Oracle9*i* PL/SQL: A Developer's Guide

BULUSU LAKSHMAN

Apress™

Oracle9*i* PL/SQL: A Developer's Guide

ISBN (pbk): 1-59059-049-X

Printed and bound in the United States of America 12345678910

Technical Reviewer: Martin Reid
Editorial Directors: Dan Appleman, Gary Cornell, Jason Gilmore, Simon Hayes, Karen Watterson, John Zukowski
Managing Editor: Grace Wong
Project Manager and Developmental Editor: Tracy Brown Collins
Copy Editors: Nicole LeClerc, Ami Knox
Production Editor: Laura Cheu
Compositor and Artist: Impressions Book and Journal Services, Inc.
Cover Designer: Kurt Krames
Indexer: Valerie Robbins
Marketing Manager: Stephanie Rodriguez
Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States, phone 1-800-SPRINGER, email orders@springer-ny.com, or visit http://www.springer-ny.com.

Outside the United States, fax +49 6221 345229, email orders@springer.de, or visit http://www.springer.de.

For information on translations, please contact Apress directly at 2560 9th Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, email info@apress.com, or visit http://www.apress.com.

The source code for this book is available to readers at http://www.apress.com in the Downloads section.

CHAPTER 2

Cursors

Think of this chapter as Cursors 101 and 201 combined. I start with a quick overview of cursors and how to use them in PL/SQL. Next, I cover the methods for processing multirow resultsets with cursors. Then I tackle cursor variables and their uses, and I wrap up with a discussion of Oracle9*i*'s new cursor expressions.

I illustrate the concept of cursors, cursor variables, and cursor expressions by taking into account an organizational hierarchy system. The case study I present uses the data model shown in Figure 2-1. The schema objects to be created are listed in Appendix A.

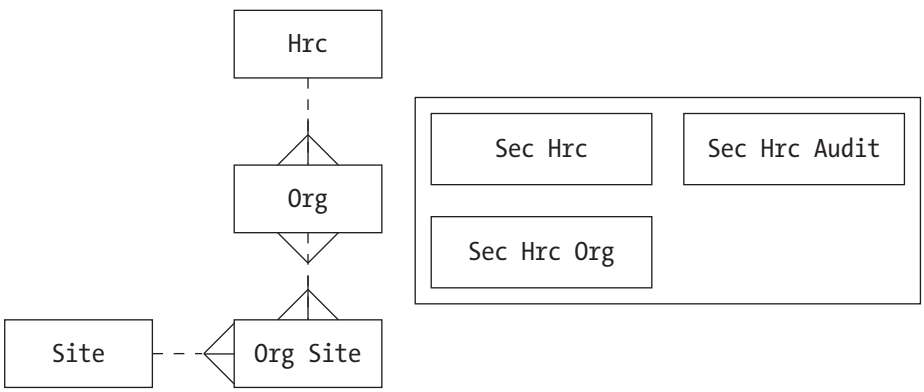

Figure 2-1. The data model of an organizational hierarchy system

Introducing Cursors

As described in Chapter 1, PL/SQL interacts with SQL by combining SQL statements with PL/SQL constructs inside a PL/SQL block. Cursors are one more PL/SQL feature that exhibits interaction with SQL by using SQL within PL/SQL. A *cursor* is a handle to a work area that holds the resultset of a multirow SQL query. Oracle opens a work area to hold the resultset of multirow queries. A cursor gives this work area a name and can be used to process the rows returned by the multirow query.

There are two types of cursors: explicit and implicit. The cursors defined earlier for handling multirow resultsets are called *explicit* cursors. *Implicit* cursors are those defined by Oracle and are associated with single-row SELECT . . . INTO statements and INSERT, UPDATE, and DELETE statements. These statements are also executed within the context of a work area and the Oracle PL/SQL engine automatically opens a cursor that points to this work area. This work area identifies the rows to be modified with the INSERT, UPDATE, DELETE, or SELECT . . . INTO statement. There's no need to declare the cursor explicitly, hence the name "implicit."

This section begins with a discussion of explicit cursors and then moves on to cover implicit cursors in detail.

Explicit Cursors

In an explicit cursor's definition, the cursor name is explicitly associated with a SELECT statement. This is done using the PL/SQL CURSOR . . . IS SELECT . . . statement. Explicit cursors can be associated with a SELECT statement only.

You can use an explicit cursor to process multirow queries, including queries that fetch one row.

Defining an Explicit Cursor

You declare an explicit cursor using the CURSOR . . . IS SELECT . . . statement in PL/SQL. Here's the syntax:

```
CURSOR cursor_name IS
     SELECT_statement ;
```

where cursor_name is the name of the cursor and SELECT_statement is any valid SQL SELECT statement without the INTO clause.

When you use a PL/SQL block, you need to declare an explicit cursor in the declaration section after the DECLARE keyword. The following is an example of an explicit cursor:

```
DECLARE
     CURSOR csr_org IS
          SELECT h.hrc_descr, o.org_short_name
          FROM   org_tab o, hrc_tab h
          WHERE o.hrc_code = h.hrc_code
           ORDER by 2;
           v_hrc_descr VARCHAR2(20);
```

```
            v_org_short_name VARCHAR2(30);
   BEGIN
     /* ... <Process the cursor resultset> ... */
     null;
  END;
/
```

When naming a cursor, you should follow the standard PL/SQL variable naming conventions. Other declarations can follow or precede a CURSOR declaration. The order of declaring cursors and other variables is immaterial. The SELECT statement associated with a cursor can't contain an INTO clause. It may, however, have GROUP BY and ORDER clauses, as well as joins and set operators such as UNION, INTERSECT, and MINUS. The scope of a cursor is the PL/SQL block in which it is defined or any of its nested blocks. Enclosing (outer) blocks can't reference a cursor defined within them.

Using an Explicit Cursor

Once you've defined a cursor, you can use it for processing the rows contained in the resultset. Here are the steps:

1. Open the cursor.

2. Fetch the results into a PL/SQL record or individual PL/SQL variables.

3. Close the cursor.

There are two ways to use an explicit cursor once it has been defined: using OPEN, FETCH, and CLOSE, and using a cursor FOR LOOP. You can do this in the executable section of a PL/SQL block in between BEGIN and END.

Using OPEN, FETCH, and CLOSE

After declaring the cursor, you have to open it as follows:

```
OPEN cursor_name;
```

where cursor_name is the name of the declared cursor.

Here's an example that illustrates opening the cursor csr_org declared previously:

```
DECLARE
     CURSOR csr_org IS
          SELECT h.hrc_descr, o.org_short_name
          FROM   org_tab o, hrc_tab h
          WHERE o.hrc_code = h.hrc_code
           ORDER by 2;
           v_hrc_descr VARCHAR2(20);
           v_org_short_name VARCHAR2(30);
   BEGIN
        OPEN csr_org;
      /* . . . <Process the cursor resultset> . . . */
     null;
  END;
/
```

Once opened, the resultset returned by the associated SELECT statement is determined and fixed. This is often termed the *active set* of rows. Also, the cursor pointer points to the first row in the active set.

CAUTION *Don't open an already opened cursor. This raises the predefined PL/SQL exception CURSOR_ALREADY_OPEN.*

The next step is to fetch the cursor into PL/SQL variables. This retrieves individual rows of data into the PL/SQL variables for processing. You fetch a cursor using the FETCH statement, which has four forms. Here's the syntax:

```
FETCH cursor_name INTO var1, var2, . . . , varN;
```

or

```
FETCH cursor_name INTO cursor_name%ROWTYPE;
```

or

```
FETCH cursor_name INTO table_name%ROWTYPE;
```

or

```
FETCH cursor_name INTO record_name;
```

Here, var1, var2, and varN represent PL/SQL variables having data types identical to the cursor SELECT columns. cursor_name%ROWTYPE represents a PL/SQL record type with attributes implicitly defined that are identical to the cursor SELECT. In this case, the record type needs to be defined explicitly. table_name%ROWTYPE represents a similar record type, but one that has attributes as the column names of the table identified by table_name. In this case, the columns in table_name should exactly match in number and data type the columns in the cursor SELECT statement. Lastly, record_name is a variable of a PL/SQL record type that's explicitly defined. In this case also, the number and data types of the individual attributes of the record should be a one-to-one match with the columns in the cursor SELECT.

Here's an example that extends the previous example of csr_org to fetching rows:

```
DECLARE
     CURSOR csr_org IS
          SELECT h.hrc_descr, o.org_short_name
          FROM   org_tab o, hrc_tab h
          WHERE o.hrc_code = h.hrc_code
           ORDER by 2;
           v_hrc_descr VARCHAR2(20);
           v_org_short_name VARCHAR2(30);
   BEGIN
        OPEN csr_org;
        FETCH csr_org INTO v_hrc_descr, v_org_short_name;
       -- This fetch fetches the first row in the active set.
      null;
  END;
/
```

Here, the first row in the active set is fetched into two PL/SQL variables named v_hrc_descr and v_org_short_name. Once the first row in the active set is fetched, it's up to the program to process the data in whatever manner desired.

Alternatively, you can declare a record variable of type cursor_name%ROWTYPE and then fetch the cursor into it. This is recommended and eliminates the use of multiple variables. Here's an example:

```
DECLARE
     CURSOR csr_org IS
          SELECT h.hrc_descr, o.org_short_name
          FROM   org_tab o, hrc_tab h
          WHERE o.hrc_code = h.hrc_code
           ORDER by 2;
```

```
            v_org_rec csr_org%ROWTYPE;
   BEGIN
         OPEN csr_org;
         FETCH csr_org INTO v_org_rec;
        -- This fetch fetches the first row in the active set.
       null;
  END;
/
```

In this case, you can access the individual columns in the record type using the same column names as in the CURSOR SELECT statement.

Note that a single FETCH fetches only one row at a time. The first FETCH statement fetches the very first row, the second FETCH statement fetches the second row, and so on. To fetch all the rows, you have to use a single FETCH statement in a loop. Each iteration of FETCH advances the cursor pointer to the next row. Once fetched, the individual rows can be processed in whatever manner desired. You can fetch sets of rows at one time by repeating the definition of the FETCH statement. For example, to fetch two rows at a time, just repeat the FETCH statement twice.

TIP *A single FETCH always fetches only one row (the current row) from the active set. To fetch multiple rows, use the FETCH statement in a loop.*

You can fetch a cursor only after you open it. The number and data types of the individual variables should exactly match the columns list in the cursor SELECT statement. In the case when the cursor is fetched into a record type (either cursor_name%ROWTYPE, table_name%ROWTYPE, or record_name), the number and data type of each attribute in the record should exactly match the columns list of the cursor SELECT statement.

CAUTION *Don't fetch from an already closed cursor. Doing so results in an "ORA-01001: invalid cursor" error or an "ORA-01002: Fetch out of sequence" error.*

TIP *Always fetch into a record type of cursor_name%ROWTYPE, or, at least fetch into a record type compatible with the cursor SELECT rather than into individual variables. This is less error-prone and also improves program readability.*

Once the processing of the rows is completed, you have to close the cursor. This frees the resources allocated to the cursor, such as the memory required for storing the active set. You close a cursor using the CLOSE statement. Here's the syntax:

```
CLOSE cursor_name;
```

TIP *You should always close an opened cursor. If you don't close it, it may result in a "too many open cursors" error. The maximum number of open cursors is determined by the init.ora initialization parameter open_cursors. The default value for this parameter in Oracle9i is 50. Don't close an already closed cursor.*

TIP *The CLOSE statement should always appear after the FETCH statement. When you use a loop to fetch the rows from a cursor, you should insert the CLOSE statement after you close the loop. Otherwise, it results in an illegal fetch.*

To determine if a cursor is already open or not, you have to use *cursor attributes.* I discuss cursor attributes in the section "Explicit Cursor Attributes."

Here's a complete example of using the csr_org cursor involving all the steps previously described:

```
DECLARE
/* Declare a cursor explicitly */
     CURSOR csr_org IS
          SELECT h.hrc_descr, o.org_short_name
```

```
          FROM   org_tab o, hrc_tab h
          WHERE o.hrc_code = h.hrc_code
           ORDER by 2;
           v_org_rec csr_org%ROWTYPE;
   BEGIN
       /* Open the cursor */
        OPEN csr_org;
        /* Format headings */
        dbms_output.put_line('Organization Details with Hierarchy');
        dbms_output.put_line('-----------------------');
        dbms_output.put_line(rpad('Hierarchy',20,' ')||' '||
                                          rpad('Organization',30,' '));
        dbms_output.put_line(rpad('-',20,'-')||' '||rpad('-',30,'-'));
       /* Fetch from the cursor resultset in a loop and display the results
*/
        LOOP
             FETCH csr_org INTO v_org_rec;
             EXIT WHEN csr_org%NOTFOUND;
             dbms_output.put_line(rpad(v_org_rec.hrc_descr,20,' ')||' '||
                              rpad(v_org_rec.org_short_name,30,' '));
       END LOOP;
     /* CLose the cursor */
       CLOSE csr_org;
 END;
/
```

Here's the output of this program:

```
Organization Details with Hierarchy
-------------------------------------------
Hierarchy            Organization
-------------------  -------------------------
CEO/COO              Office of CEO ABC Inc.
CEO/COO              Office of CEO DataPro Inc.
CEO/COO              Office of CEO XYZ Inc.
VP                   Office of VP Mktg ABC Inc.
VP                   Office of VP Sales ABC Inc.
VP                   Office of VP Tech ABC Inc.

PL/SQL procedure successfully completed.
```

The code in this program opens the cursor, fetches the rows one by one until no more rows are found, displays the information in a formatted manner, and then closes the cursor. The one thing to note here is the EXIT condition for the cursor loop. This is determined by a cursor attribute %NOTFOUND, which is defined in the statement

```
EXIT WHEN csr_org%NOTFOUND;
```

%NOTFOUND returns a boolean true when the last row has been fetched and there are no more rows left in the active set. This tells PL/SQL to stop executing the fetch loop and exit the cursor loop. Fetching past the last row results in an "ORA-01002: Fetch out of sequence" error.

TIP *Always check for the attribute %NOTFOUND* immediately after *the FETCH statement to terminate a cursor FETCH loop normally. When you use multiple FETCH statements to fetch a row set at a time, specify the EXIT WHEN cursor_name%NOTFOUND condition* immediately after every FETCH statement. *This ensures avoidance of the ORA-01002 error.*

The program in this example used a simple LOOP . . . END LOOP to fetch rows from a cursor. This serves the purpose very well. However, a WHILE LOOP can replace the simple LOOP if desired. Using a WHILE LOOP, however, demands greater caution in using the FETCH statement and specifying the EXIT condition for the loop. Here are the rules of the thumb to keep in mind when you use WHILE LOOP for FETCHing:

- FETCH once before the beginning of the LOOP.
- Specify a condition of cursor_name%FOUND as the condition of the WHILE LOOP.
- Inside the loop, process the row first and then include a second FETCH after the processing logic.
- Don't specify an EXIT condition after the FETCH statement inside the LOOP, such as EXIT WHEN . . .

Here's the same example presented previously rewritten using a WHILE fetch loop:

```
DECLARE
     CURSOR csr_org IS
          SELECT h.hrc_descr, o.org_short_name
          FROM   org_tab o, hrc_tab h
          WHERE o.hrc_code = h.hrc_code
           ORDER by 2;
           v_org_rec csr_org%ROWTYPE;
   BEGIN
        OPEN csr_org;
        dbms_output.put_line('Organization Details with Hierarchy');
        dbms_output.put_line('-----------------------');
        dbms_output.put_line(rpad('Hierarchy',20,' ')||' '||
                                          rpad('Organization',30,' '));
        dbms_output.put_line(rpad('-',20,'-')||' '||rpad('-',30,'-'));
        FETCH csr_org INTO v_org_rec;
        WHILE (csr_org%FOUND) LOOP
              dbms_output.put_line(rpad(v_org_rec.hrc_descr,20,' ')||' '||
                                  rpad(v_org_rec.org_short_name,30,' '));
              FETCH csr_org INTO v_org_rec;
       END LOOP;
       CLOSE csr_org;
 END;
/
```

The following points are worth noting:

- The first FETCH before the beginning of the WHILE LOOP is necessary to make sure the condition for the WHILE LOOP evaluates to TRUE. You do this by using the %FOUND cursor attribute, which evaluates to TRUE if at least one row is present in the active set.

- If the active set contains no rows, the WHILE LOOP isn't executed. This is in contrast to a simple LOOP . . . END LOOP, where the control enters the loop even before the first fetch.

- The processing of the data fetched by the first FETCH (outside the WHILE LOOP) is done first and then the successive row(s) are fetched.

- There is no need for an EXIT condition after the second FETCH (inside the loop).

Using a Cursor FOR LOOP

You can also use a declared cursor using a cursor FOR LOOP instead of explicitly using OPEN, FETCH, and CLOSE. A cursor FOR LOOP takes care of cursor processing using an implicit OPEN FETCH and CLOSE. Here are the steps:

1. Declare a cursor FOR LOOP. Here's an example:

```
FOR idx in cursor_name LOOP
 . . .
 . . .
END LOOP;
```

   Here, cursor_name is the name of the cursor and idx is the index of the cursor FOR LOOP and is of type cursor_name%ROWTYPE.

TIP *Using a cursor FOR LOOP doesn't make the cursor an implicit cursor. It's still an explicit cursor and has to be declared explicitly.*

2. Process the data in the active set. Here's the example of the csr_org cursor modified using a cursor FOR LOOP:

```
DECLARE
     CURSOR csr_org IS
          SELECT h.hrc_descr, o.org_short_name
          FROM   org_tab o, hrc_tab h
          WHERE o.hrc_code = h.hrc_code
           ORDER by 2;
BEGIN
       dbms_output.put_line('Organization Details with Hierarchy');
       dbms_output.put_line('------------------------');
       dbms_output.put_line(rpad('Hierarchy',20,' ')||' '||
                                          rpad('Organization',30,' '));
       dbms_output.put_line(rpad('-',20,'-')||' '||rpad('-',30,'-'));
       FOR idx IN csr_org LOOP
              dbms_output.put_line(rpad(idx.hrc_descr,20,' ')||' '||
```

```
                        rpad(idx.org_short_name,30,' '));
    END LOOP;
END;
/
```

The following points are worth noting:

- The index of the cursor FOR LOOP isn't declared. It's implicitly declared by the PL/SQL compiler as type csr_org%ROWTYPE.

TIP *Never declare the index of a cursor FOR LOOP.*

- You can access the individual columns in the cursor SELECT using the "." (dot) notation of accessing record type attributes by succeeding the index name with a dot followed by the column name in the cursor SELECT.

- There is no need to OPEN, FETCH, and CLOSE the cursor.

TIP *An important use of the cursor FOR LOOP is when you process* all *the rows in a cursor unconditionally. This is a recommended practice and is in contrast to the conventional method of OPEN, FETCH, and CLOSE, which is used to process some of the rows or to skip some rows on a certain condition.*

Avoiding Declaration of an Explicit Cursor with a Cursor FOR LOOP

In the earlier example, although the cursor FOR LOOP was used, the cursor csr_org was still declared in the declaration section of the PL/SQL block. However, you can wholly specify the cursor SELECT in the specification of the cursor FOR LOOP itself instead of an explicit declaration. This improves readability and is less error-prone. Here's the csr_org cursor rewritten in this way:

```
BEGIN
            FOR idx in (SELECT h.hrc_descr, o.org_short_name
                                FROM   org_tab o, hrc_tab h
                                 WHERE o.hrc_code = h.hrc_code ORDER by 2) LOOP
                 dbms_output.put_line(rpad(idx.hrc_descr,20,' ')||' '||
                                 rpad(idx.org_short_name,30,' '));
            END LOOP;
END;
/
```

Specifying a cursor as presented in this code still comes under the explicit category, as you have to specify the cursor SELECT explicitly.

TIP *Always avoid declaration of cursors in the declaration and specify them in the cursor FOR LOOP itself when dealing with cursors to process all of the rows unconditionally.*

Explicit Cursor Attributes

Every explicit cursor has four attributes associated with it that you can use to determine whether a cursor is open or not, whether a fetch yielded a row or not, and how many rows have been fetched so far. Table 2-1 lists these attributes.

Table 2-1. Explicit Cursor Attributes

ATTRIBUTE	USE
%FOUND	Indicates whether a FETCH yielded a row or not
%ISOPEN	Indicates whether a cursor is OPEN or not
%NOTFOUND	Indicates if a FETCH failed or if there are no more rows to fetch
%ROWCOUNT	Indicates the number of rows fetched so far

To use these four cursor attributes, you prefix the cursor name with the corresponding attribute. For example, for the csr_org cursor defined earlier, these four attributes can be referenced as csr_org%FOUND, csr_org%ISOPEN, csr%NOTFOUND, and csr%ROWCOUNT. The %FOUND, %ISOPEN, and %NOTFOUND attributes return a boolean TRUE or FALSE, and the %ROWCOUNT attribute returns a numeric value. The following sections describe these attributes in more detail.

%FOUND

You use %FOUND to determine whether a FETCH returned a row or not. You should use it after a cursor is opened, and it returns a value of TRUE if the immediate FETCH yielded a row, and a value of FALSE if the immediate FETCH did not fetch any row. Using %FOUND before opening a cursor or after closing a cursor raises the error "ORA-01001: invalid cursor" or the predefined exception INVALID_CURSOR.

I presented an example of using %FOUND during the discussion of using the WHILE LOOP to fetch multiple rows. Here's the same example repeated for illustration:

```
DECLARE
     CURSOR csr_org IS
          SELECT h.hrc_descr, o.org_short_name
          FROM   org_tab o, hrc_tab h
          WHERE o.hrc_code = h.hrc_code
           ORDER by 2;
           v_org_rec csr_org%ROWTYPE;
   BEGIN
        OPEN csr_org;
        dbms_output.put_line('Organization Details with Hierarchy');
        dbms_output.put_line('-----------------------');
         dbms_output.put_line(rpad('Hierarchy',20,' ')||' ||
                                          rpad('Organization',30,' '));
        dbms_output.put_line(rpad('-',20,'-')||' ||rpad('-',30,'-'));
        FETCH csr_org INTO v_org_rec;
        WHILE (csr_org%FOUND) LOOP
               dbms_output.put_line(rpad(v_org_rec.hrc_descr,20,' ')||' ||
                              rpad(v_org_rec.org_short_name,30,' '));
               FETCH csr_org INTO v_org_rec;
       END LOOP;
       CLOSE csr_org;
 END;
/
```

The following points are worth noting regarding the statement

```
WHILE (csr_org%FOUND) LOOP
```

- The statement appears after the first FETCH statement, and it should always appear after a FETCH statement. If %NOTFOUND is referenced before the first FETCH, it returns NULL.

- The condition csr_org%FOUND evaluates to TRUE if the first FETCH returned a row; otherwise, it evaluates to FALSE and the WHILE LOOP is never executed.

%ISOPEN

You use %ISOPEN to check if a cursor is already open or not. You use it to prevent an already opened cursor from opening or an already closed cursor from closing. It returns a value of TRUE if the referenced cursor is open; otherwise, it returns FALSE. Here's the previous example modified to use the %ISOPEN attribute:

```
DECLARE
     CURSOR csr_org IS
          SELECT h.hrc_descr, o.org_short_name
          FROM   org_tab o, hrc_tab h
          WHERE o.hrc_code = h.hrc_code
           ORDER by 2;
           v_org_rec csr_org%ROWTYPE;
   BEGIN
        IF (NOT csr_org%ISOPEN) THEN
          OPEN csr_org;
        END IF;
        dbms_output.put_line('Organization Details with Hierarchy');
        dbms_output.put_line('------------------------');
        dbms_output.put_line(rpad('Hierarchy',20,' ')||' '||
                                          rpad('Organization',30,' '));
        dbms_output.put_line(rpad('-',20,'-')||' '||rpad('-',30,'-'));
        FETCH csr_org INTO v_org_rec;
        WHILE (csr_org%FOUND) LOOP
              dbms_output.put_line(rpad(v_org_rec.hrc_descr,20,' ')||' '||
                              rpad(v_org_rec.org_short_name,30,' '));
              FETCH csr_org INTO v_org_rec;
       END LOOP;
       IF (csr_org%ISOPEN) THEN
         CLOSE csr_org;
       END IF;
 END;
/
```

Note the following points about %ISOPEN:

- csr_org%ISOPEN is negated in the beginning to check that the cursor isn't already open.
- At the end, the cursor csr_org is closed only if it's open.
- %ISOPEN can be referenced after a cursor is closed, and it returns FALSE in this case.

%NOTFOUND

You use %NOTFOUND to determine if a FETCH resulted in no rows (i.e., the FETCH failed) or there are no more rows to FETCH. It returns a value of TRUE if the immediate FETCH yielded no row and a value of FALSE if the immediate FETCH resulted in one row. Using %NOTFOUND before opening a cursor or after a cursor is closed raises the error "ORA-01001: invalid cursor" or the predefined exception INVALID_CURSOR. I presented an example of using %NOTFOUND during the discussion of using the simple LOOP to fetch multiple rows. Here's the same example repeated for illustration:

```
DECLARE
     CURSOR csr_org IS
          SELECT h.hrc_descr, o.org_short_name
          FROM   org_tab o, hrc_tab h
          WHERE o.hrc_code = h.hrc_code
           ORDER by 2;
           v_org_rec csr_org%ROWTYPE;
   BEGIN
        OPEN csr_org;
        dbms_output.put_line('Organization Details with Hierarchy');
        dbms_output.put_line('------------------------');
        dbms_output.put_line(rpad('Hierarchy',20,' )||' '||
                                          rpad('Organization',30,' '));
        dbms_output.put_line(rpad('-',20,'-')||' '||rpad('-',30,'-'));
        LOOP
             FETCH csr_org INTO v_org_rec;
             EXIT WHEN csr_org%NOTFOUND;
             dbms_output.put_line(rpad(v_org_rec.hrc_descr,20,' )||' '||
```

```
                              rpad(v_org_rec.org_short_name,30,' '));
        END LOOP;
        CLOSE csr_org;
 END;
/
```

The following points are worth noting:

- Note the statement

  ```
  EXIT WHEN csr_org%NOTFOUND;.
  ```

  It appears after the first FETCH statement, and it should always appear after a FETCH statement. If %NOTFOUND is referenced before the first FETCH or after a cursor is opened, it returns NULL.

- The condition csr_org%NOTFOUND is used as the EXIT condition for the loop. It evaluates to TRUE if the first FETCH didn't return a row and the loop is exited. If the first FETCH resulted in at least one row, it evaluates to FALSE and the loop is executed until the last row is fetched. After the last row is fetched, %NOTFOUND evaluates to TRUE and the loop is exited.

%ROWCOUNT

You use %ROWCOUNT to determine the number of rows fetched from a cursor. It returns 1 after the first fetch and is incremented by 1 after every successful fetch. It can be referenced after a cursor is opened or before the first fetch and returns zero in both cases. Using %ROWCOUNT before opening a cursor or after closing a cursor raises the error "ORA-01001: invalid cursor" or the predefined exception INVALID_CURSOR. The best use of this attribute is in a cursor FOR LOOP to determine the number of rows returned by the cursor. Since a cursor FOR LOOP is used to process *all* the rows of the cursor unconditionally, the value of this attribute after the cursor FOR LOOP is executed gives the total number of rows returned by the cursor.

In the following example, I've modified the cursor FOR LOOP presented earlier to include %ROWCOUNT:

```
DECLARE
      CURSOR csr_org IS
            SELECT h.hrc_descr, o.org_short_name
            FROM   org_tab o, hrc_tab h
            WHERE o.hrc_code = h.hrc_code
             ORDER by 2;
```

```
         num_total_rows NUMBER;
BEGIN
        dbms_output.put_line('Organization Details with Hierarchy');
        dbms_output.put_line('------------------------');
        dbms_output.put_line(rpad('Hierarchy',20,' ')||' '||
                                              rpad('Organization',30,' '));
        dbms_output.put_line(rpad('-',20,'-')||' '||rpad('-',30,'-'));
        FOR idx IN csr_org LOOP
                dbms_output.put_line(rpad(idx.hrc_descr,20,' ')||' '||
                                rpad(idx.org_short_name,30,' '));
                 num_total_rows := csr_org%ROWCOUNT;
        END LOOP;
        IF num_total_rows > 0 THEN
          dbms_output.new_line;
          dbms_output.put_line('Total Organizations = '||to_char(num_total_rows));
        END IF;
END;
/
```

Here's the output of this program:

```
Organization Details with Hierarchy
-------------------------------------------
Hierarchy            Organization
-------------------  ------------------------
CEO/COO              Office of CEO ABC Inc.
CEO/COO              Office of CEO DataPro Inc.
CEO/COO              Office of CEO XYZ Inc.
VP                   Office of VP Mktg ABC Inc.
VP                   Office of VP Sales ABC Inc.
VP                   Office of VP Tech ABC Inc.
Total Organizations = 6

PL/SQL procedure successfully completed.
```

%ROWCOUNT is an incremental count of the number of rows, and hence you can use it to check for a particular value. In this example, the first three lines after the BEGIN and before the cursor loop are displayed, irrespective of the number of rows returned by the cursor. This is true even if the cursor returned no rows. To prevent this, you can use the value of %ROWCOUNT to display them only if the cursor returns at least one row. Here's the code to do so:

```
DECLARE
     CURSOR csr_org IS
          SELECT h.hrc_descr, o.org_short_name
          FROM   org_tab o, hrc_tab h
          WHERE o.hrc_code = h.hrc_code
           ORDER by 2;
        num_total_rows NUMBER;
BEGIN
        FOR idx IN csr_org LOOP
              IF csr_org%ROWCOUNT = 1 THEN
                 dbms_output.put_line('Organization Details with Hierarchy');
                 dbms_output.put_line
                 ('------------------------');
                 dbms_output.put_line(rpad('Hierarchy',20,' ')||' '||
                                 rpad('Organization',30,' '));
                 dbms_output.put_line(rpad('-',20,'-')||' '||rpad('-',30,'-'));
              END IF;
              dbms_output.put_line(rpad(idx.hrc_descr,20,' ')||' '||
                              rpad(idx.org_short_name,30,' '));
               num_total_rows := csr_org%ROWCOUNT;
       END LOOP;
       IF num_total_rows > 0 THEN
         dbms_output.new_line;
         dbms_output.put_line('Total Organizations = '||to_char(num_total_rows));
       END IF;
END;
/
```

The following points are worth noting:

- The %ROWCOUNT is checked inside the cursor FOR LOOP.
- After the first row is fetched, the value of %ROWCOUNT is 1 and the headings are displayed. Successive fetches increment the value of %ROWCOUNT by 1 so that %ROWCOUNT is greater than 1 after the first fetch.
- After the last fetch, the cursor FOR LOOP is exited and the value of %ROWCOUNT is the total number of rows processed.

Parameterized Cursors

An explicit cursor can take parameters and return a data set for a specific parameter value. This eliminates the need to define multiple cursors and hard-code a value in each cursor. It also eliminates the need to use PL/SQL bind variables.

In the following code, I use the cursor example presented earlier in the section to illustrate parameterized cursors:

```
DECLARE
     CURSOR csr_org(p_hrc_code NUMBER) IS
          SELECT h.hrc_descr, o.org_short_name
          FROM   org_tab o, hrc_tab h
          WHERE o.hrc_code = h.hrc_code
                AND h.hrc_code = p_hrc_code
           ORDER by 2;
           v_org_rec csr_org%ROWTYPE;
   BEGIN
        OPEN csr_org(1);
        dbms_output.put_line('Organization Details with Hierarchy 1');
        dbms_output.put_line('------------------------');
        dbms_output.put_line(rpad('Hierarchy',20,' ')||' ||
                        rpad('Organization',30,' '));
        dbms_output.put_line(rpad('-',20,'-')||' ||rpad('-',30,'-'));
        LOOP
             FETCH csr_org INTO v_org_rec;
             EXIT WHEN csr_org%NOTFOUND;
             dbms_output.put_line(rpad(v_org_rec.hrc_descr,20,' ')||' ||
                                  rpad(v_org_rec.org_short_name,30,' '));
       END LOOP;
       CLOSE csr_org;
       OPEN csr_org(2);
        dbms_output.put_line('Organization Details with Hierarchy 2');
        dbms_output.put_line('------------------------');
        dbms_output.put_line(rpad('Hierarchy',20,' ')||' ||
                        rpad('Organization',30,' '));
        dbms_output.put_line(rpad('-',20,'-')||' ||rpad('-',30,'-'));
        LOOP
             FETCH csr_org INTO v_org_rec;
             EXIT WHEN csr_org%NOTFOUND;
             dbms_output.put_line(rpad(v_org_rec.hrc_descr,20,' ')||' ||
```

```
                        rpad(v_org_rec.org_short_name,30,' '));
       END LOOP;
       CLOSE csr_org;
 END;
/
```

Here's the output of this program:

```
Organization Details with Hierarchy 1
-------------------------------------------
Hierarchy            Organization
------------------   -------------------------
CEO/COO              Office of CEO ABC Inc.
CEO/COO              Office of CEO DataPro Inc.
CEO/COO              Office of CEO XYZ Inc.
Organization Details with Hierarchy 2
-------------------------------------------
Hierarchy            Organization
------------------   -------------------------
VP                   Office of VP Mktg ABC Inc.
VP                   Office of VP Sales ABC Inc.
VP                   Office of VP Tech ABC Inc.

PL/SQL procedure successfully completed.
```

You define the cursor parameters immediately after the cursor name by including the name of the parameter and its data type within parentheses. These are referred to as the *formal parameters*. The actual parameters (i.e., the actual data values for the formal parameters) are passed via the OPEN statement as shown in the previous example. Notice how the same cursor is used twice with different values of the parameters in each case.

You can rewrite the same example using a cursor FOR LOOP. In this case, the actual parameters are passed via the cursor name referenced in the cursor FOR LOOP. Here's the code:

```
DECLARE
     CURSOR csr_org(p_hrc_code NUMBER) IS
          SELECT h.hrc_descr, o.org_short_name
          FROM   org_tab o, hrc_tab h
          WHERE o.hrc_code = h.hrc_code
                AND h.hrc_code = p_hrc_code
           ORDER by 2;
           v_org_rec csr_org%ROWTYPE;
  BEGIN
```

```
        dbms_output.put_line('Organization Details with Hierarchy 1');
        dbms_output.put_line('-----------------------');
        dbms_output.put_line(rpad('Hierarchy',20,' ')||
                             ' '||rpad('Organization',30,' '));
        dbms_output.put_line(rpad('-',20,'-')||' '||rpad('-',30,'-'));
        FOR idx in csr_org(1) LOOP
             dbms_output.put_line(rpad(idx.hrc_descr,20,' ')||' '||
                            rpad(idx.org_short_name,30,' '));
       END LOOP;
        dbms_output.put_line('Organization Details with Hierarchy 2');
        dbms_output.put_line('-----------------------');;
        dbms_output.put_line(rpad('Hierarchy',20,' ')||' '||
                                            rpad('Organization',30,' '));
        dbms_output.put_line(rpad('-',20,'-')||' '||rpad('-',30,'-'));
        FOR idx in csr_org(2) LOOP
             dbms_output.put_line(rpad(idx.hrc_descr,20,' ')||' '||
                             rpad(idx.org_short_name,30,' '));
       END LOOP;
 END;
/
```

The output of this program is the same as the output of the earlier one.

Parameterized cursors are very useful in processing nested cursor loops in which an inner cursor is opened with data values passed to it from an outer opened cursor.

SELECT FOR UPDATE Cursors

You use SELECT FOR UPDATE cursors for updating the rows retrieved by a cursor. This is often required when there's a need to modify each row retrieved by a cursor without having to refetch that row. More often, SELECT FOR UPDATE cursors are required to update a column of the table defined in the cursor SELECT using a complex formula.

Defining a SELECT FOR UPDATE Cursor

A SELECT FOR UPDATE cursor is defined using the FOR UPDATE OF clause in the cursor SELECT statement, as follows:

```
DECLARE
     CURSOR csr_1 IS
           SELECT * FROM sec_hrc_tab FOR UPDATE OF hrc_descr;
BEGIN
     /* ... Open the cursor and process the resultset ... */
    null;
END;
/
```

NOTE *Notice how the column name to be updated is specified in the FOR UPDATE OF clause. If no column name is specified in the FOR UPDATE OF clause, any column of the underlying cursor table can be modified.*

Using a SELECT FOR UPDATE Cursor

Once you've defined a SELECT FOR UPDATE cursor, you use the WHERE CURRENT OF clause to process the rows returned by it. You can use this clause in an UPDATE or DELETE statement. It has the following syntax:

```
WHERE CURRENT of cursor_name;
```

where cursor_name is the name of the cursor defined with a FOR UPDATE clause.

The following is a complete example of using SELECT FOR UPDATE cursors. I use the sec_hrc_tab table to demonstrate this. First, this table is populated using an INSERT statement as follows:

```
BEGIN
    INSERT INTO sec_hrc_tab
        SELECT * FROM hrc_tab;
    COMMIT;
END;
/
```

The output can be verified as follows:

```
SQL> select * from sec_hrc_tab;

  HRC_CODE HRC_DESCR
```

```
----- -------------
          1 CEO/COO
          2 VP
          3 Director
          4 Manager
          5 Analyst
```

Then I define a SELECT FOR UPDATE cursor and use the WHERE CURRENT OF clause to update the rows retrieved by this cursor in a particular fashion. Here's the program for this:

```
DECLARE
       CURSOR csr_1 IS
            SELECT * FROM sec_hrc_tab FOR UPDATE OF hrc_descr;
            v_hrc_descr VARCHAR2(20);
BEGIN
     FOR idx IN csr_1 LOOP
             v_hrc_descr := UPPER(idx.hrc_descr);
             UPDATE sec_hrc_tab
              SET         hrc_descr = v_hrc_descr
              WHERE CURRENT OF csr_1;
     END LOOP;
     COMMIT;
END;
/
```

This program updates the hrc_descr column of each row retrieved by csr_1 with its value converted to uppercase. The output can be verified as follows:

```
 SQL> select * from sec_hrc_tab;

   HRC_CODE HRC_DESCR
   -------- ----------
          1 CEO/COO
          2 VP
          3 DIRECTOR
          4 MANAGER
          5 ANALYST
```

The mechanism of SELECT FOR UPDATE cursors works as follows:

1. The SELECT FOR UPDATE cursor puts a lock on the rows retrieved by the cursor. If it's unable to obtain a lock because some other session has placed a lock on the specific rows, it waits until it can get a lock. A COMMIT or ROLLBACK in the corresponding session frees the locks held by other sessions.

2. For each row identified by the cursor, the cursor updates the specified column of that row. That is, it keeps track of the current row and updates it, and then fetches the subsequent row and updates it. It does this without scanning the same table again. This is unlike an ordinary UPDATE or DELETE statement inside the loop, where the cursor scans the updated table again to determine the current row to be modified.

Although you could achieve the same function by using a simple UPDATE statement, this example is meant to illustrate the use of SELECT FOR UPDATE cursors.

To use WHERE CURRENT OF, you have to declare the cursor using FOR UPDATE. The reverse is not true. That is, you can use a SELECT FOR UPDATE cursor to modify the rows without using the WHERE CURRENT OF clause. Then, you have to update or delete the cursor rows using the primary key.

A SELECT FOR UPDATE cursor offers two important advantages: Namely, it locks the rows after opening the cursor and the resultset rows are identified for update, and it eliminates a second fetch of the rows for doing the update and preserves the current row by the WHERE CURRENT OF clause.

You have to do a COMMIT outside of the cursor loop when you use WHERE CURRENT OF in processing the rows of a SELECT FOR UPDATE cursor. This is because a COMMIT releases the lock on the rows that the SELECT FOR UPDATE has put a lock on, and this causes a subsequent fetch to fail.

Implicit Cursors

Of all the types of DML statements, the explicit cursors discussed previously are used for processing multirow SELECT statements. To keep track of other types of DML statements, such as INSERT, UPDATE, DELETE, and single-row SELECT . . . INTO statements, Oracle PL/SQL provides the implicit cursor, also known as the SQL cursor. Just as a SELECT statement points to a work area whether it returns a single row or multiple rows, even INSERT, UPDATE, and DELETE statements are executed within the context of a work area, and the Oracle PL/SQL engine automatically opens the implicit or SQL cursor that points to this work area. Also, after the execution of the DML statements, the implicit cursor is automatically

closed. Hence, there's no such thing as OPEN, FETCH, and CLOSE. These operations are only valid for an explicit cursor. Here's an example of an implicit cursor:

```
BEGIN
         DELETE sec_hrc_org_tab WHERE hrc_code = 1;
         INSERT INTO sec_hrc_org_tab
                SELECT h.hrc_code, h.hrc_descr,
                               o.org_id, o.org_short_name, o.org_long_name
                FROM   org_tab o, hrc_tab h
                WHERE o.hrc_code = h.hrc_code
                      AND h.hrc_code = 1;
        IF (SQL%FOUND) THEN
           dbms_output.put_line(TO_CHAR(SQL%ROWCOUNT)||
                 ' rows inserted into secondary table for hierarchy 1');
        END IF;
        COMMIT;
 END;
/
```

The output of this code can be verified as follows:

```
3 rows inserted into secondary table for hierarchy 1

PL/SQL procedure successfully completed.

SQL> select * from sec_hrc_org_tab;

  HRC_CODE HRC_DESCR                ORG_ID ORG_SHORT_NAME
----- ----------     -----  -------------
ORG_LONG_NAME
-----------------------------------------------
         1 CEO/COO                    1001 Office of CEO ABC Inc.
Office of CEO ABC Inc.

         1 CEO/COO                    1002 Office of CEO XYZ Inc.
Office of CEO XYZ Inc.

         1 CEO/COO                    1003 Office of CEO DataPro Inc.
Office of CEO DataPro Inc.
```

This code refreshes a secondary table named sec_hrc_org_tab with new rows. It first deletes all rows from the sec_hrc_org_tab table where the hrc_code matches 1. It then inserts new rows into the same table. Now the question is, did

the INSERT succeed? That is, did it insert zero or more rows? This is determined by an implicit cursor attribute, SQL%FOUND, which is defined in the statement

```
IF (SQL%FOUND) THEN
```

SQL%FOUND returns a boolean true when at least one row has been inserted into the temp_hrc_org_tab. When this happens, the code inside the IF condition is executed and the given output appears. Also note the use of the SQL%ROWCOUNT attribute. This gives the numbers of rows inserted into the sec_hrc_org_tab table. Note that the SQL%ROWCOUNT gives the number of rows affected by the immediately preceding DML statement.

Implicit Cursor Attributes

Although an implicit cursor is opened and closed automatically by the PL/SQL engine, the four attributes associated with an explicit cursor are also available for an implicit cursor. You can reference these attributes by prefixing the keyword SQL with the particular attribute. Table 2-2 lists the four attributes of the implicit cursor.

Table 2-2. Implicit Cursor Attributes

ATTRIBUTE	USE
SQL%FOUND	Indicates whether an INSERT, UPDATE, or DELETE affected any row(s) or not.
SQL%ISOPEN	Indicates whether the cursor is OPEN or not. This is FALSE always, as the implicit cursor is closed after the DML statement is executed.
SQL%NOTFOUND	Indicates if a DML statement failed to modify any rows.
SQL%ROWCOUNT	Indicates the number of rows affected by the DML statement.

Note that the name of the cursor in this case is “SQL” instead of a programmer-defined cursor name.

The SQL%FOUND, SQL%ISOPEN, and SQL%NOTFOUND attributes return a boolean TRUE or FALSE, and the SQL%ROWCOUNT attribute returns a numeric value. The following sections describe these attributes in detail.

SQL%FOUND

You use SQL%FOUND to determine whether an INSERT, UPDATE, or DELETE affected any row(s) or not, or a SELECT . . . INTO returned a row or not. You should use it immediately after the DML statement, and it returns a value of TRUE if the INSERT, UPDATE, or DELETE affected one or more rows, or the SELECT . . . INTO fetched a row. Otherwise, it returns a value of FALSE. Using SQL%FOUND before defining any DML statement yields NULL.

I provided an example of using SQL%FOUND during the discussion of implicit cursors. I repeat it here for illustration:

```
BEGIN
         DELETE sec_hrc_org_tab WHERE hrc_code = 1;
         INSERT INTO sec_hrc_org_tab
                SELECT h.hrc_code, h.hrc_descr,
                               o.org_id, o.org_short_name, o.org_long_name
                FROM   org_tab o, hrc_tab h
                WHERE o.hrc_code = h.hrc_code
                      AND h.hrc_code = 1;
        IF (SQL%FOUND) THEN
           dbms_output.put_line(TO_CHAR(SQL%ROWCOUNT)||
                  ' rows inserted into secondary table for hierarchy 1');
        END IF;
        COMMIT;
 END;
/
```

The following points are worth noting:

- The statement IF (SQL%FOUND) THEN appears immediately after the INSERT statement and it always should. If SQL%FOUND is referenced before the INSERT statement, it returns NULL.

- The condition SQL%FOUND evaluates to TRUE if the INSERT succeeded in creating one or more rows; otherwise, it evaluates to FALSE and the code inside the IF is never executed.

SQL%ISOPEN

SQL%ISOPEN is always FALSE because the implicit cursor is closed after the DML statement is executed. Hence, it's not useful to check this attribute for the same.

SQL%NOTFOUND

You use SQL%NOTFOUND to determine if an INSERT, UPDATE, or DELETE failed to modify any rows. It returns a value of TRUE if no rows were modified by the INSERT, UPDATE, or DELETE, and a value of FALSE if at least one row was modified. Using SQL%NOTFOUND before executing any DML statement yields a NULL value. Here's an example of using SQL%NOTFOUND:

```
DECLARE
   v_num_rows NUMBER;
BEGIN
         DELETE sec_hrc_org_tab WHERE hrc_code = 1;
         INSERT INTO sec_hrc_org_tab
                SELECT h.hrc_code, h.hrc_descr,
                               o.org_id, o.org_short_name, o.org_long_name
                FROM   org_tab o, hrc_tab h
                WHERE o.hrc_code = h.hrc_code
                      AND h.hrc_code = 1;
        v_num_rows := SQL%ROWCOUNT;
        IF (SQL%FOUND) THEN
           UPDATE sec_hrc_audit
           SET num_rows = v_num_rows
           WHERE hrc_code = 1;
           IF (SQL%NOTFOUND) THEN
            INSERT INTO sec_hrc_audit(hrc_code, num_rows) VALUES (1, v_num_rows);
           END IF;
         END IF;
         COMMIT;
 END;
/
```

The output of this program can be verified as follows:

```
PL/SQL procedure successfully completed.

SQL> select * from sec_hrc_org_tab;

  HRC_CODE HRC_DESCR                ORG_ID ORG_SHORT_NAME
------ ---------- ------ ---------------
ORG_LONG_NAME
---------------------------------------------
         1 CEO/COO                    1001 Office of CEO ABC Inc.
Office of CEO ABC Inc.
```

```
         1 CEO/COO                      1002 Office of CEO XYZ Inc.
Office of CEO XYZ Inc.

         1 CEO/COO                      1003 Office of CEO DataPro Inc.
Office of CEO DataPro Inc.

SQL> select * from sec_hrc_audit;

  HRC_CODE   NUM_ROWS
----- -----
 1     3
```

This code first deletes all rows from the sec_hrc_org_tab table where the hrc_code matches 1. It then inserts new rows into the same table. Now the question is, did the INSERT succeed? That is, did it insert zero or more rows? This is determined by the implicit cursor attribute SQL%FOUND, which is defined in the statement

```
IF (SQL%FOUND) THEN
```

SQL%FOUND returns a boolean true when at least one row has been inserted into the sec_hrc_org_tab. When this happens, the code inside the IF condition is executed and the UPDATE statement against the sec_hrc_audit table is executed.

Now the second question is, did this update succeed or fail? This is determined by the implicit cursor attribute SQL%NOTFOUND. If the update failed, SQL%NOTFOUND returns TRUE and a record is inserted into the sec_hrc_audit table. Notice the use of SQL% attributes immediately after each DML statement. The use of SQL%FOUND refers to its immediately preceding DML statement—that is, the first INSERT statement. The use of the SQL%NOTFOUND attribute refers to its immediately preceding DML statement—that is, the UPDATE statement. Also note the use of the SQL%ROWCOUNT attribute. This attribute gives the numbers of rows inserted into the sec_hrc_org_tab table, as it's used immediately after the INSERT statement.

SQL%ROWCOUNT

You use %ROWCOUNT to determine the number of rows affected by a DML statement. It returns a value greater than zero if the DML statement succeeded; otherwise, it returns zero. It's a good alternative to SQL%NOTFOUND. Since %NOTFOUND returns TRUE if the DML statement failed, it's equivalent to use

```
IF (SQL%ROWCOUNT = 0) THEN . . .
```

instead of

```
IF (SQL%NOTFOUND) THEN . . .
```

Here's the previous example modified to use %ROWCOUNT:

```
DECLARE
   v_num_rows NUMBER;
BEGIN
         DELETE sec_hrc_org_tab WHERE hrc_code = 1;
         INSERT INTO sec_hrc_org_tab
                SELECT h.hrc_code, h.hrc_descr,
                               o.org_id, o.org_short_name, o.org_long_name
                FROM   org_tab o, hrc_tab h
                WHERE o.hrc_code = h.hrc_code
                      AND h.hrc_code = 1;
        v_num_rows := SQL%ROWCOUNT;
        IF (SQL%FOUND) THEN
           UPDATE sec_hrc_audit
           SET num_rows = v_num_rows
           WHERE hrc_code = 1;
           IF (SQL%ROWCOUNT=0) THEN
            INSERT INTO sec_hrc_audit(hrc_code, num_rows) VALUES (1, v_num_rows);
           END IF;
         END IF;
         COMMIT;
 END;
/
```

The output of this program is same as the output of the previous example.

The following points are worth noting:

- The first SQL%ROWCOUNT returns the number of rows affected by the very first INSERT statement—that is, the number of rows inserted into the sec_hrc_org_tab table.

- The second SQL%ROWCOUNT returns the number of rows affected by the UPDATE statement against the table sec_hrc_audit.

TIP *Always check for the attributes SQL%FOUND, SQL%NOTFOUND, and SQL%ROWCOUNT* immediately after *the DML statement.*

How Using SQL%FOUND, SQL%NOTFOUND, or SQL%ROWCOUNT Replaces a SELECT COUNT(*)

Using SQL%FOUND, SQL%NOTFOUND, or SQL%ROWCOUNT replaces a SELECT COUNT(*), as you can see from the previous example. Notice the IF statement after the first statement. If this weren't there, the way to check whether the insert succeeded or not is to do a SELECT COUNT(*) from the sec_hrc_org_table into a variable and explicitly check for its value to be greater than zero. The same is true for the sec_hrc_audit table. Hence, the program will be as shown here:

```
DECLARE
   v_num_rows NUMBER;
   v_cnt NUMBER;
BEGIN
        DELETE sec_hrc_org_tab WHERE hrc_code = 1;
        INSERT INTO sec_hrc_org_tab
              SELECT h.hrc_code, h.hrc_descr,
                              o.org_id, o.org_short_name, o.org_long_name
              FROM   org_tab o, hrc_tab h
              WHERE o.hrc_code = h.hrc_code
                    AND h.hrc_code = 1;
       SELECT COUNT(*)
      INTO     v_num_rows
      FROM   sec_hrc_org_tab
      WHERE hrc_code = 1;
     IF (v_num_rows >0) THEN
        SELECT COUNT(*)
        INTO    v_cnt
        FROM  sec_hrc_audit
        WHERE hrc_code = 1;
        IF (v_cnt > 0) THEN
          UPDATE sec_hrc_audit
          SET num_rows = v_num_rows
          WHERE hrc_code = 1;
```

```
        ELSIF (v_cnt=0) THEN
            INSERT INTO sec_hrc_audit(hrc_code, num_rows) VALUES (1, v_num_rows);
          END IF;
         END IF;
         COMMIT;
 END;
/
```

The output of this program is same as the output of the previous example.

Even if you don't use a SELECT COUNT(*), at least use a SELECT . . . INTO instead. Using implicit cursor attributes saves this overhead.

Cursor Variables

As mentioned in the earlier section, "Introducing Cursors," an explicit cursor once declared was associated with a specific query—only the one specific query that was known at compile time. In this way, the cursor declared was static and couldn't be changed at runtime. It always pointed to the same work area until the execution of the program completed. However, you may sometimes want to have a variable that can point to different work areas depending on runtime conditions. PL/SQL 2.2 onward offers this facility by means of cursor variables.

A *cursor variable* is a single PL/SQL variable that you can associate with different queries at runtime. The same variable can point to different work areas. In this way, cursor variables and cursors are analogous to PL/SQL variables and constants, but from a cursor perspective. A cursor variable acts like a pointer that holds the address of a specific work area defined by the query it's pointing to.

Before PL/SQL 2.3, cursor variables were available for use in host environments such as Pro*C. As of PL/SQL 2.3 onward, cursor variables are available for use in both server- and client-side PL/SQL as well as in host environments.

Why Use Cursor Variables?

The primary advantage of using cursor variables is their capability to pass resultsets between stored subprograms. Before cursor variables, this wasn't possible. Now, with cursor variables, the work area that a cursor variable points to remains accessible as long as the variable points to it. Hence, you can point a cursor variable to a work area by opening a cursor for it, and then any application such as Pro*C, an Oracle client, or another server application can fetch from the corresponding resultset.

Another advantage of cursor variables is their introduction of a sort of dynamism, in that a single cursor variable can be associated with multiple queries at runtime.

Defining a Cursor Variable

Defining a cursor variable consists of defining a pointer of type REF CURSOR and defining a variable of this type. These steps are outlined in the following sections.

Defining a Pointer of Type CURSOR

In PL/SQL, a pointer is declared using the syntax

```
REF type
```

The keyword REF implies that the new type so defined is a pointer to the defined type.

PL/SQL offers two types of REF types: CURSOR and an object type. So, the definition of a cursor variable involves the definition of a REF CURSOR first, as shown here:

```
TYPE rc IS REF CURSOR;
```

Defining a Variable of Type REF CURSOR

Once you've defined a REF CURSOR type, the next step is to declare a variable of this type. Here's the code for this:

```
v_rc rc;
```

So the complete declaration of a cursor variable is as follows:

```
TYPE rc IS REF CURSOR;
v_rc rc;
```

This code suggests that rc is a pointer of type CURSOR and v_rc (in fact, any variable) defined of type rc points to a SQL cursor.

Strong and Weak REF CURSOR Types

The REF CURSOR type defined earlier is called a *weak* REF CURSOR type. This is because it doesn't dictate the return type of the cursor. Hence, it can point to any SELECT query with any number of columns. Weak cursor types are available in PL/SQL 2.3 and higher versions.

PL/SQL lets you define a *strong* REF CURSOR having a return type using the following syntax:

```
TYPE ref_type_name IS REF CURSOR RETURN return_type;
```

Here, ref_type_name is the name of the new pointer name and return_type is a record type of either %ROWTYPE or a user-defined record type. For example, you can declare strong REF CURSORS as follows:

```
TYPE rc is REF CURSOR RETURN hrc_tab%ROWTYPE;
v_rc rc;
```

or

```
TYPE hrc_rec is RECORD (hrc_code NUMBER, hrc_name VARCHAR2(20));
TYPE rc IS REF CURSOR RETURN hrc_rec;
```

In the case of a strong REF CURSOR, the query that's associated with it should be type-compatible one to one with the return type of the corresponding REF CURSOR.

Using a Cursor Variable

Once you've defined a cursor variable, you can use it to associate it with a query. Here are the steps:

1. Allocate memory.
2. Open the cursor variable for a query.
3. Fetch the results into a PL/SQL record or individual PL/SQL variables.
4. Close the cursor variable.

The following sections provide more detail about each step in the process.

Allocate Memory

Once you declare a cursor variable in PL/SQL, the PL/SQL engine in PL/SQL 2.3 and higher versions automatically allocates memory for storage of rows. Prior to PL/SQL 2.3, a host environment was needed to explicitly allocate memory to a cursor variable.

Opening the Cursor Variable

Once you've defined a cursor variable, you have to open it for a multirow query, either with an arbitrary number of columns in the case of a weak REF CURSOR or with a type-compatible query in the case of a strong REF CURSOR. Opening the cursor variable identifies the associated query, executes it, and also identifies the resultset.

You open a cursor variable using the OPEN-FOR statement. Here's the syntax:

```
OPEN {cursor_variable_name | :host_cursor_variable_name} FOR
{  select_query
 | dynamic_string [USING bind_variable[, bind_variable] . . . ] };
```

where cursor_variable_name is the name of the declared cursor variable and select_query is the SELECT query associated with the cursor variable. Also, host_cursor_variable_name is the name of the cursor variable declared in a PL/SQL host environment (such as Pro*C), and bind_variable represents the name of a PL/SQL bind variable. dynamic_string represents a dynamic SQL string instead of a hard-coded SELECT statement. You open cursor variables for dynamic strings using native dynamic SQL.

CROSS-REFERENCE *Chapter 7 covers opening cursor variables for dynamic strings using native dynamic SQL.*

Here's an example that illustrates opening the cursor variable for the previously declared weak cursor variable v_rc:

```
DECLARE
     TYPE rc is REF CURSOR;
     v_rc rc;
BEGIN
```

```
    OPEN v_rc FOR SELECT * from hrc_tab;
    /* . . . FETCH the results and process the resultset */
   null;
END;
/
```

TIP *You can't define any parameters while opening a cursor variable for a query. However, the associated query can reference PL/SQL variables, parameters, host variables, and functions.*

Fetching the Results into a PL/SQL Record or Individual PL/SQL Variables

The next step is to fetch the cursor variable into a PL/SQL record or individual variables. This retrieves individual rows of data into the PL/SQL variables for processing. You fetch a cursor variable using the FETCH statement, which has three forms. Here's the syntax:

```
FETCH cursor_variable_name INTO var1, var2, . . . , varN;
```

or

```
FETCH cursor_variable_name INTO table_name%ROWTYPE;
```

or

```
FETCH cursor__variable_name INTO record_name;
```

Here, var1, var2, and varN represent PL/SQL variables having data types identical to the cursor variable query. table_name%ROWTYPE represents a PL/SQL record type with attributes implicitly defined as the column names of the table identified by table_name, which are identical to the cursor variable SELECT. In this case, you need to explicitly define the record type. Lastly, record_name is a variable of a PL/SQL record type that's explicitly defined. In this case also, the number and data types of the individual attributes of the record should exactly match the columns in the cursor variable SELECT.

Here's an example that extends the previous example of v_rc to fetching rows:

```
DECLARE
     TYPE rc is REF CURSOR;
     v_rc rc;
```

```
     hrc_rec hrc_tab%ROWTYPE;
BEGIN
    OPEN v_rc FOR SELECT * from hrc_tab;
    LOOP
        FETCH v_rc INTO hrc_rec;
        EXIT WHEN v_rc%NOTFOUND;
        /* ... Process the individual records */
        null;
     END LOOP;
END;
/
```

The number and data types of the individual variables should exactly match the columns list in the cursor variable's associated SELECT statement. If the cursor is fetched into a record type (either table_name%ROWTYPE or record_name), the number and data type of each attribute in the record should exactly match the columns list of the cursor variable associated SELECT statement. If this isn't the case, then PL/SQL raises an error at compile time if the cursor variable is strongly typed, and a predefined exception called ROWTYPE_MISMATCH at runtime if the cursor variable is weakly typed.

CAUTION *Never fetch from a cursor variable before opening it.*

TIP *Always fetch into a record type of table_name%ROWTYPE, or at least fetch into a record type compatible with the cursor SELECT rather than into individual variables. This is less error-prone and also improves program readability.*

Similar to static cursors, a single FETCH always fetches only one row (the current row) from the active set. To fetch multiple rows, use the FETCH statement in a LOOP.

Closing the Cursor Variable

Once the processing of the rows is completed, you can close the cursor variable. Closing the cursor variable frees the resources allocated to the query but doesn't necessarily free the storage of the cursor variable itself. The cursor variable is freed when the variable is out of scope. You close a cursor using the CLOSE statement. Here's the syntax:

```
CLOSE cursor_variable_name;
```

TIP *The CLOSE statement should always appear after the FETCH statement. When you use a loop to fetch the rows from a cursor variable, you should insert the CLOSE statement after you close the loop. Otherwise, it results in an illegal fetch. Don't fetch from an already closed cursor variable, and don't close an already closed cursor variable.*

Here's a complete example of using the v_rc cursor, involving all the steps previously covered:

```
DECLARE
      TYPE rc is REF CURSOR;
      v_rc rc;
      hrc_rec hrc_tab%ROWTYPE;
BEGIN
     OPEN v_rc FOR SELECT * from hrc_tab;
     dbms_output.put_line('Hierarchy Details');
     dbms_output.put_line('------------------------');
     dbms_output.put_line('Code'||' '||rpad('Description',20,' '));
     dbms_output.put_line(rpad('-',4,'-')||' '||rpad('-',20,'-'));
     LOOP
         FETCH v_rc INTO hrc_rec;
         EXIT WHEN v_rc%NOTFOUND;
         dbms_output.put_line(to_char(hrc_rec.hrc_code)||' '||
                                              rpad(hrc_rec.hrc_descr,20,' '));
      END LOOP;
      CLOSE v_rc;
END;
/
```

Here's the output of this program:

```
Hierarchy Details
-----------------
Code Description
-- ------------
1 CEO/COO
2 VP
3 Director
4 Manager
5 Analyst

PL/SQL procedure successfully completed.
```

This code is similar to the code used for static cursors, except that it uses cursor variables instead of cursors.

TIP *The scope of a cursor variable is the scope of the PL/SQL block in which it is defined.*

Cursor Variables Assignment

One way to make a cursor variable point to a query work area is to open a query for the cursor variable. You saw this earlier. Here, I describe a second way to make a cursor variable point to a query work area. Simply assign the cursor variable to an already OPENed cursor variable. Here's an example of cursor variable assignment:

```
DECLARE
      TYPE rc is REF CURSOR;
      v_rc1 rc;
      v_rc2 rc;
      hrc_rec hrc_tab%ROWTYPE;
BEGIN
    OPEN v_rc1 FOR SELECT * from hrc_tab;
    dbms_output.put_line('Hierarchy Details');
    dbms_output.put_line('------------------------');
    dbms_output.put_line('Code'||' '||rpad('Description',20,' '));
    dbms_output.put_line(rpad('-',4,'-')||' '||rpad('-',20,'-'));
    /* Assign v_rc1 to v_rc2 */
    v_rc2 := v_rc1;
```

```
    LOOP
        /* Fetch from the second cursor variable, i.e., v_rc2 */
        FETCH v_rc2 INTO hrc_rec;
        EXIT WHEN v_rc2%NOTFOUND;
        dbms_output.put_line(to_char(hrc_rec.hrc_code)||' '||
                                              rpad(hrc_rec.hrc_descr,20,' '));
     END LOOP;
     CLOSE v_rc2;
END;
/
```

The output of this program is the same as the output of the earlier example without the assignment. Note that closing v_rc2 also closes v_rc1 and vice versa.

However, if the source cursor variable is strongly typed, the target cursor variable must be of the same type as the source cursor variable. This restriction doesn't apply if the source cursor variable is weakly typed. Here's an example that illustrates this concept:

```
DECLARE
     TYPE rc1 is REF CURSOR RETURN hrc_tab%ROWTYPE;
     TYPE rc2 is REF CURSOR RETURN hrc_tab%ROWTYPE;
     TYPE rc is REF CURSOR;
     v_rc1 rc1;
     v_rc2 rc2;
     v_rc3 rc;
     v_rc4 rc;
     hrc_rec hrc_tab%ROWTYPE;
BEGIN
    OPEN v_rc1 FOR SELECT * from hrc_tab;
   /* Assign v_rc1 to v_rc2 */
    v_rc2 := v_rc1; – This causes type error.
    v_rc3 := v_rc1; – This succeeds.
    v_rc4 := v_rc3; – This succeeds.
    /* . . . FETCH and process . . . */
    null;
END;
/
```

CAUTION *Don't assign an unopened cursor variable to another cursor variable. Doing so causes the error INVALID_CURSOR.*

TIP *You can't assign a null value to a cursor variable. Also, you can't test cursor variables for equality, inequality, or nullity.*

Cursor Variable Attributes

All the attributes associated with explicit cursors are available with cursor variables. You can use the four explicit cursor attributes with cursor variables by referencing them as cursor_variable_name%ISOPEN, cursor_variable_name%FOUND, cursor_variable_name%NOTFOUND, and cursor_variable_name%ROWCOUNT.

SYS_REFCURSOR Type in PL/SQL 9i

PL/SQL 9*i* makes available a type called SYS_REFCURSOR that defines a generic weak cursor. You can use it as follows:

```
DECLARE
     v_rc SYS_REFCURSOR;
BEGIN
     OPEN v_rc FOR SELECT * from hrc_tab;
    /* . . . FETCH and process the resultset  . . . */
   null;
END;
/
```

Before Oracle9*i* you needed to perform two steps:

1. Define a type of REF CURSOR.

2. Define the cursor variable of this type.

SYS_REFCURSOR makes it convenient to define a cursor variable in a single step. However, you can use it to define only weak cursor variables. Here's an example of using SYS_REFCURSOR for cursor variable processing:

```
DECLARE
     v_rc SYS_REFCURSOR;
     hrc_rec hrc_tab%ROWTYPE;
```

```
BEGIN
    OPEN v_rc FOR SELECT * from hrc_tab;
    dbms_output.put_line('Hierarchy Details');
     dbms_output.put_line('-----------------------');
    dbms_output.put_line('Code'||' '||rpad('Description',20,' '));
    dbms_output.put_line(rpad('-',4,'-')||' '||rpad('-',20,'-'));
    LOOP
        FETCH v_rc INTO hrc_rec;
        EXIT WHEN v_rc%NOTFOUND;
        dbms_output.put_line(to_char(hrc_rec.hrc_code)||' '||
                                            rpad(hrc_rec.hrc_descr,20,' '));
     END LOOP;
     CLOSE v_rc;
END;
/
```

Dynamism in Using Cursor Variables

The real use of cursor variables is when you have a need to open multiple queries using the same cursor variable or to dynamically assign different queries to the same cursor variable depending on runtime conditions. I discuss two examples in the following sections that illustrate the dynamism involved in using cursor variables.

Example 1: Opening Multiple Queries Using the Same Cursor Variable

To open multiple queries using the same cursor variable, use this code:

```
DECLARE
     TYPE rc is REF CURSOR;
     v_rc rc;
     hrc_rec hrc_tab%ROWTYPE;
     v_hrc_descr VARCHAR2(20);
     v_org_short_name VARCHAR2(30);
BEGIN
    OPEN v_rc FOR SELECT * from hrc_tab;
    dbms_output.put_line('Hierarchy Details');
    dbms_output.put_line('-----------------------');
    dbms_output.put_line('Code'||' '||rpad('Description',20,' '));
    dbms_output.put_line(rpad('-',4,'-')||' '||rpad('-',20,'-'));
    LOOP
```

```
        FETCH v_rc INTO hrc_rec;
        EXIT WHEN v_rc%NOTFOUND;
        dbms_output.put_line(to_char(hrc_rec.hrc_code)||' '||
                                              rpad(hrc_rec.hrc_descr,20,' '));
     END LOOP;
     OPEN v_rc FOR SELECT h.hrc_descr, o.org_short_name
                                  FROM org_tab o, hrc_tab h
                                  WHERE o.hrc_code = h.hrc_code;
    dbms_output.put_line('Hierarchy and Organization Details');
    dbms_output.put_line('------------------------');
    dbms_output.put_line(rpad('Hierarchy',20,' ')||' '||
                                         rpad('Description',30,' '));
    dbms_output.put_line(rpad('-',20,'-')||' '||rpad('-',30,'-'));
    LOOP
        FETCH v_rc INTO v_hrc_descr, v_org_short_name;
        EXIT WHEN v_rc%NOTFOUND;
        dbms_output.put_line(rpad(v_hrc_descr,20,' ')||' '||
                                              rpad(v_org_short_name,30,' '));
     END LOOP;
     CLOSE v_rc;
END;
/
```

Here's the output of this program:

```
Hierarchy Details
---------------
Code Description
-- --------------
1 CEO/COO
2 VP
3 Director
4 Manager
5 Analyst
Hierarchy and Organization Details
--------------------------
Hierarchy            Description
--------- -----------------
CEO/COO              Office of CEO ABC Inc.
CEO/COO              Office of CEO XYZ Inc.
CEO/COO              Office of CEO DataPro Inc.
VP                   Office of VP Sales ABC Inc.
VP                   Office of VP Mktg ABC Inc.
VP                   Office of VP Tech ABC Inc.

PL/SQL procedure successfully completed.
```

The following points are worth noting:

- The same cursor variable v_rc is used to point to two different queries.
- After you open v_rc for the first query and fetch the results, v_rc isn't closed. It's simply reopened for a second query and a new resultset is identified.

TIP *Once you've opened a cursor variable for a query, the resultset is fixed. You have to reopen the cursor variable to make it point to a different query.*

TIP *You don't need to close a cursor variable before you reopen it for a different query.*

Example 2: Assigning Different Queries to the Same Cursor Variable Depending on Runtime Conditions

Consider a scenario where a report is required of all organizations and their hierarchy levels depending on different conditions, such as the following:

- All organizations that are located in more than one site
- All organizations that don't have a particular hierarchy level
- All organizations that belong to the highest hierarchy level
- All organizations having the same hierarchy as those in a particular site

In this case, it suffices to use a single cursor variable that can be opened for different SELECT statements depending on the report option. I implement this as a SQL procedure (a stored subprogram) that takes the report option as the parameter.

CROSS-REFERENCE *Chapter 5 covers stored subprograms.*

Here's the code for the procedure:

```
CREATE OR REPLACE PROCEDURE p_print_report(p_report_no NUMBER, p_title VARCHAR2)
IS
  TYPE rc IS REF CURSOR;
    v_rc  rc;
    v_hrc_descr VARCHAR2(20);
    v_org_short_name VARCHAR2(30);
BEGIN
    IF (p_report_no = 1) THEN
       OPEN v_rc FOR SELECT h.hrc_descr, o.org_short_name
                                    FROM   org_tab o, hrc_tab h
                                    WHERE o.hrc_code = h.hrc_code
                                        AND 1 < (SELECT count(os.site_no)
                                                 FROM  org_site_tab os
                                                 WHERE os.org_id = o.org_id);
    ELSIF (p_report_no = 2) THEN
       OPEN v_rc FOR SELECT h.hrc_descr, o.org_short_name
                                    FROM   org_tab o, hrc_tab h
                                    WHERE o.hrc_code = h.hrc_code
                                         AND NOT EXISTS
                                                  (SELECT *
                                                   FROM  org_tab o1
                                                   WHERE o1.org_id = o.org_id
                                                         AND o1.hrc_code = 2 );
    END IF;
    dbms_output.put_line(p_title);
    dbms_output.put_line(rpad('-', length(p_title),'-'));
    dbms_output.put_line(rpad('Hierarchy',20,' ')||' '||
                                       rpad('Description',30,' '));
    dbms_output.put_line(rpad('-',20,'-')||' '||rpad('-',30,'-'));
    LOOP
        FETCH v_rc INTO v_hrc_descr, v_org_short_name;
        EXIT WHEN v_rc%NOTFOUND;
        dbms_output.put_line(rpad(v_hrc_descr,20,' ')||' '||
                                            rpad(v_org_short_name,30,' '));
```

```
     END LOOP;
     CLOSE v_rc;
END p_print_report;
/
```

You can now execute this procedure in a SQL*Plus environment by passing the report number and the corresponding title.

For the first report mentioned previously, here's the code and its output:

```
 SQL> set serverout on;
SQL> exec p_print_report(1, 'List of Organizations located in more than one site')
List of Organizations located in more than one site
--------------------------
Hierarchy             Description
--------- --------------
VP                     Office of VP Sales ABC Inc.
VP                     Office of VP Mktg ABC Inc.

 PL/SQL procedure successfully completed.
```

For the second report mentioned previously, here's the code and its output:

```
 SQL> exec p_print_report(2, 'List of Organizations not having a VP')
List of Organizations not having a VP
------------------------
Hierarchy             Description
--------- ----------------
CEO/COO                Office of CEO ABC Inc.
CEO/COO                Office of CEO XYZ Inc.
CEO/COO                Office of CEO DataPro Inc.

 PL/SQL procedure successfully completed.
```

TIP *Cursor variables and cursors aren't interchangeable. One can't be used in place of the other.*

TIP *Cursor variables can't be stored in the database. That is, database table columns can't be of type REF CURSOR or SYS_REFCURSOR.*

Returning Resultsets from Stored Subprograms

You can use cursor variables to return resultsets from stored functions and procedures as well as packaged functions and procedures.

CROSS-REFERENCE *I discuss using returning resultsets from stored procedures in Chapter 5.*

Cursor Expressions

Oracle9*i* has incorporated the facility to nest cursors in PL/SQL cursor declarations in the form of cursor expressions. In this section, I discuss the method of declaring and using cursor expressions in PL/SQL 9*i*. I also outline the method of passing cursors as actual parameters to functions.

Why Use Cursor Expressions?

Cursor expressions eliminate the use of declaring and using multiple cursors and hence result in a more effective optimization scheme by the SQL engine as it involves only one SQL statement as opposed to multiple cursors, which result in multiple SQL statements. Also, cursor expressions eliminate the use of complicated joins involved in SQL SELECT statements. As a third benefit, Oracle9*i* removes the limitation of using cursor expressions in SQL embedded in PL/SQL code. Now you can use cursor expressions as part of PL/SQL cursors. Also, when you use dynamic SQL, you can use cursor expressions and fetch into REF CURSOR variables. In this case, they support complex binds and defines needed for REF CURSORS. This isn't supported by DBMS_SQL.

Declaring Cursor Expressions

Basically, a *cursor expression* is a cursor declaration in PL/SQL in which the cursor SELECT statement contains one column as a cursor. This results in the declaration of nested cursors. A cursor expression is declared using this syntax:

```
CURSOR <parent-cursor-name> is
  SELECT col_name, CURSOR (SELECT  . . . ) . . .
```

Here's an example of a cursor expression:

```
CURSOR csr_hierarchy IS
  SELECT h.hrc_descr,
         CURSOR(SELECT o.org_long_name
                FROM   org_tab o
                WHERE o.hrc_code = h.hrc_code) long_name
  FROM  hrc_tab h;
```

This provides the functionality of a single query returning sets of values from multiple tables.

Prior to Oracle9*i*, CURSOR subqueries were supported in top-level SQL SELECT statements only. For example, a SELECT statement such as this:

```
SELECT h.hrc_descr,
          CURSOR(SELECT o.org_long_name
                 FROM   org_tab o
                 WHERE o.hrc_code = h.hrc_code) long_name
  FROM  hrc_tab h;
```

runs perfectly well in releases prior to Oracle9*i*, with the following output in SQL*Plus:

```
SQL> SELECT h.hrc_descr,
  2           CURSOR(SELECT o.org_long_name
  3                  FROM   org_tab o
  4                  WHERE o.hrc_code = h.hrc_code) long_name
  5    FROM  hrc_tab h;

HRC_DESCR            LONG_NAME
---------- -----------
CEO/COO              CURSOR STATEMENT : 2

CURSOR STATEMENT : 2

ORG_LONG_NAME
-----------------------------
```

```
Office of CEO ABC Inc.
Office of CEO XYZ Inc.
Office of CEO DataPro Inc.

VP                  CURSOR STATEMENT : 2

CURSOR STATEMENT : 2

ORG_LONG_NAME
------------------------------------------
Office of VP Sales ABC Inc.
Office of VP Mktg ABC Inc.
Office of VP Tech ABC Inc.

Director            CURSOR STATEMENT : 2

CURSOR STATEMENT : 2

no rows selected


HRC_DESCR           LONG_NAME
---------- ------------
Manager             CURSOR STATEMENT : 2

CURSOR STATEMENT : 2

no rows selected


HRC_DESCR           LONG_NAME
---------- ------------
Analyst             CURSOR STATEMENT : 2

CURSOR STATEMENT : 2

no rows selected
```

However, before Oracle9*i*, declaring a cursor in PL/SQL with this SELECT statement resulted in the compilation error shown here:

```
SQL> DECLARE
  2    CURSOR c1 IS
  3      SELECT h.hrc_descr,
  4           CURSOR(SELECT o.org_long_name
  5                  FROM   org_tab o
  6                  WHERE o.hrc_code = h.hrc_code) long_name
  7    FROM  hrc_tab h;
  8  BEGIN
  9    NULL;
 10  END;
```

```
 11  /
         CURSOR(SELECT o.org_long_name
                *
 ERROR at line 4:
 ORA-06550: line 4, column 17:
 PLS-00103: Encountered the symbol "SELECT" when expecting one of the following:
 ( ) - + mod not null others <an identifier>
 <a double-quoted delimited-identifier> <a bind variable>
 table avg count current exists max min prior sql stddev sum
 variance execute multiset the both leading trailing forall
 year month DAY_ HOUR_ MINUTE_ second TIMEZONE_HOUR_
 TIMEZONE_MINUTE_ time timestamp interval date
 <a string literal with character set specification>
 <a number> <a single-quoted SQL stri
 ORA-06550: line 6, column 48:
 PLS-00103: Encountered the symbol "LONG_NAME" when expecting one of the
 following:
 ; return returning and or
```

In Oracle8*i* and earlier, you could achieve the same function in PL/SQL by using two cursors with corresponding cursor FOR LOOPs. Here's the code for the same:

```
BEGIN
   FOR  i  IN (SELECT hrc_code, hrc_descr FROM hrc_tab) LOOP
      FOR  j  IN (SELECT org_long_name
                         FROM    org_tab
                         WHERE hrc_code = i.hrc_code) LOOP
          dbms_output.put_line(i.hrc_descr||' '||j.org_long_name);
      END LOOP;
   END LOOP;
END;
/
```

Using a cursor expression has the advantage of using only one SELECT statement to achieve the result. As such, it is optimized more effectively. The method of using a cursor expression in PL/SQL 9*i* is explained in the next section, "Using Cursor Expressions."

TIP *Multiple nesting using the CURSOR (subquery SELECT) is allowed.*

A cursor expression isn't allowed for an implicit cursor, in a view declaration, or in a subquery of a parent query. It is allowed in a parent query (i.e., the outermost SELECT list of a query).

Using Cursor Expressions

As I mentioned earlier, a cursor expression enables a single query to return sets of values from multiple tables. Here are the steps for using a cursor expression:

1. Declare the cursor expression with nested cursors.

2. Open the parent cursor. There's no need to open the nested cursors.

3. Use nested loops that fetch first from the rows of the result set and then from any nested cursors within these rows.

4. Declare a REF CURSOR to hold the nested cursor resultset while fetching.

5. Close the parent cursor. There's no need to close the nested cursors.

I wrote a PL/SQL function to use the cursor expression declared here. Here's the code:

```
create or replace function f_cursor_exp return NUMBER
is
TYPE rc is REF CURSOR;
/* declare the cursor expression */
CURSOR csr_hierarchy IS
  SELECT h.hrc_descr,
         CURSOR(SELECT o.org_long_name
                FROM   org_tab o
                WHERE o.hrc_code = h.hrc_code) long_name
  FROM  hrc_tab h;
/* Declare a REF CURSOR variable to hold the nested cursor resultset
    while fetching. */
hrc_rec rc;
v_hrc_descr VARCHAR2(20);
v_org_long_name VARCHAR2(60);
BEGIN
  /* Open the parent cursor */
  OPEN csr_hierarchy;
  LOOP
```

```
/* fetch the column csr_hierarchy.hrc_descr,
     then loop through the resultset of the nested cursor. */
     FETCH csr_hierarchy INTO v_hrc_descr, hrc_rec;
     EXIT WHEN csr_hierarchy%notfound;
/* Use a nested loop that fetches from the nested cursor
     within the parent rows. */
     LOOP
       -- Directly fetch from the nested cursor, there is no need to open it.
       FETCH hrc_rec INTO v_org_long_name;
       EXIT WHEN hrc_rec%notfound;
       DBMS_OUTPUT.PUT_LINE(v_hrc_descr ||' '||v_org_long_name);
     END LOOP;
  END LOOP;
/* Close the parent cursor. No need to close the nested cursor. */
  close csr_hierarchy;
  RETURN (0);
EXCEPTION WHEN OTHERS THEN
  RETURN (SQLCODE);
END;
/
```

The following points are worth noting:

- There's no need to open the nested cursor. It's implicitly opened when a row is fetched from the parent cursor.
- There's no need to close the nested cursor. It's implicitly closed when the parent cursor is closed.

Cursor Expressions Using Multiple Levels of Nested Cursors

This example demonstrates multiple levels of nested cursors. In the following code, I display the complete hierarchy, org, and org-site details:

```
create or replace function f_cursor_exp_complex return NUMBER
is
TYPE rc is REF CURSOR;
/* declare the cursor expression */
CURSOR csr_hierarchy IS
  SELECT h.hrc_descr,
```

```
          CURSOR(SELECT o.org_long_name,
                        CURSOR (SELECT s.site_descr
                                FROM   org_site_tab os, site_tab s
                                WHERE  os.site_no = s.site_no
                                  AND  os.org_id = o.org_id) as site_name
                 FROM   org_tab o
                 WHERE o.hrc_code = h.hrc_code) long_name
  FROM  hrc_tab h;
/* Declare two REF CURSOR variables to hold the nested cursor resultset
   while fetching. */
hrc_rec rc;
org_rec rc;
v_hrc_descr VARCHAR2(20);
v_org_long_name VARCHAR2(60);
v_site_name VARCHAR2(20);
BEGIN
  /* Open the parent cursor */
  OPEN csr_hierarchy;
  LOOP
/* fetch the column csr_hierarchy.hrc_descr,
    then loop through the resultset of the nested cursors. */
    FETCH csr_hierarchy INTO v_hrc_descr, hrc_rec;
    EXIT WHEN csr_hierarchy%notfound;
    LOOP
/* Use a nested loop that fetches from the first nested cursor
    within the parent rows */
      FETCH hrc_rec INTO v_org_long_name, org_rec;
      EXIT WHEN hrc_rec%notfound;
      LOOP
  -- Directly fetch from the second nested cursor, there is no need to open it
        FETCH org_rec INTO v_site_name;
        EXIT WHEN org_rec%notfound;
        DBMS_OUTPUT.PUT_LINE(v_hrc_descr ||' '||v_org_long_name||' '||
                                                        v_site_name);
      END LOOP;
    END LOOP;
  END LOOP;
/* Close the parent cursor. No need to close the nested cursors. */
  close csr_hierarchy;
  RETURN (0);
EXCEPTION WHEN OTHERS THEN
  RETURN (SQLCODE);
END;
/
```

You can now execute this function as shown here:

```
SQL> set serverout on;
SQL>  VAR ret_code NUMBER;
SQL> exec :ret_code := f_cursor_exp_complex;
```

Cursor Expressions as Arguments to Functions Called from SQL

I mentioned earlier that you can use cursor variables as formal parameters to a function. Also, cursor expressions refer to actual cursors. Now the following question arises: Can cursor expressions be used as actual parameters to such functions having REF CURSORS or SYS_REFCURSOR as formal parameter types? The answer to this question is yes, provided the function is called in a top-level SQL statement only.

Consider the second example presented in the earlier section "Dynamism in Using Cursor Variables." It describes a scenario in which a report is required of all organizations and their hierarchy levels depending on different conditions such as

- All organizations that are located in more than one site
- All organizations that don't have a particular hierarchy level
- All organizations that belong to the highest hierarchy level
- All organizations having same hierarchy as those in a particular site

In this case, it suffices to write a function that takes a cursor expression as input along with the title of the report and generates the report. The cursor expression is passed as an actual parameter with different WHERE conditions each time, but the columns in the SELECT will be the same each time. Here's the code for this function:

```
CREATE OR REPLACE FUNCTION f_report(p_cursor SYS_REFCURSOR, p_title VARCHAR2)
RETURN NUMBER
IS
   v_hrc_descr VARCHAR2(20);
   v_org_short_name VARCHAR2(30);
   v_ret_code NUMBER;
BEGIN
```

```
        BEGIN
          dbms_output.put_line(p_title);
          dbms_output.put_line(rpad('Hierarchy',20,' ')||'  '||
                                              rpad('Organization',30,' '));
          dbms_output.put_line(rpad('-',20,'-')||'  '||rpad('-',30,'-'));
          LOOP
               FETCH p_cursor INTO v_hrc_descr, v_org_short_name;
               EXIT WHEN p_cursor%NOTFOUND;
               dbms_output.put_line(rpad(v_hrc_descr,20,' ')||'  '||
                              rpad(v_org_short_name,30,' '));
         END LOOP;
          v_ret_code := 1;
    EXCEPTION WHEN OTHERS THEN
         v_ret_code := SQLCODE;
    END;
     RETURN (v_ret_code);
END;
/
```

You can now invoke this function with a cursor expression as an actual parameter to generate the different reports mentioned previously. Here's the SELECT statement:

```
SELECT 'Report Generated on '||TO_CHAR(SYSDATE,'MM/DD/YYYY') "Report1"
FROM   DUAL
WHERE f_report(
             CURSOR(SELECT h.hrc_descr, o.org_short_name
                              FROM   hrc_tab h, org_tab o
                              WHERE o.hrc_code = h.hrc_code
                                    AND 1 < (SELECT count(os.site_no)
                                             FROM  org_site_tab os
                                             WHERE os.org_id = o.org_id)
                             ),
                  'List of Organizations located in more than one site'
                            ) = 1;
```

Because dbms_output.put_line is being called from inside a function used in a SQL SELECT, the output buffer should be flushed. You do this by executing a small procedure called "flush," as follows:

```
CREATE OR REPLACE PROCEDURE flush
IS
 BEGIN
       NULL;
END;
/
```

Here's the output of this SELECT statement after executing flush:

```
SQL> SELECT 'Report Generated on '||TO_CHAR(SYSDATE,'MM/DD/YYYY') "Report1"
  2  FROM   DUAL
  3  WHERE f_report(
  4              CURSOR(SELECT h.hrc_descr, o.org_short_name
  5                            FROM   hrc_tab h, org_tab o
  6                            WHERE o.hrc_code = h.hrc_code
  7                                  AND 1 < (SELECT count(os.site_no)
  8                                             FROM  org_site_tab os
  9                                            WHERE os.org_id = o.org_id)
 10            ),  'List of Organizations located in more than one site'
 11                           ) = 1;

Report1
----------------
Report Generated on 02/13/2002

SQL> exec flush
List of Organizations located in more than one site
Hierarchy             Organization
--------  --------------
VP                    Office of VP Sales ABC Inc.
VP                    Office of VP Mktg ABC Inc.

PL/SQL procedure successfully completed.
```

You can use the same function to generate a different report—for example, a report that contains a list of organizations that don't have a vice president (VP). In this case, the function is invoked with a different cursor expression. Here's the second SELECT statement:

```
SELECT 'Report Generated on '||TO_CHAR(SYSDATE,'MM/DD/YYYY') "Report2"
FROM   DUAL
WHERE f_report(
       CURSOR(SELECT h.hrc_descr, o.org_short_name
                            FROM   hrc_tab h, org_tab o
                            WHERE o.hrc_code = h.hrc_code
                                  AND NOT EXISTS  (SELECT *
```

```
                                        FROM  org_tab o1
                                        WHERE o1.org_id = o.org_id
                                          AND o1.hrc_code = 2 )
                    ), 'List of Organizations not having a VP'
                   ) = 1;
```

Here's the output of the second SELECT statement (the output buffer is flushed in this case also):

```
SQL> SELECT 'Report Generated on '||TO_CHAR(SYSDATE,'MM/DD/YYYY') "Report2"
  2  FROM   DUAL
  3  WHERE f_report(
  4        CURSOR(SELECT h.hrc_descr, o.org_short_name
  5                              FROM   hrc_tab h, org_tab o
  6                              WHERE o.hrc_code = h.hrc_code
  7                                    AND NOT EXISTS  (SELECT *
  8                                                     FROM  org_tab o1
  9                                             WHERE o1.org_id = o.org_id
 10                                                AND o1.hrc_code = 2 )
 11                       ), 'List of Organizations not having a VP'
 12                      ) = 1;

Report2
----------------
Report Generated on 02/13/2002

SQL> exec flush
List of Organizations not having a VP
Hierarchy             Organization
--------- ---------------
CEO/COO               Office of CEO ABC Inc.
CEO/COO               Office of CEO XYZ Inc.
CEO/COO               Office of CEO DataPro Inc.

PL/SQL procedure successfully completed.
```

Instead of using the function f_report with dbms_output.put_line called to display output, you can directly generate the output using a SELECT column list. For this I use the following function:

```
CREATE OR REPLACE FUNCTION f_cursor(p_cursor SYS_REFCURSOR)
RETURN NUMBER
IS
   v_org_short_name VARCHAR2(30);
   v_cnt NUMBER := 0;
   v_ret_code NUMBER;
BEGIN
         BEGIN
            LOOP
                FETCH p_cursor INTO v_org_short_name;
                EXIT WHEN p_cursor%NOTFOUND;
                v_cnt := v_cnt + 1;
            END LOOP;
           IF (v_cnt  > 0) THEN
              v_ret_code := 1;
           ELSE
              v_ret_code := 0;
           END IF;
    EXCEPTION WHEN OTHERS THEN
         v_ret_code := SQLCODE;
    END;
     RETURN (v_ret_code);
END;
/
```

Then you can generate the first report by using the following SELECT statement (there's no need to flush the output buffer):

```
SELECT rpad(h.hrc_descr,20,' ') "Hierarchy",
               rpad(o.org_short_name,30,' ') "Organization"
FROM    hrc_tab h, org_tab o
WHERE h.hrc_code = o.hrc_code
       AND f_cursor(
             CURSOR(SELECT o1.org_short_name
                              FROM   org_tab o1
                              WHERE o1.org_id = o.org_id
                                     AND 1 < (SELECT count(os.site_no)
                                              FROM  org_site_tab os
                                              WHERE os.org_id = o1.org_id)
                             )
                            ) = 1;
```

Here's the output generated:

```
SQL> SELECT rpad(h.hrc_descr,20,' ') "Hierarchy",
  2                   rpad(o.org_short_name,30,' ') "Organization"
  3  FROM    hrc_tab h, org_tab o
  4  WHERE h.hrc_code = o.hrc_code
  5        AND f_cursor(
  6              CURSOR(SELECT o1.org_short_name
  7                                FROM   org_tab o1
  8                                WHERE o1.org_id = o.org_id
  9                                     AND 1 < (SELECT count(os.site_no)
 10                                              FROM  org_site_tab os
 11                                             WHERE os.org_id = o1.org_id)
 12                               )
 13                              ) = 1;

Hierarchy            Organization
-------- ----------------
VP                   Office of VP Sales ABC Inc.
VP                   Office of VP Mktg ABC Inc.
```

Similarly, you can generate the second report by using the following SELECT (there's no need to flush the output buffer):

```
SELECT rpad(h.hrc_descr,20,' ') "Hierarchy",
               rpad(o.org_short_name,30,' ') "Organization"
FROM    hrc_tab h, org_tab o
WHERE h.hrc_code = o.hrc_code
       AND f_cursor(
             CURSOR(SELECT o1.org_short_name
                               FROM   org_tab o1
                               WHERE o1.org_id = o.org_id
                                      AND NOT EXISTS  (SELECT *
                                               FROM  org_tab o2
                                               WHERE o2.org_id = o1.org_id
                                                     AND o2.hrc_code = 2 )
                      )
                     ) = 1;
```

Here's the output generated:

```
SQL> SELECT rpad(h.hrc_descr,20,' ') "Hierarchy",
  2       rpad(o.org_short_name,30,' ') "Organization"
  3  FROM    hrc_tab h, org_tab o
  4  WHERE h.hrc_code = o.hrc_code
  5        AND f_cursor(
  6              CURSOR(SELECT o1.org_short_name
  7                                FROM   org_tab o1
  8                                WHERE o1.org_id = o.org_id
  9                                      AND NOT EXISTS  (SELECT *
 10                                                       FROM  org_tab o2
 11                                            WHERE o2.org_id = o1.org_id
 12                                              AND o2.hrc_code = 2 )
 13                      )
 14                     ) = 1;

Hierarchy            Organization
-------- ---------------
CEO/COO              Office of CEO ABC Inc.
CEO/COO              Office of CEO XYZ Inc.
CEO/COO              Office of CEO DataPro Inc.

PL/SQL procedure successfully completed.
```

Both the functions f_report and f_cursor are invoked by passing a cursor expression as an actual parameter.

You can't use cursor expressions as actual parameters to functions with formal parameters of type REF CURSOR or SYS_REFCURSOR if the function is called in PL/SQL. For example, the following code is invalid:

```
DECLARE
      v_num NUMBER;
BEGIN
      v_num := f_report(
              CURSOR(SELECT h.hrc_descr, o.org_long_name
                                FROM   hrc_tab h, org_tab o
                                WHERE o.hrc_code = h.hrc_code
                                       AND 1 < (SELECT count(os.site_no)
                                                FROM  org_site_tab os
                                                WHERE os.org_id = o.org_id)
                              ),
                        'List of Organizations located in more than one site'
                  ) ;
END;
/
```

The preceding code raises the following error:

```
SQL> DECLARE
  2       v_num NUMBER;
  3  BEGIN
  4       v_num := f_report(
  5              CURSOR(SELECT h.hrc_descr, o.org_long_name
  6                                FROM   hrc_tab h, org_tab o
  7                                WHERE o.hrc_code = h.hrc_code
  8                                      AND 1 < (SELECT count(os.site_no)
  9                                               FROM  org_site_tab os
 10                                             WHERE os.org_id = o.org_id)
 11                               ),
 12                'List of Organizations located in more than one site'
 13                   ) ;
 14  END;
 15  /
            CURSOR(SELECT h.hrc_descr, o.org_long_name
            *
ERROR at line 5:
ORA-06550: line 5, column 13:
PLS-00405: subquery not allowed in this context
ORA-06550: line 4, column 6:
PL/SQL: Statement ignored
```

Summary

This chapter thoroughly covered PL/SQL cursors. You learned the various methods of using cursors and cursor variables in a PL/SQL environment. You also learned about cursor expressions, a new feature of PL/SQL 9*i*.

The next chapter presents a discussion of user-defined record types and index-by tables in PL/SQL.