER
  DISPLAY DUMMY-INTEGER
  MOVE SQLDFND IN BNDDSC TO SQLDNUM IN BNDDSC.

  MOVE 8 TO NUMBER-OF-ROWS.
  PERFORM GET-ALL-VALUES THRU GET-ALL-VALUES-EXIT
  
  VARYING ROW-INDEX FROM 1 BY 1
  UNTIL ROW-INDEX > NUMBER-OF-ROWS.

* Execute the SQL statement.
  EXEC SQL FOR :NUMBER-OF-ROWS
  
  EXECUTE S1 USING DESCRIPTOR BNDDSC
  END-EXEC.

  DISPLAY "INSERTED " WITH NO ADVANCING.
  MOVE SQLERRD(3) TO DUMMY-INTEGER.
  DISPLAY DUMMY-INTEGER WITH NO ADVANCING.
  DISPLAY " ROWS.".
  GO TO END-SQL.

SQL-ERROR.

* Display any SQL error message and code.
  DISPLAY SQLERRMC.
  EXEC SQL ROLLBACK WORK RELEASE END-EXEC.
  STOP RUN.
EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
EXEC SQL COMMIT WORK RELEASE END-EXEC.
STOP RUN.

INIT-BNDDSC.
* Start of COBOL PERFORM procedures, initialize the bind descriptor.
MOVE 80 TO BNDDH-MAX-VNAMEL(COLUMN-INDEX).
CALL "SQLADR" USING
  BND-DH-VNAME(COLUMN-INDEX)
  BNDDH-VNAME(COLUMN-INDEX).
MOVE 80 TO BNDDI-MAX-VNAMEL(COLUMN-INDEX).
CALL "SQLADR" USING
  BND-DI-VNAME(COLUMN-INDEX)
  BNDDI-VNAME (COLUMN-INDEX).
MOVE 10 TO BNDDVLEN(COLUMN-INDEX).
CALL "SQLADR" USING
  BND-ELEMENT(COLUMN-INDEX,1)
  BNDDV(COLUMN-INDEX).
MOVE ZERO TO BNDDI(COLUMN-INDEX).
CALL "SQLADR" USING
  BND-DI(COLUMN-INDEX)
  BNDDI(COLUMN-INDEX).
MOVE ZERO TO BNDDFMT(COLUMN-INDEX).
MOVE ZERO TO BNDDFMTL(COLUMN-INDEX).
MOVE ZERO TO BNDDFCLP(COLUMN-INDEX).
MOVE ZERO TO BNDDFCRCP(COLUMN-INDEX).
INIT-BNDDSC-EXIT.
EXIT.

Z-BIND-TYPE.
* Replace the 0s DESCRIBEd into the datatype table with 1s to avoid an “invalid datatype” Oracle error.
MOVE 1 TO BNDDVTYPE(COLUMN-INDEX).
Z-BIND-TYPE-EXIT.
EXIT.

GET-ALL-VALUES.
* Get the bind variables for each row.
DISPLAY “ENTER VALUES FOR ROW NUMBER “,ROW-INDEX.
PERFORM GET-BIND-VARS
  VARYING COLUMN-INDEX FROM 1 BY 1
  UNTIL COLUMN-INDEX > SQLDFND IN BNDDSC.
GET-ALL-VALUES-EXIT.
EXIT.
GET-BIND-VARS.
* Get the value of each bind variable.
   DISPLAY " ENTER VALUE FOR ",BND-DH-VNAME(COLUMN-INDEX)
   WITH NO ADVANCING.
   ACCEPT BND-ELEMENT(COLUMN-INDEX,ROW-INDEX).
GET-BIND-VARS-EXIT.
EXIT.
Sample Program 10: Dynamic SQL Method 4

This program shows the basic steps required to use dynamic SQL Method 4. After logging on to Oracle, the program 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 variables. If the input SQL statement is a query, the program FETCHes each row of data, then CLOSEs the cursor.

IDENTIFICATION DIVISION.
PROGRAM-ID. DYNSQL4.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
  01 BNDDSC.
    02 SQLDNUM PIC S9(9) COMP VALUE 20.
    02 SQLDFND PIC S9(9) COMP.
    02 BNDDVVAR OCCURS 20 TIMES.
      03 BNDDV PIC S9(9) COMP.
      03 BNDDFMT PIC S9(9) COMP.
      03 BNDDVLEN PIC S9(9) COMP.
      03 BNDDFML PIC S9(4) COMP.
      03 BNDDVTYP PIC S9(4) COMP.
      03 BNDDI PIC S9(9) COMP.
      03 BNDDH-VNAME PIC S9(9) COMP.
      03 BNDDH-MAX-VNAMEL PIC S9(4) COMP.
      03 BNDDH-CUR-VNAMEL PIC S9(4) COMP.
      03 BNDDI-VNAME PIC S9(9) COMP.
      03 BNDDI-MAX-VNAMEL PIC S9(4) COMP.
      03 BNDDI-CUR-VNAMEL PIC S9(4) COMP.
      03 BNDDFCLP PIC S9(9) COMP.
      03 BNDDFCRCRP PIC S9(9) COMP.
    01 XBNDDI.
      03 BND-DI OCCURS 20 TIMES PIC S9(4) COMP.
    01 XBNDDDIVNAME.
      03 BND-DI-VNAME OCCURS 20 TIMES PIC X(80).
    01 XBNDDV.
      03 BND-DV OCCURS 20 TIMES PIC X(80).
    01 XBNDDHVNNAME.
      03 BND-DH-VNAME OCCURS 20 TIMES PIC X(80).
01 SELDSC.
  02 SQLDNUM PIC S9(9) COMP VALUE 20.
  02 SQLDFND PIC S9(9) COMP.
  02 SELDVAR OCCURS 20 TIMES.
    03 SELDV PIC S9(9) COMP.
    03 SELDPMT PIC S9(9) COMP.
    03 SELDVLM PIC S9(9) COMP.
    03 SELDPMTL PIC S9(4) COMP.
    03 SELDVTYP PIC S9(4) COMP.
    03 SELDI PIC S9(9) COMP.
  03 SELDH–VNAME PIC S9(9) COMP.
  03 SELDH–MAX–VNAMEL PIC S9(4) COMP.
  03 SELDH–CUR–VNAMEL PIC S9(4) COMP.
  03 SELDI–VNAME PIC S9(9) COMP.
  03 SELDI–MAX–VNAMEL PIC S9(4) COMP.
  03 SELDI–CUR–VNAMEL PIC S9(4) COMP.
  03 SELDFCLP PIC S9(9) COMP.
  03 SELDFCRCP PIC S9(9) COMP.

01 XSELDI.
  03 SEL–DI OCCURS 20 TIMES PIC S9(4) COMP.

01 XSELDIVNAME.
  03 SEL–DI–VNAME OCCURS 20 TIMES PIC X(80).

01 XSELDV.
  03 SEL–DV OCCURS 20 TIMES PIC X(80).

01 XSELDHVNAME.
  03 SEL–DH–VNAME OCCURS 20 TIMES PIC X(80).

01 TABLE–INDEX PIC 9(3).
01 VAR–COUNT PIC 9(2).
01 ROW–COUNT PIC 9(4).
01 NO–MORE–DATA PIC X(1) VALUE "N".
01 NULLS–ALLOWED PIC S9(9) COMP.
01 PRECISION PIC S9(9) COMP.
01 SCALE PIC S9(9) COMP.
01 DISPLAY–LENGTH PIC S9(9) COMP.
01 MAX–LENGTH PIC S9(9) COMP VALUE 80.
01 COLUMN–NAME PIC X(30).

EXEC SQL BEGIN DECLARE SECTION END–EXEC.
  01 USERNAME PIC X(20).
  01 PASSWD PIC X(20).
  01 DYN–STATEMENT PIC X(80).
EXEC SQL END DECLARE SECTION END–EXEC.
EXEC SQL INCLUDE SQLCA END–EXEC.
PROCEDURE DIVISION.
START–MAIN.
EXEC SQL
    WHENEVER SQLERROR GOTO SQL–ERROR END–EXEC.
DISPLAY "USERNAME: " WITH NO ADVANCING.
ACCEPT USERNAME.
DISPLAY "PASSWORD: " WITH NO ADVANCING.
ACCEPT PASSWORD.
EXEC SQL
    CONNECT :USERNAME IDENTIFIED BY :PASSWORD END–EXEC.
DISPLAY "CONNECTED TO ORACLE AS USER: ", USERNAME.

* Initialize the bind and select descriptors.
PERFORM INIT–BNDDSC
    VARYING TABLE–INDEX FROM 1 BY 1 UNTIL TABLE–INDEX > 20.
PERFORM INIT–SELDSC
    VARYING TABLE–INDEX FROM 1 BY 1 UNTIL TABLE–INDEX > 20.

* Get a SQL statement from the operator.
DISPLAY "ENTER SQL STATEMENT WITHOUT TERMINATOR:".
DISPLAY "">
ACCEPT DYN–STATEMENT.
DISPLAY "".

* Prepare the SQL statement and declare a cursor.
EXEC SQL
    PREPARE S1 FROM :DYN–STATEMENT END–EXEC.
EXEC SQL
    DECLARE C1 CURSOR FOR S1 END–EXEC.

* Describe any bind variables.
EXEC SQL
    DESCRIBE BIND VARIABLES FOR S1 INTO BNDDSC END–EXEC.
IF SQLDFND IN BNDDSC < 0
    DISPLAY "TOO MANY BIND VARIABLES."
    GO TO END–SQL
ELSE
    DISPLAY "NUMBER OF BIND VARIABLES: " WITH NO ADVANCING
    MOVE SQLDFND IN BNDDSC TO VAR–COUNT
    DISPLAY VAR–COUNT
    MOVE SQLDFND IN BNDDSC TO SQLDNUM IN BNDDSC END–IF.
* Replace the 0s described into the datatype fields of the bind
descriptor with 1s to avoid an Oracle "invalid datatype"
error.
MOVE 1 TO TABLE–INDEX.
FIX–BIND–TYPE.
MOVE 1 TO BNDDVTYP(TABLE–INDEX).
ADD 1 TO TABLE–INDEX.
IF TABLE–INDEX <= 20
   GO TO FIX–BIND–TYPE.

* Let the user fill in the bind variables.
IF SQLDFND IN BNDDSC = 0
   GO TO DESCRIBE–ITEMS.
MOVE 1 TO TABLE–INDEX.
GET–BIND–VAR.
DISPLAY "ENTER VALUE FOR ", BND–DH–VNAME(TABLE–INDEX).
ACCEPT BND–DV(TABLE–INDEX).
ADD 1 TO TABLE–INDEX.
IF TABLE–INDEX <= SQLDFND IN BNDDSC
   GO TO GET–BIND–VAR.

DESCRIBE–ITEMS.
* Open the cursor and describe the select-list items.
EXEC SQL
   OPEN C1 USING DESCRIPTOR BNDDSC
END–EXEC.
EXEC SQL
   DESCRIBE SELECT LIST FOR S1 INTO SELDSC
END–EXEC.

IF SQLDFND IN SELDSC < 0
   DISPLAY "TOO MANY SELECT–LIST ITEMS."
   GO TO END–SQL
ELSE
   DISPLAY "NUMBER OF SELECT–LIST ITEMS: ", SQLDFND IN SELDSC
   WITH NO ADVANCING
   MOVE SQLDFND IN SELDSC TO VAR–COUNT
   DISPLAY VAR–COUNT
   DISPLAY " "
   MOVE SQLDFND IN SELDSC TO SQLDNUM IN SELDSC
END–IF.

* Coerce the datatype of all select-list items to VARCHAR2.
IF SQLDNUM IN SELDSC > 0
   PERFORM COERC–COLUMN–TYPE
   VARYING TABLE–INDEX FROM 1 BY 1
   UNTIL TABLE–INDEX > SQLDNUM IN SELDSC
   DISPLAY " ".

Implementing Dynamic SQL Method 4  4 – 45
* Fetch each row and print each select-list value.
  IF SQLDNUM IN SELDSC > 0
      PERFORM FETCH-ROWS UNTIL NO-MORE-DATA = "Y".
      DISPLAY " "
      DISPLAY "NUMBER OF ROWS PROCESSED: " WITH NO ADVANCING.
      MOVE SQLERRD(3) TO ROW-COUNT.
      DISPLAY ROW-COUNT.
  END-EXEC.
  EXEC SQL
  CLOSE C1
  END-EXEC.
  EXEC SQL
  COMMIT WORK RELEASE
  END-EXEC.
  DISPLAY " ".
  DISPLAY "HAVE A GOOD DAY!".
  DISPLAY " ".
  STOP RUN.

SQL-ERROR.
* Display Oracle error message and code.
  DISPLAY " ".
  DISPLAY SQLERRMC.

END-EXEC.
EXEC SQL
  WHENEVER SQLERROR CONTINUE
END-EXEC.
EXEC SQL
  ROLLBACK WORK RELEASE
END-EXEC.
STOP RUN.
* PERFORMed subroutines begin here:

INIT–BNDDSC.

* Initialize the bind descriptor.
  MOVE SPACES TO BND–DH–VNAME(TABLE–INDEX).
  MOVE 80 TO BNDDH–MAX–VNAMEL(TABLE–INDEX).
  CALL "SQLADR" USING
      BND–DH–VNAME(TABLE–INDEX)
      BNDDH–VNAME(TABLE–INDEX).
  MOVE SPACES TO BND–DI–VNAME(TABLE–INDEX).
  MOVE 80 TO BNDDI–MAX–VNAMEL(TABLE–INDEX).
  CALL "SQLADR" USING
      BND–DI–VNAME(TABLE–INDEX)
      BNDDI–VNAME(TABLE–INDEX).
  MOVE SPACES TO BND–DV(TABLE–INDEX).
  MOVE 80 TO BNDDVLN(TABLE–INDEX).
  CALL "SQLADR" USING
      BND–DV(TABLE–INDEX)
      BNDDV(TABLE–INDEX).
  MOVE ZERO TO BND–DI(TABLE–INDEX).
  CALL "SQLADR" USING
      BND–DI(TABLE–INDEX)
      BNDDI(TABLE–INDEX).
  MOVE ZERO TO BNDDFMT(TABLE–INDEX).
  MOVE ZERO TO BNDDFMTL(TABLE–INDEX).
  MOVE ZERO TO BNDDFCNP(TABLE–INDEX).
  MOVE ZERO TO BNDDFCRCP(TABLE–INDEX).
  EXIT.
INIT-SELDSC.
* Initialize the select descriptor.
MOVE SPACES TO SEL-DH-VNAME(TABLE-INDEX).
MOVE 80 TO SELDH-MAX-VNAME(TABLE-INDEX).
CALL "SQLADR" USING
   SEL-DH-VNAME(TABLE-INDEX)
   SELDH-VNAME(TABLE-INDEX).
MOVE SPACES TO SEL-DI-VNAME(TABLE-INDEX).
MOVE 80 TO SELDI-MAX-VNAME(TABLE-INDEX).
CALL "SQLADR" USING
   SEL-DI-VNAME(TABLE-INDEX)
   SELDI-VNAME (TABLE-INDEX).
MOVE SPACES TO SEL-DV(TABLE-INDEX).
MOVE 80 TO SELDVNLN(TABLE-INDEX).
CALL "SQLADR" USING
   SEL-DV(TABLE-INDEX)
   SELDV(TABLE-INDEX).
MOVE ZERO TO SEL-DI(TABLE-INDEX).
CALL "SQLADR" USING
   SEL-DI(TABLE-INDEX)
   SELDI(TABLE-INDEX).
MOVE ZERO TO SELDFMT(TABLE-INDEX).
MOVE ZERO TO SELDFMTL(TABLE-INDEX).
MOVE ZERO TO SELDFCLP(TABLE-INDEX).
MOVE ZERO TO SELDFCRCP(TABLE-INDEX).
EXIT.
COERC–COLUMN–TYPE.
  * Coerce select-list datatypes to VARCHAR2.
    CALL "SQLNL" USING
      SELDVYP(TABLE–INDEX)
      SELDVYP(TABLE–INDEX)
      NULLS–ALLOWED.
  *
  * If datatype is date, lengthen to 9 characters.
    IF SELDVYP(TABLE–INDEX) = 12
      MOVE 9 TO SELDVNL(TABLE–INDEX).
    *
  * IF DATATYPE IS NUMBER, SET LENGTH TO PRECISION.
    IF SELDVYP(TABLE–INDEX) = 2
      CALL "SQLPRC" USING SELDVNL(TABLE–INDEX) PRECISION SCALE.
      MOVE 0 TO DISPLAY–LENGTH.
    IF SELDVYP(TABLE–INDEX) = 2 AND PRECISION = 0
      MOVE 40 TO DISPLAY–LENGTH.
    IF SELDVYP(TABLE–INDEX) = 2 AND PRECISION > 0
      ADD 2 TO PRECISION
      MOVE PRECISION TO DISPLAY–LENGTH.
    IF SELDVYP(TABLE–INDEX) = 2
      IF DISPLAY–LENGTH > MAX–LENGTH
        DISPLAY "COLUMN VALUE TOO LARGE FOR DATA BUFFER."
        GO TO END–SQL
      ELSE
        MOVE DISPLAY–LENGTH TO SELDVNL(TABLE–INDEX).
    *
  * Coerce datatypes to VARCHAR2.
    MOVE 1 TO SELDVYP(TABLE–INDEX).
  *
  * Display column heading.
    MOVE SEL–DH–VNAME(TABLE–INDEX) TO COLUMN–NAME.
    DISPLAY COLUMN–NAME(1:SELDVNL(TABLE–INDEX)), " "
    WITH NO ADVANCING.
    EXIT.
**FETCH-ROWS.**

* Fetch a row and print the select-list value.

```COBOL
EXEC SQL
   FETCH C1 USING DESCRIPTOR SELDSC
END-EXEC.
```

IF SQLCODE NOT = 0
   MOVE "Y" TO NO-MORE-DATA.
IF SQLCODE = 0
   PERFORM PRINT-COLUMN-VALUES
      VARYING TABLE-INDEX FROM 1 BY 1
      UNTIL TABLE-INDEX > SQLDNUM IN SELDSC
      DISPLAY " ".
EXIT.

**PRINT-COLUMN-VALUES.**

* Print a select-list value.

```COBOL
DISPLAY SEL-DV(TABLE-INDEX)(1:SELDVLN(TABLE-INDEX)), " "
   WITH NO ADVANCING.
EXIT.
```
Some details of Pro*COBOL programming vary from one system to another. This appendix is a collection of all system-specific issues regarding Pro*COBOL. References are provided, where applicable, to other sources in your document set.
System–Specific References in Chapter 1

The references in this section appear in Chapter 1 using similar order and headings.

COBOL Versions
The Pro*COBOL Precompiler supports the standard implementation of COBOL for your operating system (usually COBOL–85 or COBOL–74). Some platforms may support both COBOL implementations. Check your Oracle system–specific documentation.

Host Variables
How you declare and name host variables depends on which COBOL compiler you use. Check your COBOL user’s guide for details about declaring and naming host variables.

Declaring
Declare host variables in the Declare Section according to COBOL rules, specifying a COBOL datatype supported by Oracle. Table 1 – 2 on page 1 – 10 shows the COBOL datatypes and pseudotypes you can specify in the Declare Section. However, your COBOL implementation might not include all of them.

Naming
Host variable names must consist only of letters, digits, and hyphens, and must begin with a letter. They can be any length, but only the first 31 characters are significant. Your compiler might allow a different maximum length.

INCLUDE Statements
You can INCLUDE any file. When you precompile your Pro*COBOL program, each EXEC SQL INCLUDE statement is replaced by a copy of the file named in the statement.

If your system uses file extensions but you do not specify one, the Pro*COBOL Precompiler assumes the default extension for source files (usually COB). The default extension is system–dependent. Check your Oracle system–specific documentation.

If your system uses directories, you can set a directory path for INCLUDED files by specifying the precompiler option INCLUDE=path. You must use INCLUDE to specify a directory path for nonstandard files unless they are stored in the current directory. The syntax for specifying a directory path is system–specific. Check your Oracle system–specific documentation.
**MAXLITERAL Default**  With the MAXLITERAL precompiler option you can specify the maximum length of string literals generated by the precompiler, so that compiler limits are not exceeded. The MAXLITERAL default value is 256, but you might have to specify a lower value.

For example, if your COBOL compiler cannot handle string literals longer than 132 characters, specify “MAXLITERAL=132.” Check your COBOL compiler user’s guide. For more information about the MAXLITERAL option, see the *Programmer’s Guide to the Oracle Precompilers*.

**PIC N Clause for Multi-byte NLS Characters**  Some COBOL compilers may not support the use of the ANSI standard PIC N clause for declaring multi-byte NLS character variables. Check your COBOL user’s guide before writing source code that uses the PIC N clause to declare multi-byte NLS character variables.

---

**System–Specific Reference in Chapter 3**  The reference in this section appears in Chapter 3 under the same heading.

**Sample Programs**  All the sample programs in this chapter are available online. The names of the online files are shown on page 3 – 1. However, the exact filenames are system dependent. For more information, check your Oracle system–specific documentation.
Index

A
abbreviations, 1 – 2
ALLOCATE statement, 1 – 36
allocating cursor variables, 1 – 36
automatic logons, 1 – 45

B
bind descriptor, 4 – 4
bind SQLDA, 4 – 3
BNDDFCLP variable (SQLDA), 4 – 12
BNDDFCRCP variable (SQLDA), 4 – 12
BNDDFMT variable (SQLDA), 4 – 8
BNDDFMTL variable (SQLDA), 4 – 9
BNDDH–CUR–VNAME variable (SQLDA), 4 – 11
BNDDH–MAX–VNAME variable (SQLDA), 4 – 11
BNDDH–VNAME variable (SQLDA), 4 – 11
BNDDI variable (SQLDA), 4 – 11
BNDDI–CUR–VNAME variable (SQLDA), 4 – 12
BNDDI–MAX–VNAME variable (SQLDA), 4 – 12
BNDDI–VNAME variable (SQLDA), 4 – 12
BNDDV variable (SQLDA), 4 – 7
BNDDVLN variable (SQLDA), 4 – 8
BNDDVTYP variable (SQLDA), 4 – 10

C
character host variables
as output variables, 1 – 25, 1 – 27
handling, 1 – 24
multi-byte, 1 – 26
server handling, 1 – 25, 1 – 27
types, 1 – 24
CLOSE statement
cursor variable, 1 – 38
in dynamic SQL Method 4, 4 – 35
COBOL datatypes, 1 – 10
COBOL–74 restrictions, 1 – 7
coding area, for paragraph names, 1 – 5
coding conventions, 1 – 2
comments
ANSI SQL–style, 1 – 3
C–style, 1 – 3
embedded SQL statements, 1 – 3
composite type, 4 – 16
concurrent logons, 1 – 46
CONNECT statement
logging on to Oracle, 1 – 43
placement, 1 – 43
requirements, 1 – 43
connecting to Oracle, 1 – 43
automatically, 1 – 45
concurrently, 1 – 46
example of, 1 – 43
via SQL*Net, 1 – 46
continuation lines, syntax, 1 – 3
CONTINUE action
  (WHENEVER statement), 2 – 14
cursor cache, purpose, 2 – 17
cursor variables
  advantages, 1 – 35
  allocating, 1 – 36
  closing, 1 – 38
  declaring, 1 – 35
  error conditions, 1 – 39
  fetching from, 1 – 38
  heap memory usage, 1 – 36
  opening
    anonymous block, 1 – 38
    stored procedure, 1 – 36
  restrictions, 1 – 39
  scope, 1 – 35

declaring
cursor variables, 1 – 35
host tables, 1 – 18
host variables, 1 – 10
indicator variables, 1 – 16
multi-byte NLS variables, 1 – 26
ORAC, 2 – 18
SQLCA, 2 – 11
SQLDA, 4 – 6
VARCHAR variables, 1 – 21
default
  error handling, 2 – 13
  setting of FORMAT option, 1 – 2
  setting of LITDELIM option, 1 – 3
  setting of ORACA option, 2 – 18
DEPENDING ON clause, 1 – 18
DESCRIBE BIND VARIABLES statement, in
dynamic SQL Method 4, 4 – 27
DESCRIBE SELECT LIST statement, in dynamic
SQL Method 4, 4 – 31
descriptors
  bind descriptor, 4 – 4
  purpose, 4 – 4
  select descriptor, 4 – 4
  SQLADR subroutine, 4 – 3
dimension of host tables, 1 – 18
directory path, INCLUDE files, 1 – 9
DO action (WHENEVER statement), 2 – 14
dynamic SQL, restrictions, 1 – 5
dynamic SQL Method 4
  CLOSE statement, 4 – 35
  DECLARE CURSOR statement, 4 – 27
  DESCRIBE statement, 4 – 27, 4 – 31
  external datatypes, 4 – 14
  FETCH statement, 4 – 35
  FOR clause, 4 – 37
  host tables, 4 – 37
  internal datatypes, 4 – 14
  OPEN statement, 4 – 31
  PREPARE statement, 4 – 27
  prerequisites, 4 – 13
  purpose of descriptors, 4 – 4
  requirements, 4 – 2
  sequence of statements used, 4 – 22

D
data description
  language (DDL), embedded, 1 – 5
datatypes
  COBOL, 1 – 10
  coercing NUMBER to VARCHAR2, 4 – 17
  conversions, 1 – 31
  dealing with Oracle internal, 4 – 17
  descriptor codes, 4 – 17
  equivalencing
    description, 1 – 32
    example, 1 – 33
  external, 4 – 15
  internal, 4 – 14
  need to coerce, 4 – 17
  PL/SQL equivalents, 4 – 16
  when to reset, 4 – 17
DDL. See data definition language
DECLARE CURSOR statement,
in dynamic SQL Method 4, 4 – 27
declare section
  allowable statements, 1 – 7
  COBOL datatypes supported, 1 – 10
  defining usernames and passwords, 1 – 43
  purpose, 1 – 7
  requirements, 1 – 7
  rules for defining, 1 – 7
dynamic SQL Method 4 (continued)
  SQLDA, 4 – 4
  steps for, 4 – 21

E
elementary items, 1 – 14
embedded DDL, 1 – 5
embedded PL/SQL
  host variables, 1 – 34
  indicator variables, 1 – 34
  multi-byte NLS features, 1 – 34
  requirements, 1 – 34
  VARCHAR variables, 1 – 34
  where allowed, 1 – 34
embedded SQL statements
  associating paragraph names with, 1 – 5
  comments, 1 – 3
  continuing from one line to the next, 1 – 3
  figurative constants, 1 – 4
  referencing host tables, 1 – 19
  referencing host variables, 1 – 13
  referencing indicator variables, 1 – 16
  requirements, 1 – 4
  syntax, 1 – 4
  terminator, 1 – 6
equivalencing datatypes, 1 – 32
error conditions, cursor variable, 1 – 39
error handling
  alternatives, 2 – 2
  default, 2 – 13
  using status variables
    combinations, 2 – 6 to 2 – 8
    SQLCA, 2 – 3, 2 – 9
    SQLCODE, 2 – 2, 2 – 5
    SQLSTATE, 2 – 2, 2 – 5
error message text,
  SQLGLM subroutine, 2 – 12
error messages, maximum length, 2 – 13
error reporting
  error message text, 2 – 12
  key components of, 2 – 11
  parse error offset, 2 – 11
  rows-processed count, 2 – 11
  status codes, 2 – 11
  warning flags, 2 – 11
EXEC SQL clause, 1 – 4
external datatypes
  dynamic SQL Method 4, 4 – 14
  general, 1 – 30
F
FETCH statement
  cursor variable, 1 – 38
  in dynamic SQL Method 4, 4 – 35
figurative constants,
  embedded SQL statements, 1 – 4
file extension, for INCLUDE files, 1 – 8
flags. See warning flags
FOR clause, in dynamic SQL Method 4, 4 – 37
FORMAT option, purpose, 1 – 2
G
GOTO action (WHENEVER statement), 2 – 14
group items
  declaring, 1 – 14
  implicit VARCHAR, 1 – 22
H
heap memory,
  allocating cursor variables, 1 – 36
host tables
  declaring, 1 – 18
  dimensioning, 1 – 18
  in dynamic SQL Method 4, 4 – 37
  multi-dimensional, 1 – 18
  referencing, 1 – 19
  restrictions, 1 – 18
  support for, 1 – 12
  variable-length, 1 – 18
host variables
  declaring, 1 – 2, 1 – 7, 1 – 10
  definition, 1 – 4
  initializing, 1 – 12
  naming, 1 – 13, 1 – 15
  referencing, 1 – 13
Index – 4 Pro*COBOL Supplement to the Oracle Precompilers Guide

host variables (continued)
  restrictions, 1 – 4, 1 – 13, 1 – 15
  with PL/SQL, 1 – 34
hyphenation, of host variable names, 1 – 4

I
implicit VARCHAR, 1 – 22
INCLUDE statement
  case-sensitive operating systems, 1 – 9
  declaring the ORACA, 2 – 18
  declaring the SQLCA, 2 – 11
  declaring the SQLDA, 4 – 6
  effect of, 1 – 8
indicator tables
  example, 1 – 20
  purpose, 1 – 20
indicator variables
  association with host variables, 1 – 16
  declaring, 1 – 2, 1 – 16
  function, 1 – 16
  nulls, 1 – 34
  referencing, 1 – 16
  required size, 1 – 16
  truncated values, 1 – 34
  with PL/SQL, 1 – 34
internal datatypes
  dynamic SQL Method 4, 4 – 14
  general, 1 – 29
IS NULL operator, for testing null values, 1 – 5

L
LITDELIM option, 1 – 3
logging on
  automatically, 1 – 45
  requirements, 1 – 43

M
MAXLITERAL option, 1 – 5, A – 3
message text. See error message text

MODE option
  effects of, 1 – 24
  status variables, 2 – 2
multi-byte NLS features
  datatypes, 1 – 5, 1 – 26
  with PL/SQL, 1 – 34

N
naming
  host variables, 1 – 4
  select-list items, 4 – 4
NOT FOUND condition
  (WHENEVER statement), 2 – 14
nulls
  handling
    in dynamic SQL Method 4, 4 – 20
    indicator variables, 1 – 34
  meaning in SQL (NVL function), 1 – 5
  SQLNUL subroutine, 4 – 20
NUMBER datatype,
  using the SQLPRC subroutine with, 4 – 18
NVL function, for retrieving null values, 1 – 5

O
OPEN statement,
  in dynamic SQL Method 4, 4 – 31
opening a cursor variable, 1 – 36
operators, relational, 1 – 6
ORACA
  declaring, 2 – 18
  enabling, 2 – 18
  precompiler option, 2 – 18
  purpose, 2 – 3, 2 – 17
  Oracle Communications Area, ORACA, 2 – 17

P
paragraph names
  associating with SQL statements, 1 – 5
  coding area for, 1 – 5
parse error offset, 2 – 11
password
defining, 1 – 43
hardcoding, 1 – 43
PL/SQL
datatype equivalents, 4 – 16
embedded, 1 – 34
opening a cursor variable
anonymous block, 1 – 38
stored procedure, 1 – 36
precision
definition of, 4 – 18
using SQLPRC to extract, 4 – 18
when not specified, 4 – 18
precompiler options
ASSUME_SQLCODE, 2 – 4, 2 – 7
FORMAT, 1 – 2
LITDELIM, 1 – 3
MAXLITERAL, 1 – 5, A – 3
MODE, 1 – 24, 2 – 2, 2 – 3
ORACA, 2 – 18
SQLCHECK, 1 – 34
PREPARE statement,
in dynamic SQL Method 4, 4 – 27
programming guidelines, 1 – 2

R
REDEFINES clause
purpose, 1 – 6
restrictions, 1 – 6
reference cursor. See cursor variables
referring
host tables, 1 – 19
host variables, 1 – 13
indicator variables, 1 – 16
VARCHAR variables, 1 – 22
relational operators, 1 – 6
restrictions
COBOL–74, 1 – 7
cursor variables, 1 – 39
dynamic SQL, 1 – 5
host tables, 1 – 18
host variables, 1 – 13, 1 – 15
declaring, 1 – 13
naming, 1 – 4
referencing, 1 – 15
REDEFINES clause, 1 – 6
SQLGLM subroutine, 2 – 13
SQLIEM subroutine, 2 – 13
rows–processed count, 2 – 11

S
sample programs
calling a stored procedure, 3 – 30
cursor operations, 3 – 5
cursor variables
PL/SQL source, 1 – 39
Pro*COBOL source, 1 – 40
datatype equivalencing, 3 – 11
dynamic SQL Method 1, 3 – 19
dynamic SQL Method 2, 3 – 22
dynamic SQL Method 3, 3 – 26
dynamic SQL Method 4, 4 – 42
fetching in batches, 3 – 8
list, 3 – 1
Oracle Forms user exit, 3 – 16
simple query, 3 – 2
scalar type, 4 – 16
scale
definition, 4 – 18
extracting with SQLPRC, 4 – 18
when negative, 4 – 18
scope
cursor variables, 1 – 35
WHENEVER statement, 2 – 16
SELDFCLP variable (SQLDA), 4 – 12
SELDFCRCP variable (SQLDA), 4 – 12
SELDFMT variable (SQLDA), 4 – 8
SELDFTML variable (SQLDA), 4 – 9
SELDH–CUR–VNAMEL
variable (SQLDA), 4 – 11
SELDH–MAX–VNAMEL
variable (SQLDA), 4 – 11
SELDH–VNAME variable (SQLDA), 4 – 11
SELDI variable (SQLDA), 4 – 11
Index – 6 Pro*COBOL Supplement to the Oracle Precompilers Guide

SELDI–CUR–VNAMEL
variable (SQLDA), 4 – 12
SELDI–MAX–VNAMEL
variable (SQLDA), 4 – 12
SELDI–VNAME variable (SQLDA), 4 – 12
SELDV variable (SQLDA), 4 – 7
SELDVLN variable (SQLDA), 4 – 8
SELDVTYP variable (SQLDA), 4 – 10
select descriptor, 4 – 4
select SQLDA, purpose of, 4 – 3
select–list items, naming, 4 – 4
SQL Descriptor Area. See SQLDA
SQL*Net, for concurrent logons, 1 – 46
SQLADR subroutine
example, 4 – 24
parameters, 4 – 13
storing buffer addresses, 4 – 3
syntax, 4 – 13
SQLCA status variable
data structure, 2 – 10
declaring, 2 – 11
effect of MODE option, 2 – 3
explicit versus implicit checking, 2 – 3
purpose, 2 – 9
SQLCHECK option, 1 – 34
SQLCODE status variable
ASSUME_SQLCODE option, 2 – 4
declaring, 2 – 5
description, 2 – 2
effect of MODE option, 2 – 3
SQL92 deprecated feature, 2 – 2
usage, 2 – 3
SQLDA
BNDDFCLP variable, 4 – 12
BNDDFCRCP variable, 4 – 12
BNDDFMT variable, 4 – 8
BNDDFMTL variable, 4 – 9
BNDDH–CUR–VNAMEL variable, 4 – 11
BNDDH–MAX–VNAMEL variable, 4 – 11
BNDDH–VNAME variable, 4 – 11
BNDDI variable, 4 – 11
BNDDI–CUR–VNAMEL variable, 4 – 12
BNDDI–MAX–VNAMEL variable, 4 – 12
BNDDI–VNAME variable, 4 – 12
BNDDV variable, 4 – 7
BNDDVLN variable, 4 – 8
BNDDVTYP variable, 4 – 10
declaring, 4 – 6
example, 4 – 6
purpose, 4 – 4
SELDFCLP variable, 4 – 12
SELDFCRCP variable, 4 – 12
SELDFMT variable, 4 – 8
SELDFMTL variable, 4 – 9
SELDH–CUR–VNAMEL variable, 4 – 11
SELDH–MAX–VNAMEL variable, 4 – 11
SELDH–VNAME variable, 4 – 11
SELDI variable, 4 – 11
SELDI–CUR–VNAMEL variable, 4 – 12
SELDI–MAX–VNAMEL variable, 4 – 12
SELDI–VNAME variable, 4 – 12
SELDV variable, 4 – 7
SELDVLN variable, 4 – 8
SELDVTYP variable, 4 – 10
SQLDFND variable, 4 – 7
SQLDFND variable (SQLDA), 4 – 7
SQLDNUM variable, 4 – 7
SQLDNUM variable (SQLDA), 4 – 7
SQLERROR condition
(WHENEVER statement), 2 – 14
SQLGLM subroutine
example, 2 – 13
parameters, 2 – 12
purpose, 2 – 12
restrictions, 2 – 13
syntax, 2 – 12
SQLIEM subroutine, restrictions, 2 – 13
SQLNUL subroutine
example, 4 – 20
parameters, 4 – 20
purpose, 4 – 20
syntax, 4 – 20
SQLPR2 subroutine, 4 – 19
SQLPRC subroutine
example, 4 – 19
parameters, 4 – 18
purpose, 4 – 18
syntax, 4 – 18
SQLSTATE status variable
  declaring, 2 – 5
  effect of MODE option, 2 – 3
  usage, 2 – 3
SQLWARNING condition
  (WHENEVER statement), 2 – 14
status codes for error reporting, 2 – 11
STOP action (WHENEVER statement), 2 – 14
stored procedure
  opening a cursor, 1 – 36, 1 – 39
  sample programs, 1 – 39, 3 – 30
string literals, continuing to the next line, 1 – 3
syntax
  continuation lines, 1 – 3
  embedded SQL statements, 1 – 4
  SQLADR subroutine, 4 – 13
  SQLGLM subroutine, 2 – 12
  SQLNUL subroutine, 4 – 20
  SQLPRC, 4 – 18

T
  tables. See host tables
terminator for
  embedded SQL statements, 1 – 6
terminating values, indicator variables, 1 – 34
Trusted Oracle7, 4 – 16

U
  username
    defining, 1 – 43
    hardcoding, 1 – 43

V
  VALUE clause,
    initializing host variables, 1 – 12
  VAR statement, syntax for, 1 – 32
  VARCHAR variables
    advantages, 1 – 28
    as input variables, 1 – 28
    as output variables, 1 – 28
    declaring, 1 – 21
    implicit group items, 1 – 22
    length element, 1 – 21
    maximum length, 1 – 21
    referencing, 1 – 22
    server handling, 1 – 28
    string element, 1 – 21
    structure, 1 – 21
    versus fixed–length strings, 1 – 28
    with PL/SQL, 1 – 34
  VARYING keyword,
    versus VARYING phrase, 1 – 21

W
  warning flags for error reporting, 2 – 11
WHENEVER statement
  CONTINUE action, 2 – 14
  DO action, 2 – 14
  example, 2 – 15
  GOTO action, 2 – 14
  NOT FOUND condition, 2 – 14
  purpose, 2 – 13
  scope, 2 – 16
  SQLERROR condition, 2 – 14
  SQLWARNING condition, 2 – 14
  STOP action, 2 – 14
  syntax, 2 – 13
Reader’s Comment Form

Pro*COBOL® Supplement to the Oracle® Precompilers Guide
Part No. A42524–1

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.
Pro*COBOL® Supplement to the Oracle® Precompilers Guide

Release 1.8