CHAPTER 15
Data Retrieval

One of the hallmarks of the PL/SQL language is its tight integration with the Oracle database, both for changing data in database tables and for extracting information from those tables. This chapter explores the many features available in PL/SQL to query data from the database and make that data available within PL/SQL programs.

When you execute a SQL statement from PL/SQL, the Oracle database assigns a private work area for that statement and also manages the data specified by the SQL statement in the system global area (SGA). The private work area contains information about the SQL statement and the set of data returned or affected by that statement.

PL/SQL provides a number of ways to name this work area and manipulate the information within it, all of which involve defining and working with cursors. They include:

Implicit cursors
: A simple and direct SELECT...INTO retrieves a single row of data into local program variables. It's the easiest (and often the most efficient) path to your data, but it can often lead to coding the same or similar SELECTs in multiple places in your code.

Explicit cursors
: You can declare the query explicitly in your declaration section (local block or package). In this way, you can open and fetch from the cursor in one or more programs, with a granularity of control not available with implicit cursors.

Cursor variables
: Offering an additional level of flexibility, cursor variables (declared from a REF CURSOR type) allow you to pass a *pointer* to a query's underlying result set from one program to another. Any program with access to that variable can open, fetch from, or close the cursor.

Cursor expressions
: The CURSOR expression transforms a SELECT statement into a REF CURSOR result set and can be used with table functions to improve the performance of applications.

Dynamic SQL queries
: Oracle allows you to construct and execute queries dynamically at runtime using either native dynamic SQL (a.k.a. NDS, covered in Chapter 16) or DBMS_SQL. Details on this built-in package are available in the Oracle documentation as well as in *Oracle Built-in Packages*.

This chapter explores implicit cursors, explicit cursors, cursor variables, and cursor expressions in detail.

Cursor Basics

In its simplest form, a *cursor* is a pointer to the results of a query run against one or more tables in the database. For example, the following cursor declaration associates the entire employee table with the cursor named employee_cur:

```
CURSOR employee_cur IS SELECT * FROM employee;
```

Once I have declared the cursor, I can open it:

```
OPEN employee_cur;
```

Then I can fetch rows from it:

```
FETCH employee_cur INTO employee_rec;
```

Finally, I can close the cursor:

```
CLOSE employee_cur;
```

In this case, each record fetched from this cursor represents an entire record in the employee table. You can, however, associate any valid SELECT statement with a cursor. In the next example I have a join of three tables in my cursor declaration:

```
DECLARE
   CURSOR joke_feedback_cur
   IS
      SELECT J.name, R.laugh_volume, C.name
        FROM joke J, response R, comedian C
       WHERE J.joke_id = R.joke_id
         AND R.joker_id = C.joker_id;
BEGIN
   ...
END;
```

Here, the cursor does not act as a pointer into any actual table in the database. Instead, the cursor is a pointer into the virtual table or implicit view represented by the SELECT statement (SELECT is called a virtual table because the data it produces has the same structure as a table—rows and columns—but it exists only for the duration of the execution of the SQL statement). If the triple join returns 20 rows, each containing three columns, then the cursor functions as a pointer into those 20 rows.

Some Data Retrieval Terms

You have lots of options in PL/SQL for executing SQL, and all of them occur as some type of cursor inside your PL/SQL program. Before we dive into the details of the various approaches, this section will familiarize you with the types and terminology of data retrieval:

Static SQL
: A SQL statement is *static* if it is fully specified, or fixed, at the time the code containing that statement is compiled.

Dynamic SQL
: A SQL statement is *dynamic* if it is constructed at runtime and then executed, so you don't completely specify the SQL statement in the code you write. You can execute dynamic SQL either through the use of the built-in DBMS_SQL package or with native dynamic SQL.

Result set
: This is the set of rows identified by the database as fulfilling the request for data specified by the SQL statement. The result set is cached in the SGA to improve the performance of accessing and modifying the data in that set. The database maintains a pointer into the result set, which I will refer to in this chapter as the *current row*.

Implicit cursor
: PL/SQL declares and manages an implicit cursor every time you execute a SQL DML statement (INSERT, UPDATE, MERGE, or DELETE) or a SELECT INTO that returns a single row from the database directly into a PL/SQL data structure. This kind of cursor is called *implicit* because the database automatically handles many of the cursor-related operations for you, such as allocating a cursor, opening the cursor, fetching records, and even closing the cursor (although this is not an excuse to write code that relies on this behavior).

Explicit cursor
: This is a SELECT statement that you declare as a cursor explicitly in your application code. You then also explicitly perform each operation against that cursor (open, fetch, close, etc.). You will generally use explicit cursors when you need to retrieve multiple rows from data sources using static SQL.

Cursor variable
: This is a variable you declare that references or points to a cursor object in the database. As a true variable, a cursor variable can change its value (i.e., the cursor

or result set it points to) as your program executes. The variable can refer to different cursor objects (queries) at different times. You can also pass a cursor variable as a parameter to a procedure or function. Cursor variables are very useful when you're passing result set information from a PL/SQL program to another environment, such as Java or Visual Basic.

Cursor attribute
A cursor attribute takes the form %*attribute_name* and is appended to the name of a cursor or cursor variable. The attribute returns information about the state of the cursor, such as "Is the cursor open?" and "How many rows have been retrieved for this cursor?" Cursor attributes work in slightly different ways for implicit and explicit cursors and for dynamic SQL. These variations are explored throughout this chapter.

SELECT FOR UPDATE
This statement is a special variation of the normal SELECT, which proactively issues row locks on each row of data retrieved by the query. Use SELECT FOR UPDATE only when you need to reserve data you are querying to ensure that no one changes the data while you are processing it.

Bulk processing
In Oracle8*i* Database and later, PL/SQL offers the BULK COLLECT syntax for queries that allows you to fetch multiple rows from the database in a single or bulk step.

Typical Query Operations

Regardless of the type of cursor, PL/SQL performs the same operations to execute a SQL statement from within your program. In some cases, PL/SQL takes these steps for you. In others, such as with explicit cursors, you will code and execute these steps yourself:

Parse
The first step in processing a SQL statement is to parse it to make sure it is valid and to determine the execution plan (using either the rule- or cost-based optimizer, depending on how your DBA has set the OPTIMIZER_MODE parameter for your database, database statistics, query hints, etc.).

Bind
When you bind, you associate values from your program (host variables) with placeholders inside your SQL statement. With static SQL, the PL/SQL engine itself performs these binds. With dynamic SQL, you must explicitly request a binding of variable values if you want to use bind variables.

Open
When you open a cursor, the result set for the SQL statement is determined using any bind variables that have been set. The pointer to the active or current row is set

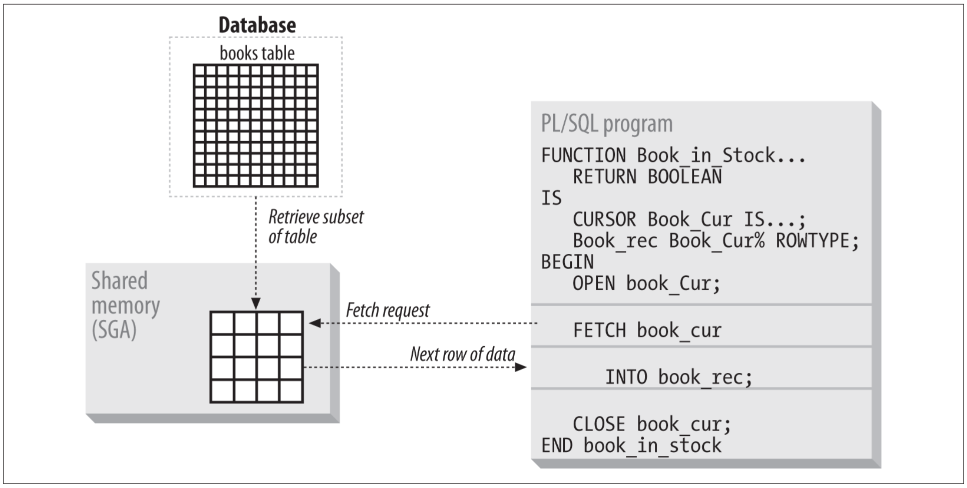
to the first row. Sometimes you will not explicitly open a cursor; instead, the PL/SQL engine will perform this operation for you (as with implicit cursors or native dynamic SQL).

Execute
: In the execute phase, the statement is run within the SQL engine.

Fetch
: If you are performing a query, the FETCH command retrieves the next row from the cursor's result set. Each time you fetch, PL/SQL moves the pointer forward in the result set. When you are working with explicit cursors, remember that FETCH does nothing (does not raise an error) if there are no more rows to retrieve—you must use cursor attributes to identify this condition.

Close
: This step closes the cursor and releases all memory used by the cursor. Once closed, the cursor no longer has a result set. Sometimes you will not explicitly close a cursor; instead, the PL/SQL engine will perform this operation for you (as with implicit cursors or native dynamic SQL).

Figure 15-1 shows how some of these different operations are used to fetch information from the database into your PL/SQL program.



```
FUNCTION Book_in_Stock...
   RETURN BOOLEAN
IS
   CURSOR Book_Cur IS...;
   Book_rec Book_Cur% ROWTYPE;
BEGIN
   OPEN book_Cur;

   FETCH book_cur

      INTO book_rec;

   CLOSE book_cur;
END book_in_stock
```

Figure 15-1. Simplified view of cursor fetch operation

Introduction to Cursor Attributes

This section describes each of the different cursor attributes at a high level. They are explored in more detail for each of the kinds of cursors throughout this chapter, as well as in Chapter 14 and Chapter 16.

PL/SQL offers a total of six cursor attributes, as shown in Table 15-1.

Table 15-1. Cursor attributes

Name	Description
%FOUND	Returns TRUE if the record was fetched successfully, and FALSE otherwise
%NOTFOUND	Returns TRUE if the record was not fetched successfully, and FALSE otherwise
%ROWCOUNT	Returns the number of records fetched from the cursor at that point in time
%ISOPEN	Returns TRUE if the cursor is open, and FALSE otherwise
%BULK_ROWCOUNT	Returns the number of records modified by the FORALL statement for each collection element
%BULK_EXCEPTIONS	Returns exception information for rows modified by the FORALL statement for each collection element

To reference a cursor attribute, attach it with "%" to the name of the cursor or cursor variable about which you want information, as in:

```
cursor_name%attribute_name
```

For implicit cursors, the cursor name is hardcoded as "SQL" (e.g., SQL%NOTFOUND).

The following sections offer brief descriptions of each cursor attribute.

The %FOUND attribute

The %FOUND attribute reports on the status of your most recent FETCH against the cursor. This attribute evaluates to TRUE if the most recent FETCH against the cursor returned a row, or FALSE if no row was returned.

If the cursor has not yet been opened, the database raises the INVALID_CURSOR exception.

In the following example, I loop through all the callers in the caller_cur cursor, assign all calls entered before today to that particular caller, and then fetch the next record. If I have reached the last record, then the explicit cursor's %FOUND attribute is set to FALSE, and I exit the simple loop. After my UPDATE statement, I check the implicit cursor's %FOUND attribute as well:

```
FOR caller_rec IN caller_cur
LOOP
      UPDATE call
      SET caller_id = caller_rec.caller_id
    WHERE call_timestamp < SYSDATE;

   IF SQL%FOUND THEN
      DBMS_OUTPUT.PUT_LINE ('Calls updated for ' || caller_rec.caller_id);
   END IF;
END LOOP;
```

The %NOTFOUND attribute

The %NOTFOUND attribute is the opposite of %FOUND. It returns TRUE if the most recent FETCH against the cursor did not return a row, often because the final row has already been fetched. If the cursor is unable to return a row because of an error, the appropriate exception is raised.

If the cursor has not yet been opened, the database raises the INVALID_CURSOR exception.

When should you use %FOUND and when should you use %NOTFOUND? Use whichever formulation fits most naturally in your code. In the previous example, I issued the following statement to exit my loop:

```
EXIT WHEN NOT caller_cur%FOUND;
```

An alternate and perhaps more readable formulation might use %NOTFOUND instead, as follows:

```
EXIT WHEN caller_cur%NOTFOUND;
```

The %ROWCOUNT attribute

The %ROWCOUNT attribute returns the number of records fetched so far from a cursor at the time the attribute is queried. When you first open a cursor, its %ROWCOUNT is set to zero. If you reference the %ROWCOUNT attribute of a cursor that is not open, you will raise the INVALID_CURSOR exception. After each record is fetched, %ROWCOUNT is increased by one.

Use %ROWCOUNT to verify that the expected number of rows have been fetched (or updated, in the case of DML) or to stop your program from executing after a certain number of iterations.

Here is an example:

```
BEGIN
   UPDATE employees SET last_name = 'FEUERSTEIN';

   DBMS_OUTPUT.PUT_LINE (SQL%ROWCOUNT);
END;
```

The %ISOPEN attribute

The %ISOPEN attribute returns TRUE if the cursor is open; otherwise, it returns FALSE. Here is an example of a common usage, making sure that cursors aren't left open when something unexpected occurs:

```
DECLARE
   CURSOR happiness_cur IS SELECT simple_delights FROM ...;
BEGIN
   OPEN happiness_cur;
```

```
      ...
      IF happiness_cur%ISOPEN THEN ...
   EXCEPTION
         WHEN OTHERS THEN
              IF happiness_cur%ISOPEN THEN
                    close happiness_cur;
              END IF;
   END;
```

The %BULK_ROWCOUNT attribute

The %BULK_ROWCOUNT attribute, designed for use with the FORALL statement, returns the number of rows processed by each DML execution. This attribute has the semantics of an associative array. It is covered in Chapter 21.

The %BULK_EXCEPTIONS attribute

The %BULK_EXCEPTIONS attribute, designed for use with the FORALL statement, returns exception information that may have been raised by each DML execution. This attribute (covered in Chapter 21) has the semantics of an associative array of records.

You can reference cursor attributes in your PL/SQL code, as shown in the preceding examples, but you cannot use those attributes inside a SQL statement. For example, if you try to use the %ROWCOUNT attribute in the WHERE clause of a SELECT:

```
SELECT caller_id, company_id FROM caller
 WHERE company_id = company_cur%ROWCOUNT;
```

you will get the compile error *PLS-00229: Attribute expression within SQL expression.*

Referencing PL/SQL Variables in a Cursor

Since a cursor must be associated with a SQL statement, every cursor must reference at least one table from the database and determine from that (and from the WHERE clause) which rows will be returned in the active set. This does not mean, however, that a PL/SQL cursor's SELECT may return only database information.

The list of expressions that appears after the SELECT keyword and before the FROM keyword is called the *select list*. In native SQL, this select list may contain both columns and expressions (SQL functions on those columns, constants, etc.). In PL/SQL, the select list of a SELECT may contain PL/SQL variables and complex expressions.

You can reference local PL/SQL program data (PL/SQL variables and constants) as well as host language bind variables in the WHERE, GROUP BY, and HAVING clauses of the cursor's SELECT statement. You can and should also *qualify* a reference to a

PL/SQL variable with its scope name (procedure name, package name, etc.), especially within a SQL statement. For more information on this topic, check out “Scope”.

Choosing Between Explicit and Implicit Cursors

In years past, it was common for “Oracle gurus” (including yours truly) to solemnly declare that you should *never* use implicit cursors for single-row fetches, and then explain that implicit cursors follow the ISO standard and always perform two fetches, making them less efficient than explicit cursors (for which you can just fetch a single time).

The first two editions of this book repeated that “wisdom,” but in the third edition we broke from tradition (along with many others). The bottom line is that from Oracle8 Database onward, as a result of very specific optimizations, it is very likely that your implicit cursor will now run *more*—not less—efficiently than the equivalent explicit cursor.

So does that mean that you should now always use implicit cursors, just as previously you should “always” have used explicit cursors? Not at all. There are still good reasons to use explicit cursors, including the following:

- In some cases, explicit cursors can still be more efficient. You should test your critical, often-executed queries in both formats to see which will be better in that particular situation.
- Explicit cursors offer much tighter programmatic control. If a row is not found, for example, the database will not raise an exception, instead forcing the execution block to shut down.

I suggest that the question to answer is not “implicit or explicit?” but rather, “encapsulate or expose?” And the answer is (new wisdom revealed):

> You should always encapsulate a single-row query, hiding the query behind a function interface and passing back the data through the RETURN clause.

In other words, don’t worry about explicit versus implicit. Instead, worry about how you can tune and maintain your code if single-row queries are duplicated throughout it.

And *stop* worrying by taking the time to encapsulate the queries behind functions, preferably package-based functions. Then you and all the other developers on your team can simply call the function in question whenever that data is needed. If Oracle ever changes its query behavior, rendering your previous “best practice” less than best, just change the implementation of that single function. Everyone’s code will immediately benefit!

Working with Implicit Cursors

PL/SQL declares and manages an implicit cursor every time you execute a SQL DML statement (INSERT, UPDATE, MERGE, or DELETE) or a SELECT INTO that returns data from the database directly into a PL/SQL data structure. This kind of cursor is called *implicit* because the database implicitly or automatically handles many of the cursor-related operations for you, such as allocating memory for a cursor, opening the cursor, fetching, and so on.

> Implicit DML statements are covered in Chapter 14. This chapter is concerned only with implicit SQL queries.

An implicit cursor is a SELECT statement that has these special characteristics:

- The SELECT statement appears in the executable section of your block; it is not defined in the declaration section, as explicit cursors are.
- The query contains an INTO clause (or BULK COLLECT INTO for bulk processing). The INTO clause is a part of the PL/SQL (not the SQL) language and is the mechanism used to transfer data from the database into local PL/SQL data structures.
- You do not open, fetch, or close the SELECT statement; all of these operations are done for you.

The general structure of an implicit query is as follows:

```
SELECT column_list
  [BULK COLLECT] INTO PL/SQL variable list ...rest of SELECT statement...
```

If you use an implicit cursor, the database performs the open, fetches, and close for you automatically; these actions are outside your programmatic control. You can, however, obtain information about the most recently executed SQL statement by examining the values in the implicit SQL cursor attributes, as explained later in this chapter.

> In the following sections, the term *implicit cursor* means a SELECT INTO statement that retrieves (or attempts to retrieve) a single row of data. In Chapter 21, I'll discuss the SELECT BULK COLLECT INTO variation that allows you to retrieve multiple rows of data with a single implicit query.

Implicit Cursor Examples

A common use of implicit cursors is to perform a lookup based on a primary key. In the following example, I look up the title of a book based on its ISBN:

```
DECLARE
   l_title books.title%TYPE;
BEGIN
   SELECT title
     INTO l_title
     FROM books
    WHERE isbn = '0-596-00121-5';
```

Once I have fetched the title into my local variable, l_title, I can manipulate that information—for example, by changing the variable's value, displaying the title, or passing the title on to another PL/SQL program for processing.

Here is an example of an implicit query that retrieves an entire row of information into a record:

```
DECLARE
   l_book books%ROWTYPE;
BEGIN
   SELECT *
     INTO l_book
     FROM books
    WHERE isbn = '0-596-00121-5';
```

You can also retrieve group-level information from a query. The following single-row query calculates and returns the total salary for a department. Once again, PL/SQL creates an implicit cursor for this statement:

```
SELECT SUM (salary)
  INTO department_total
  FROM employees
 WHERE department_id = 10;
```

Because PL/SQL is so tightly integrated with the Oracle database, you can also easily retrieve complex datatypes, such as objects and collections, within your implicit cursor.

All of these examples illustrate the use of implicit queries to retrieve a single row's worth of information. If you want to retrieve more than one row, you must either use an explicit cursor for that query or use the BULK COLLECT INTO clause (discussed in Chapter 21) in your query.

As mentioned earlier, I recommend that you always "hide" single-row queries like those just shown behind a function interface. This concept was explored in detail in "Choosing Between Explicit and Implicit Cursors" on page 493.

Error Handling with Implicit Cursors

The implicit cursor version of the SELECT statement is kind of a black box. You pass the SQL statement to the SQL engine in the database, and it returns a single row of information. You can't get inside the separate operations of the cursor, such as the open, fetch, and close stages. You are also stuck with the fact that the Oracle database automatically raises exceptions from within the implicit SELECT for two common outcomes:

- The query does not find any rows matching your criteria. In this case, the database raises the NO_DATA_FOUND exception.
- The SELECT statement returns more than one row. In this case, the database raises the TOO_MANY_ROWS exception.

When either of these scenarios occurs (as well as any other exceptions raised when you're executing a SQL statement), execution of the current block terminates and control is passed to the exception section. You have no control over this process flow; you cannot tell the database that with this implicit cursor you actually expect not to find any rows, and it is not an error. Instead, whenever you code an implicit cursor (and, therefore, are expecting to retrieve just one row of data), you should include an exception section that traps and handles these two exceptions (and perhaps others, depending on your application logic).

In the following block of code, I query the title of a book based on its ISBN, but I also anticipate the possible problems that may arise:

```
DECLARE
   l_isbn books.isbn%TYPE := '0-596-00121-5';
   l_title books.title%TYPE;
BEGIN
   SELECT title
     INTO l_title
     FROM books
    WHERE isbn = l_isbn;
EXCEPTION
   WHEN NO_DATA_FOUND
   THEN
      DBMS_OUTPUT.PUT_LINE ('Unknown book: ' || l_isbn);
   WHEN TOO_MANY_ROWS
   THEN
      /* This package defined in errpkg.pkg */
      errpkg.record_and_stop ('Data integrity error for: ' || l_isbn);
      RAISE;
END;
```

One of the problems with using implicit queries is that there is an awesome temptation to make assumptions about the data being retrieved, such as:

- “There can never possibly be more than one row in the book table for a given ISBN; we have constraints in place to guarantee that.”
- “There will always be an entry in the book table for Steven and Bill’s *Oracle PL/SQL Programming*. I don’t have to worry about NO_DATA_FOUND.”

The consequence of such assumptions is often that we developers neglect to include exception handlers for our implicit queries.

Now, it may well be true that today, with the current set of data, a query will return only a single row. If the nature of the data ever changes, however, you may find that the SELECT statement that formerly identified a single row now returns several. Your program will raise an exception, the exception will not be properly handled, and this could cause problems in your code.

You should, as a rule, always include handlers for NO_DATA_FOUND and TOO_MANY_ROWS whenever you write an implicit query. More generally, you should include error handlers for any errors that you can reasonably anticipate will occur in your program. The action you take when an error does arise will vary. Consider the code that retrieves a book title for an ISBN. In the following function, notice that my two error handlers act very differently: NO_DATA_FOUND returns a value, while TOO_MANY_ROWS logs the error and re-raises the exception, causing the function to actually fail (see Chapter 6 for more information about the *errpkg.pkg* package):

```
FUNCTION book_title (isbn_in   IN   books.isbn%TYPE)
   RETURN books.title%TYPE
IS
   return_value   book.title%TYPE;
BEGIN
  SELECT title
    INTO return_value
    FROM books
    WHERE isbn = isbn_in;

    RETURN return_value;
EXCEPTION
    WHEN NO_DATA_FOUND
    THEN
      RETURN NULL;
    WHEN TOO_MANY_ROWS
    THEN
      errpkg.record_and_stop ('Data integrity error for: '
                || isbn_in);
      RAISE;
END;
```

Here is the reasoning behind these varied treatments: the point of my function is to return the name of a book, which can never be NULL. The function can also be used to validate an ISBN (e.g., “Does a book exist for this ISBN?”). For this reason, I really don’t want my function to raise an exception when no book is found for an ISBN; that may actually constitute a successful condition, depending on how the function is being used. The logic may be, “If a book does not exist with this ISBN, then it can be used for a new book,” which might be coded as:

```
IF book_title ('0-596-00121-7') IS NULL
THEN ...
```

In other words, the fact that no book exists for that ISBN is not an error and should not be treated as one within my general lookup function.

On the other hand, if the query raises the TOO_MANY_ROWS exception, I have a real problem: there should never be two different books with the same ISBN. So, in this case, I need to log the error and then stop the application.

Implicit SQL Cursor Attributes

The Oracle database allows you to access information about the most recently executed implicit cursor by referencing the special implicit cursor attributes shown in Table 15-2. The table describes the significance of the values returned by these attributes for an implicit SQL query (SELECT INTO). Because the cursors are implicit, they have no name, and therefore the keyword “SQL” is used to denote the implicit cursor.

Table 15-2. Implicit SQL cursor attributes for queries

Name	Description
SQL%FOUND	Returns TRUE if one row (or more, in the case of BULK COLLECT INTO) was fetched successfully, and FALSE otherwise (in which case the database will also raise the NO_DATA_FOUND exception).
SQL%NOTFOUND	Returns TRUE if a row was not fetched successfully (in which case the database will also raise the NO_DATA_FOUND exception), and FALSE otherwise.
SQL%ROWCOUNT	Returns the number of rows fetched from the specified cursor thus far. For a SELECT INTO, this will be 1 if a row was found and 0 if the database raises the NO_DATA_FOUND exception.
SQL%ISOPEN	Always returns FALSE for implicit cursors because the database opens and closes implicit cursors automatically.

All the implicit cursor attributes return NULL if no implicit cursors have yet been executed in the session. Otherwise, the values of the attributes always refer to the most recently executed SQL statement, regardless of the block or program from which the SQL statement was executed. For more information about this behavior, see “Cursor Attributes for DML Operations” on page 466. You can also run the *query_implicit_attributes.sql* script on the book’s website to test out these values yourself.

Let’s make sure you understand the implications of this last point. Consider the following two programs:

```
PROCEDURE remove_from_circulation
   (isbn_in in books.isbn%TYPE)
IS
```

```
BEGIN
   DELETE FROM book WHERE isbn = isbn_in;
END;

PROCEDURE show_book_count
IS
   l_count   INTEGER;
BEGIN
   SELECT COUNT (*)
     INTO l_count
     FROM books;

   -- No such book!
   remove_from_circulation ('0-000-00000-0');

   DBMS_OUTPUT.put_line (SQL%ROWCOUNT);
END;
```

No matter how many rows of data are in the book table, I will always see "0" displayed in the output window. Because I call remove_from_circulation after my SELECT INTO statement, the SQL%ROWCOUNT attribute reflects the outcome of my silly, impossible DELETE statement, and not the query.

If you want to make certain that you are checking the values for the right SQL statement, you should save attribute values to local variables immediately after execution of the SQL statement. I demonstrate this technique in the following example:

```
PROCEDURE show_book_count
IS
   l_count   INTEGER;
   l_numfound PLS_INTEGER;
BEGIN
   SELECT COUNT (*)
     INTO l_count
     FROM books;

   -- Take snapshot of attribute value:
   l_numfound := SQL%ROWCOUNT;

   -- No such book!
   remove_from_circulation ('0-000-00000-0');

   -- Now I can go back to the previous attribute value.
   DBMS_OUTPUT.put_line (l_numfound);
END;
```

Working with Explicit Cursors

An explicit cursor is a SELECT statement that is explicitly defined in the declaration section of your code and, in the process, assigned a name. There is no such thing as an explicit cursor for INSERT, UPDATE, MERGE, and DELETE statements.

With explicit cursors, you have complete control over the different PL/SQL steps involved in retrieving information from the database. You decide when to OPEN the cursor, when to FETCH records from the cursor (and therefore from the table or tables in the SELECT statement of the cursor), how many records to fetch, and when to CLOSE the cursor. Information about the current state of your cursor is available through examination of cursor attributes. This granularity of control makes the explicit cursor an invaluable tool for your development effort.

Let's look at an example. The following function determines (and returns) the level of jealousy I should feel for my friends, based on their location:

```
FUNCTION jealousy_level (
   NAME_IN   IN   friends.NAME%TYPE) RETURN NUMBER
AS
   CURSOR jealousy_cur
   IS
      SELECT location FROM friends
       WHERE NAME = UPPER (NAME_IN);

   jealousy_rec   jealousy_cur%ROWTYPE;
   retval         NUMBER;
BEGIN
   OPEN jealousy_cur;

   FETCH jealousy_cur INTO jealousy_rec;

   IF jealousy_cur%FOUND
   THEN
      IF jealousy_rec.location = 'PUERTO RICO'
         THEN retval := 10;
      ELSIF jealousy_rec.location = 'CHICAGO'
         THEN retval := 1;
      END IF;
   END IF;

   CLOSE jealousy_cur;

   RETURN retval;
EXCEPTION
   WHEN OTHERS THEN
      IF jealousy_cur%ISOPEN THEN
         CLOSE jealousy_cur;
      END IF;
END;
```

This PL/SQL block performs the cursor actions outlined in the following table.

Line(s)	Action
4–7	Declare the cursor.
9	Declare a record based on that cursor.
12	Open the cursor.
14	Fetch a single row from the cursor.
16	Check a cursor attribute to determine if a row was found.
18–22	Examine the contents of the fetched row to calculate my level of jealousy.
25	Close the cursor.
28–32	Make sure that I clean up after myself in case something unexpected happens (precautionary code).

The next few sections examine each step in detail. In these sections, the word *cursor* refers to an explicit cursor unless otherwise noted.

Declaring Explicit Cursors

To use an explicit cursor, you must first declare it in the declaration section of your PL/SQL block or in a package, as shown here:

```
CURSOR cursor_name [ ( [ parameter [, parameter ...] ) ]
   [ RETURN return_specification ]
   IS SELECT_statement
       [FOR UPDATE [OF [column_list]];
```

where *cursor_name* is the name of the cursor, *return_specification* is an optional RETURN clause for the cursor, and *SELECT_statement* is any valid SQL SELECT statement. You can also pass arguments into a cursor through the optional parameter list described in "Cursor Parameters" on page 512. Finally, you can specify a list of columns that you intend to update after a SELECT...FOR UPDATE statement (also discussed later). Once you have declared a cursor, you can OPEN it and FETCH from it.

Here are some examples of explicit cursor declarations:

A cursor without parameters

The result set of this cursor contains all the company IDs in the table:

```
CURSOR company_cur IS
   SELECT company_id FROM company;
```

A cursor with parameters

The result set of this cursor is the name of the company that matches the company ID passed to the cursor via the parameter:

```
CURSOR name_cur (company_id_in IN NUMBER)
IS
   SELECT name FROM company
    WHERE company_id = company_id_in;
```

A cursor with a RETURN clause

The result set of this cursor is all columns (in the same structure as the underlying table) from all employee records in department 10:

```
CURSOR emp_cur RETURN employees%ROWTYPE
IS
   SELECT * FROM employees
    WHERE department_id = 10;
```

Naming your cursor

The name of an explicit cursor can be up to 30 characters in length and follows the rules for any other identifier in PL/SQL. A cursor name is not a PL/SQL variable. Instead, it is an undeclared identifier used to point to or refer to the query. You cannot assign values to a cursor, nor can you use it in an expression. You can only reference that explicit cursor by name within OPEN, FETCH, and CLOSE statements, and use it to qualify the reference to a cursor attribute.

Declaring cursors in packages

You can declare explicit cursors in any declaration section of a PL/SQL block. This means that you can declare such cursors within packages and at the package level, as well as within a subprogram in the package. I'll explore packages in general in Chapter 18. You may want to look ahead at that chapter to acquaint yourself with the basics of packages before plunging into the topic of declaring cursors in packages.

Here are two examples:

```
PACKAGE book_info
IS
   CURSOR titles_cur
   IS
      SELECT title
        FROM books;

   CURSOR books_cur (title_filter_in IN books.title%TYPE)
      RETURN books%ROWTYPE
   IS
      SELECT *
        FROM books
       WHERE title LIKE title_filter_in;
END;
```

The first cursor, titles_cur, returns just the titles of books. The second cursor, books_cur, returns a record for each row in the book table whose title passes the filter provided as a parameter (such as "All books that contain 'PL/SQL'"). Notice that the second cursor also utilizes the RETURN clause of a cursor, in essence declaring publicly the structure of the data that each FETCH against that cursor will return.

The RETURN clause of a cursor may be made up of any of the following datatype structures:

- A record defined from a database table, using the %ROWTYPE attribute
- A record defined from another, previously defined cursor, also using the %ROWTYPE attribute
- A record defined from a programmer-defined record

The number of expressions in the cursor's select list must match the number of columns in the record identified by *table_name*%ROWTYPE, *cursor*%ROWTYPE, or *record_type*. The datatypes of the elements must also be compatible. For example, if the second element in the select list is type NUMBER, then the second column in the RETURN record cannot be type VARCHAR2 or BOOLEAN.

Before exploring the RETURN clause and its advantages, let's first address a different question: why should you bother putting cursors into packages? Why not simply declare your explicit cursors wherever you need them directly in the declaration sections of particular procedures, functions, or anonymous blocks?

The answer is simple and persuasive. By defining cursors in packages, you can more easily reuse those queries and avoid writing the same logical retrieval statement over and over again throughout your application. By implementing that query in just one place and referencing it in many locations, you make it easier to enhance and maintain that query. You will also realize some performance gains by minimizing the number of times your queries will need to be parsed.

You should also consider creating a function that returns a cursor variable, based on a REF CURSOR. The calling program can then fetch rows through the cursor variable. See "Cursor Variables and REF CURSORs" on page 519 for more information.

If you declare cursors in packages for reuse, you need to be aware of one important factor. Data structures, including cursors, that are declared at the "package level" (not inside any particular function or procedure) maintain their values or persist for your entire session. This means that a packaged cursor will stay open until you explicitly close it or until your session ends. Cursors declared in local blocks of code close automatically when those blocks terminate execution.

Now let's explore this RETURN clause and why you might want to take advantage of it. One of the interesting variations on a cursor declaration within a package involves the ability to separate the cursor's header from its body. The header of a cursor, much like the header of a function, contains just that information a developer needs in order to write code to work with the cursor: the cursor's name, any parameters, and the type of data being returned. The body of a cursor is its SELECT statement.

Here is a rewrite of the books_cur cursor in the book_info package that illustrates this technique:

```
PACKAGE book_info
IS
   CURSOR books_cur (title_filter_in IN books.title%TYPE)
      RETURN books%ROWTYPE;
END;

PACKAGE BODY book_info
IS
   CURSOR books_cur (title_filter_in IN books.title%TYPE)
      RETURN books%ROWTYPE
   IS
      SELECT *
        FROM books
       WHERE title LIKE title_filter_in;
END;
```

Notice that everything up to but not including the IS keyword is the specification, while everything following the IS keyword is the body.

There are two reasons that you might want to divide your cursor as just shown:

To hide information
: Packaged cursors are essentially black boxes. This is advantageous to developers because they never have to code or even see the SELECT statement. They only need to know what records the cursor returns, in what order it returns them, and which columns are in the column list. They can simply use it as another predefined element in their applications.

To minimize recompilation
: If I hide the query definition inside the package body, I can make changes to the SELECT statement without making any changes to the cursor header in the package specification. This allows me to enhance, fix, and recompile my code without recompiling my specification, which means that all the programs dependent on that package will not be marked invalid and will not need to be recompiled.

Opening Explicit Cursors

The first step in using a cursor is to define it in the declaration section. The next step is to open that cursor. The syntax for the OPEN statement is simplicity itself:

```
OPEN cursor_name [ ( argument [, argument ...] ) ];
```

where *cursor_name* is the name of the cursor you declared, and the *argument*s are the values to be passed if the cursor was declared with a parameter list.

Oracle also offers the OPEN *cursor* FOR syntax, which is utilized in both cursor variables (see "Cursor Variables and REF CURSORs" on page 519) and native dynamic SQL (see Chapter 16).

When you open a cursor, PL/SQL executes the query for that cursor. It also identifies the active set of data—that is, the rows from all involved tables that meet the criteria in the WHERE clause and join conditions. The OPEN does not actually retrieve any of these rows; that action is performed by the FETCH statement.

Regardless of when you perform the first fetch, however, the read consistency model in the Oracle database guarantees that all fetches will reflect the data as it existed when the cursor was opened. In other words, from the moment you open your cursor until the moment that cursor is closed, all data fetched through the cursor will ignore any inserts, updates, and deletes performed by any active sessions after the cursor was opened.

Furthermore, if the SELECT statement in your cursor uses a FOR UPDATE clause, all the rows identified by the query are locked when the cursor is opened. (This feature is covered in the section "SELECT...FOR UPDATE" on page 515.)

If you try to open a cursor that is already open, you will get the following error:

```
ORA-06511: PL/SQL: cursor already open
```

You can be sure of a cursor's status by checking the %ISOPEN cursor attribute before you try to open the cursor:

```
IF NOT company_cur%ISOPEN
THEN
   OPEN company_cur;
END IF;
```

The section "Explicit Cursor Attributes" on page 510 explains the different cursor attributes and how to best use them in your programs.

If you are using a cursor FOR loop, you do not need to open (or fetch from or close) the cursor explicitly. Instead, the PL/SQL engine does that for you.

Fetching from Explicit Cursors

A SELECT statement establishes a virtual table; its return set is a series of rows determined by the WHERE clause (or lack thereof), with columns determined by the column list of the SELECT. So, a cursor represents that virtual table within your PL/SQL program. In almost every situation, the point of declaring and opening a cursor is to return, or fetch, the rows of data from the cursor and then manipulate the information retrieved. PL/SQL provides a FETCH statement for this action.

The general syntax for a FETCH is:

```
FETCH cursor_name INTO record_or_variable_list;
```

where *cursor_name* is the name of the cursor from which the record is fetched, and *record_or_variable_list* is the PL/SQL data structure(s) into which the next row of the active set of records is copied. You can fetch into a record structure (declared with the %ROWTYPE attribute or a TYPE declaration statement), or you can fetch into a list of one or more variables (PL/SQL variables or application-specific bind variables such as Oracle Forms items).

Examples of explicit cursors

The following examples illustrate the variety of possible fetches:

- Fetch into a PL/SQL record:

  ```
  DECLARE
     CURSOR company_cur is SELECT ...;
     company_rec company_cur%ROWTYPE;
  BEGIN
     OPEN company_cur;
     FETCH company_cur INTO company_rec;
  ```

- Fetch into a variable:

  ```
  FETCH new_balance_cur INTO new_balance_dollars;
  ```

- Fetch into a collection row, a variable, and an Oracle Forms bind variable:

  ```
  FETCH emp_name_cur INTO emp_name (1), hiredate, :dept.min_salary;
  ```

You should always fetch into a record that was defined with %ROWTYPE against the cursor; avoid fetching into lists of variables. Fetching into a record usually means that you write less code and have more flexibility to change the select list without having to change the FETCH statement.

Fetching past the last row

Once you open an explicit cursor, you can FETCH from it until there are no more records left in the active set. Oddly enough, though, you can also continue to FETCH past the last record.

In this case, PL/SQL will not raise any exceptions—it just won't actually be doing anything. Because there is nothing left to fetch, it will not alter the values of the variables in the INTO list of the FETCH. More specifically, the FETCH operation will not set those values to NULL.

You should therefore never test the values of INTO variables to determine if the FETCH against the cursor succeeded. Instead, you should check the value of the %FOUND or %NOTFOUND attribute, as explained in the section "Explicit Cursor Attributes" on page 510.

Column Aliases in Explicit Cursors

The SELECT statement of the cursor includes the list of columns that are returned by that cursor. As with any SELECT statement, this column list may contain either actual column names or column expressions, which are also referred to as *calculated* or *virtual columns*.

A *column alias* is an alternative name you provide to a column or column expression in a query. You may have used column aliases in SQL*Plus to improve the readability of ad hoc report output. In that situation, such aliases are completely optional. In an explicit cursor, on the other hand, column aliases are required for calculated columns when:

- You FETCH into a record declared with a %ROWTYPE declaration against that cursor.
- You want to reference the calculated column in your program.

Consider the following query. For all companies with sales activity during 2001, the SELECT statement retrieves the company name and the total amount invoiced to that company (assume that the default date format mask for this instance is DD-MON-YYYY):

```
SELECT company_name, SUM (inv_amt)
  FROM company c, invoice i
 WHERE c.company_id = i.company_id
   AND TO_CHAR (i.invoice_date, 'YYYY') = '2001';
```

The output is:

```
     COMPANY_NAME                         SUM (INV_AMT)
---------------                      -------------
ACME TURBO INC.                      1000
WASHINGTON HAIR CO.                  25.20
```

SUM (INV_AMT) does not make a particularly attractive column header for a report, but it works well enough for a quick dip into the data as an ad hoc query. Let's now use this same query in an explicit cursor and add a column alias:

```
DECLARE
   CURSOR comp_cur IS
```

```
      SELECT c.name, SUM (inv_amt) total_sales
        FROM company C, invoice I
       WHERE C.company_id = I.company_id
         AND TO_CHAR (i.invoice_date, 'YYYY') = '2001';
   comp_rec comp_cur%ROWTYPE;
BEGIN
   OPEN comp_cur;
   FETCH comp_cur INTO comp_rec;
   ...
END;
```

Without the alias, I have no way of referencing the column within the comp_rec record structure. With the alias in place, I can get at that information just as I would any other column or expression in the query:

```
IF comp_rec.total_sales > 5000
THEN
   DBMS_OUTPUT.PUT_LINE
      (' You have exceeded your credit limit of $5000 by ' ||
       TO_CHAR (comp_rec.total_sales - 5000, '$9999'));
END IF;
```

If you fetch a row into a record declared with %ROWTYPE, the only way to access the column or column expression value is by the column name; after all, the record obtains its structure from the cursor itself.

Closing Explicit Cursors

Early on I was taught to clean up after myself, and I tend to be a bit obsessive (albeit selectively) about this. Cleaning up after oneself is an important rule to follow in programming and can be crucial when it comes to cursor management. So be sure to close a cursor when you are done with it!

Here is the syntax for a CLOSE cursor statement:

```
CLOSE cursor_name;
```

where *cursor_name* is the name of the cursor you are closing.

Here are some special considerations regarding the closing of explicit cursors:

- If you declare and open a cursor in a program, be sure to close it when you are done. Otherwise, you may have just allowed a memory leak to creep into your code—and that's not good! Strictly speaking, a cursor (like any other data structure) should be automatically closed and destroyed when it goes out of scope. In fact, in many cases PL/SQL does check for and implicitly close any open cursors at the end of a procedure call, function call, or anonymous block. However, the overhead involved in doing that is significant, so for the sake of efficiency there are cases where PL/SQL does *not* immediately check for and close cursors opened in a PL/SQL block. In addition, REF CURSORs are, by design, never closed implicitly. The one thing you can count on is that whenever the outermost PL/SQL block ends and control is returned to SQL or some other calling program, PL/SQL will at that point implicitly close any cursors (but not REF CURSORs) left open by that block or nested blocks.

> Oracle Technology Network offers a detailed analysis of how and when PL/SQL closes cursors in an article titled "Cursor reuse in PL/SQL static SQL." Nested anonymous blocks provide an example of one case in which PL/SQL does not implicitly close cursors. For an interesting discussion of this issue, see Jonathan Gennick's article "Does PL/SQL Implicitly Close Cursors?".

- If you declare a cursor in a package at the package level and then open it in a particular block or program, that cursor will stay open until you explicitly close it or until your session closes. Therefore, it is extremely important that you include a CLOSE statement for any packaged cursors as soon as you are done with them (and in the exception section as well), as in the following:

```
BEGIN
   OPEN my_package.my_cursor;

   ... do stuff with the cursor ...

   CLOSE my_package.my_cursor;
EXCEPTION
   WHEN OTHERS
   THEN
      IF mypackage.my_cursor%ISOPEN THEN
           CLOSE my_package.my_cursor;
       END IF;
END;
```

- You can close a cursor only if it is currently open. Otherwise, the database will raise an INVALID_CURSOR exception. You can check a cursor's status with the %ISOPEN cursor attribute before you try to close the cursor:

```
IF company_cur%ISOPEN
THEN
   CLOSE company_cur;
END IF;
```

  Attempts to close a cursor that is already closed (or was never opened) will result in an *ORA-1001: Invalid cursor error.*

- If you leave too many cursors open, you may exceed the value set by the database initialization parameter, OPEN_CURSORS (the value is on a per-session basis). If this happens, you will encounter the dreaded error message *ORA-01000: maximum open cursors exceeded.*

If you get this message, check your usage of package-based cursors to make sure they are closed when no longer needed.

Explicit Cursor Attributes

Oracle offers four attributes (%FOUND, %NOTFOUND, %ISOPEN, %ROWCOUNT) that allow you to retrieve information about the state of your cursor. Reference these attributes using this syntax:

```
cursor%attribute
```

where *cursor* is the name of the cursor you have declared.

Table 15-3 describes the significance of the values returned by these attributes for explicit cursors.

Table 15-3. Values returned by cursor attributes

Name	Description
cursor%FOUND	Returns TRUE if a record was fetched successfully
cursor%NOTFOUND	Returns TRUE if a record was not fetched successfully
cursor%ROWCOUNT	Returns the number of records fetched from the specified cursor at that point in time
cursor%ISOPEN	Returns TRUE if the specified cursor is open

Table 15-4 shows you the attribute values you can expect to see both before and after the specified cursor operations.

Table 15-4. Cursor attribute values

Operation	%FOUND	%NOTFOUND	%ISOPEN	%ROWCOUNT
Before OPEN	ORA-01001 raised	ORA-01001 raised	FALSE	ORA-01001 raised
After OPEN	NULL	NULL	TRUE	0
Before first FETCH	NULL	NULL	TRUE	0
After first FETCH	TRUE	FALSE	TRUE	1
Before subsequent FETCH(es)	TRUE	FALSE	TRUE	1
After subsequent FETCH(es)	TRUE	FALSE	TRUE	Data-dependent
Before last FETCH	TRUE	FALSE	TRUE	Data-dependent
After last FETCH	FALSE	TRUE	TRUE	Data-dependent
Before CLOSE	FALSE	TRUE	TRUE	Data-dependent
After CLOSE	Exception	Exception	FALSE	Exception

Here are some things to keep in mind as you work with cursor attributes for explicit cursors:

- If you try to use %FOUND, %NOTFOUND, or %ROWCOUNT before the cursor is opened or after it is closed, the database will raise an INVALID_CURSOR error (ORA-01001).
- If the result set is empty after the very first FETCH, the attributes will return values as follows: %FOUND = FALSE, %NOTFOUND = TRUE, and %ROWCOUNT = 0.
- If you are using BULK COLLECT, %ROWCOUNT will return the number of rows fetched into the associated collections. For more details, see Chapter 21.

The following code showcases many of these attributes:

```
PACKAGE bookinfo_pkg
IS
   CURSOR bard_cur
       IS SELECT title, date_published
     FROM books
    WHERE UPPER(author) LIKE 'SHAKESPEARE%';
END bookinfo_pkg;

DECLARE
   bard_rec   bookinfo_pkg.bard_cur%ROWTYPE;
BEGIN
   /* Check to see if the cursor is already opened.
      This may be the case as it is a packaged cursor.
      If so, first close it and then reopen it to
      ensure a "fresh" result set.
   */
   IF bookinfo_pkg.bard_cur%ISOPEN
   THEN
      CLOSE bookinfo_pkg.bard_cur;
   END IF;

   OPEN bookinfo_pkg.bard_cur;

   -- Fetch each row, but stop when I've displayed the
   -- first five works by Shakespeare or when I have
   -- run out of rows.
   LOOP
      FETCH bookinfo_pkg.bard_cur INTO bard_rec;
      EXIT WHEN bookinfo_pkg.bard_cur%NOTFOUND
             OR bookinfo_pkg.bard_cur%ROWCOUNT > 5;
      DBMS_OUTPUT.put_line (
            bookinfo_pkg.bard_cur%ROWCOUNT
         || ') '
         || bard_rec.title
         || ', published in '
         || TO_CHAR (bard_rec.date_published, 'YYYY')
      );
   END LOOP;
   CLOSE bookinfo_pkg.bard_cur;
END;
```

Cursor Parameters

In this book you've already seen examples of the use of parameters with procedures and functions. Parameters provide a way to pass information into and out of a module. Used properly, parameters improve the usefulness and flexibility of modules.

PL/SQL allows you to pass parameters into cursors. The same rationale for using parameters in modules applies to parameters for cursors:

It makes the cursor more reusable
: Instead of hardcoding a value into the WHERE clause of a query to select particular information, you can use a parameter and then pass different values to the WHERE clause each time a cursor is opened.

It avoids scoping problems
: When you pass parameters instead of hardcoding values, the result set for that cursor is not tied to a specific variable in a program or block. If your program has nested blocks, you can define the cursor at a higher-level (enclosing) block and use it in any of the subblocks with variables defined in those local blocks.

You can specify as many cursor parameters as you need. When you OPEN the cursor, you need to include an argument in the parameter list for each parameter, except for trailing parameters that have default values.

When should you parameterize your cursors? I apply the same rule of thumb to cursors as to procedures and functions; if I am going to use the cursor in more than one place with different values for the same WHERE clause, I should create a parameter for the cursor.

Let's take a look at the difference between parameterized and unparameterized cursors. First, here is a cursor without any parameters:

```
CURSOR joke_cur IS
   SELECT name, category, last_used_date
     FROM jokes;
```

The result set of this cursor is all the rows in the jokes table. If I just wanted to retrieve all jokes in the HUSBAND category, I would need to add a WHERE clause:

```
CURSOR joke_cur IS
   SELECT name, category, last_used_date
     FROM jokes
    WHERE category = 'HUSBAND';
```

I didn't use a cursor parameter to accomplish this task, nor did I need to. The joke_cur cursor now retrieves only those jokes about husbands. That's all well and good, but what if I also wanted to see light-bulb jokes and then chicken-and-egg jokes and finally, as my 10-year-old niece would certainly demand, all my knock-knock jokes?

Generalizing cursors with parameters

I really don't want to write a separate cursor for each category—that is definitely not a data-driven approach to programming. Instead, I would much rather be able to change the joke cursor so that it can accept different categories and return the appropriate rows. The best (though not the only) way to do this is with a cursor parameter:

```
PROCEDURE explain_joke (main_category_in IN joke_category.category_id%TYPE)
IS
   /*
   || Cursor with parameter list consisting of a single
   || string parameter.
   */
   CURSOR joke_cur (category_in IN VARCHAR2)
   IS
      SELECT name, category, last_used_date
        FROM joke
       WHERE category = UPPER (category_in);

   joke_rec joke_cur%ROWTYPE;

BEGIN
   /* Now when I open the cursor, I also pass the argument. */
   OPEN joke_cur (main_category_in);
   FETCH joke_cur INTO joke_rec;
```

I added a parameter list after the cursor name and before the IS keyword. I took out the hardcoded "HUSBAND" and replaced it with "UPPER (category_in)" so that I could enter "HUSBAND", "husband", or "HuSbAnD" and the cursor would still work. Now when I open the cursor, I specify the value I want to pass as the category by including that value (which can be a literal, a constant, or an expression) inside parentheses. At the moment the cursor is opened, the SELECT statement is parsed and bound using the specified value for category_in. The result set is identified, and the cursor is ready for fetching.

Opening cursors with parameters

I can OPEN that same cursor with any category I like. Now I don't have to write a separate cursor to accommodate this requirement:

```
OPEN joke_cur (jokes_pkg.category);
OPEN joke_cur ('husband');
OPEN joke_cur ('politician');
OPEN joke_cur (jokes_pkg.relation || '-IN-LAW');
```

The most common place to use a parameter in a cursor is in the WHERE clause, but you can make reference to it anywhere in the SELECT statement, as shown here:

```
DECLARE
   CURSOR joke_cur (category_in IN VARCHAR2)
   IS
      SELECT name, category_in, last_used_date
        FROM joke
       WHERE category = UPPER (category_in);
```

Instead of returning the category from the table, I simply pass back the category_in parameter in the select list. The result will be the same either way because my WHERE clause restricts categories to the parameter value.

Scope of cursor parameters

The scope of the cursor parameter is confined to that cursor. You cannot refer to the cursor parameter outside of the SELECT statement associated with the cursor. The following PL/SQL fragment will not compile because the program_name identifier is not a local variable in the block. Instead, it is a formal parameter for the cursor and is defined only inside the cursor:

```
DECLARE
   CURSOR scariness_cur (program_name VARCHAR2)
   IS
      SELECT SUM (scary_level) total_scary_level
        FROM tales_from_the_crypt
       WHERE prog_name = program_name;
BEGIN
   program_name := 'THE BREATHING MUMMY'; /* Illegal reference */
   OPEN scariness_cur (program_name);
   ...
   CLOSE scariness_cur;
END;
```

Cursor parameter modes

The syntax for cursor parameters is very similar to that of procedures and functions, with the restriction that a cursor parameter can be an IN parameter only. You cannot specify OUT or IN OUT modes for cursor parameters. The OUT and IN OUT modes are used to pass values out of a procedure through that parameter. This doesn't make sense for a cursor. Values cannot be passed back out of a cursor through the parameter list. You can retrieve information from a cursor only by fetching a record and copying values from the column list with an INTO clause. (See Chapter 17 for more information on the parameter mode.)

Default values for parameters

Cursor parameters can be assigned default values. Here is an example of a parameterized cursor with a default value:

```
CURSOR emp_cur (emp_id_in NUMBER := 0)
IS
```

```
      SELECT employee_id, emp_name
        FROM employee
       WHERE employee_id = emp_id_in;
```

So, if Joe Smith's employee ID were 1001, the following statements would set my_emp_id to 1001 and my_emp_name to JOE SMITH:

```
OPEN emp_cur (1001);
FETCH emp_cur INTO my_emp_id, my_emp_name;
```

Because the emp_id_in parameter has a default value, I can also open and fetch from the cursor without specifying a value for the parameter. If I do not specify a value for the parameter, the cursor uses the default value.

SELECT...FOR UPDATE

When you issue a SELECT statement against the database to query some records, no locks are placed on the selected rows. In general, this is a wonderful feature because the number of records locked at any given time is kept to the absolute minimum: only those records that have been changed but not yet committed are locked. Even then, others are able to read those records as they appeared before the change (the "before image" of the data).

There are times, however, when you will want to lock a set of records even before you change them in your program. Oracle offers the FOR UPDATE clause of the SELECT statement to perform this locking.

When you issue a SELECT...FOR UPDATE statement, the database automatically obtains row-level locks on all the rows identified by the SELECT statement, holding the records "for your changes only" as you move through the rows retrieved by the cursor. It's as if you've issued an UPDATE statement against the rows, but you haven't—you've merely SELECTed them. No one else will be able to change any of these records until you perform a ROLLBACK or a COMMIT—but other sessions can still read the data.

Here are two examples of the FOR UPDATE clause used in a cursor:

```
CURSOR toys_cur IS
   SELECT name, manufacturer, preference_level, sell_at_yardsale_flag
     FROM my_sons_collection
    WHERE hours_used = 0
      FOR UPDATE;

CURSOR fall_jobs_cur IS
   SELECT task, expected_hours, tools_required, do_it_yourself_flag
     FROM winterize
    WHERE year_of_task = TO_CHAR (SYSDATE, 'YYYY')
      FOR UPDATE OF task;
```

The first cursor uses the unqualified FOR UPDATE clause, while the second cursor qualifies the FOR UPDATE with a column name from the query.

You can use the FOR UPDATE clause in a SELECT against multiple tables. In this case, rows in a table are locked only if the FOR UPDATE clause references a column in that table. In the following example, the FOR UPDATE clause does not result in any locked rows in the winterize table:

```
CURSOR fall_jobs_cur
IS
  SELECT w.task, w.expected_hours,
         w.tools_required,
         w.do_it_yourself_flag
    FROM winterize w, husband_config hc
   WHERE w.year_of_task = TO_CHAR (SYSDATE, 'YYYY')
     AND w.task_id = hc.task_id
  FOR UPDATE OF hc.max_procrastination_allowed;
```

The FOR UPDATE OF clause mentions only the max_procrastination_allowed column; no columns in the winterize table are listed. As a result, no rows in the winterize table will be locked. It is important to minimize the amount of data you lock so that you decrease the impact you have on other sessions. Other sessions may be blocked by your locks, waiting for you to complete your transaction so they can proceed with their own DML statements.

If you simply state FOR UPDATE in the query and do not include one or more columns after the OF keyword, the database will then lock all identified rows across all tables listed in the FROM clause.

Furthermore, you do not have to actually UPDATE or DELETE any records just because you issue a SELECT...FOR UPDATE statement—that act simply states your intention to be able to do so (and prevents others from doing the same).

Finally, you can append the optional keyword NOWAIT to the FOR UPDATE clause to tell the database not to wait if the table has been locked by another user. In this case, control will be returned immediately to your program so that you can perform other work, or simply wait for a period of time before trying again. You can also append WAIT to specify the maximum number of seconds the database should wait to obtain the lock. If no wait behavior is specified, then your session will be blocked until the table is available. For remote objects, the database initialization parameter, DISTRIBUTED_LOCK_TIMEOUT, is used to set the limit.

Releasing Locks with COMMIT

As soon as a cursor with a FOR UPDATE clause is OPENed, all rows identified in the result set of the cursor are locked and remain locked until your session ends or your code explicitly issues either a COMMIT or a ROLLBACK. When either of these occurs, the locks on the rows are released. As a result, you cannot execute another FETCH against a FOR UPDATE cursor after a COMMIT or ROLLBACK. You will have lost your position in the cursor.

Consider the following program, which assigns winterization chores:[1]

```
DECLARE
   /* All the jobs in the fall to prepare for the winter */
   CURSOR fall_jobs_cur
   IS
      SELECT task, expected_hours, tools_required, do_it_yourself_flag
        FROM winterize
       WHERE year = TO_NUMBER (TO_CHAR (SYSDATE, 'YYYY'))
         AND completed_flag = 'NOTYET' FOR UPDATE;
BEGIN
   /* For each job fetched by the cursor... */
   FOR job_rec IN fall_jobs_cur
   LOOP
      IF job_rec.do_it_yourself_flag = 'YOUCANDOIT'
      THEN
         /*
         || I have found my next job. Assign it to myself (like someone
         || else is going to do it!) and then commit the changes.
         */
         UPDATE winterize SET responsible = 'STEVEN'
          WHERE task = job_rec.task
            AND year = TO_NUMBER (TO_CHAR (SYSDATE, 'YYYY'));
         COMMIT;
      END IF;
   END LOOP;
END;
```

Suppose this loop finds its first YOUCANDOIT job. It then commits an assignment of a job to STEVEN. When it tries to FETCH the next record, the program raises the following exception:

```
ORA-01002: fetch out of sequence
```

If you ever need to execute a COMMIT or ROLLBACK as you FETCH records from a SELECT FOR UPDATE cursor, you should include code (such as a loop EXIT or other conditional logic) to halt any further fetches from the cursor.

1. Caveat: I don't want to set false expectations, especially with my wife. The code in this block is purely an example. In reality, I set the max_procrastination_allowed to five years and let my house decay until I can afford to pay someone else to do something, or my wife does it, or she gives me an ultimatum. Now you know why I decided to write books and software, rather than do things in the "real world."

The WHERE CURRENT OF Clause

PL/SQL provides the WHERE CURRENT OF clause for both UPDATE and DELETE statements inside a cursor. This clause allows you to easily make changes to the most recently fetched row of data.

To update columns in the most recently fetched row, specify:

```
UPDATE table_name
   SET set_clause
 WHERE CURRENT OF cursor_name;
```

To delete from the database the row for the most recently fetched record, specify:

```
DELETE
  FROM table_name
 WHERE CURRENT OF cursor_name;
```

Notice that the WHERE CURRENT OF clause references the cursor, not the record into which the next row fetched is deposited.

The most important advantage of using WHERE CURRENT OF to change the last row fetched is that you do not have to code in two (or more) places the criteria used to uniquely identify a row in a table. Without WHERE CURRENT OF, you would need to repeat the WHERE clause of your cursor in the WHERE clause of the associated UPDATEs and DELETEs. As a result, if the table structure changed in a way that affected the construction of the primary key, you would have to update each SQL statement to support this change. If you use WHERE CURRENT OF, on the other hand, you modify only the WHERE clause of the SELECT statement.

This might seem like a relatively minor issue, but it is one of many areas in your code where you can leverage subtle features in PL/SQL to minimize code redundancies. Utilization of WHERE CURRENT OF, %TYPE and %ROWTYPE declaration attributes, cursor FOR loops, local modularization, and other PL/SQL language constructs can significantly reduce the pain of maintaining your Oracle-based applications.

Let's see how this clause would improve the example in the previous section. In the jobs cursor FOR loop, I want to UPDATE the record that was most recently FETCHed by the cursor. I do this in the UPDATE statement by repeating the same WHERE used in the cursor, because "(task, year)" makes up the primary key of this table:

```
WHERE task = job_rec.task
  AND year = TO_CHAR (SYSDATE, 'YYYY');
```

This is a less than ideal situation, as previously explained: I have coded the same logic in two places, and this code must be kept synchronized. It would be so much more convenient and natural to be able to code the equivalent of the following statements:

- "Delete the row I just fetched."

- “Update these columns in the row I just fetched.”

A perfect fit for WHERE CURRENT OF! The next version of my winterization program uses this clause. I have also switched from a FOR loop to a simple loop because I want to exit conditionally from the loop (possible but not recommended with a FOR loop):

```
DECLARE
   CURSOR fall_jobs_cur IS SELECT ... same as before ... ;
   job_rec fall_jobs_cur%ROWTYPE;
BEGIN
   OPEN fall_jobs_cur;
   LOOP
      FETCH fall_jobs_cur INTO job_rec;

      EXIT WHEN fall_jobs_cur%NOTFOUND;

      IF job_rec.do_it_yourself_flag = 'YOUCANDOIT'
      THEN
         UPDATE winterize SET responsible = 'STEVEN'
          WHERE CURRENT OF fall_jobs_cur;
         COMMIT;
         EXIT;
      END IF;
   END LOOP;
   CLOSE fall_jobs_cur;
END;
```

Cursor Variables and REF CURSORs

A cursor variable is a variable that points to or references an underlying cursor. Unlike an explicit cursor, which names the PL/SQL work area for the result set, a cursor variable is a reference to that work area. Explicit and implicit cursors are static in that they are tied to specific queries. The cursor variable can be opened for any query, and even for different queries within a single program execution.

The most important benefit of the cursor variable is that it provides a mechanism for passing results of queries (the rows returned by fetches against a cursor) between different PL/SQL programs—even between client and server PL/SQL programs. Prior to PL/SQL Release 2.3, you would have had to fetch all data from the cursor, store it in PL/SQL variables (perhaps a collection), and then pass those variables as arguments. With cursor variables, you simply pass the reference to that cursor. This improves performance and streamlines your code.

It also means that the cursor is, in effect, shared among the programs that have access to the cursor variable. In a client-server environment, for example, a program on the client side could open and start fetching from the cursor variable, and then pass that variable as an argument to a stored procedure on the server. This stored program could then continue fetching and pass control back to the client program to close the cursor.

You can also perform the same steps between different stored programs on the same or different database instances.

This process, shown in Figure 15-2, offers dramatic new possibilities for data sharing and cursor management in PL/SQL programs.

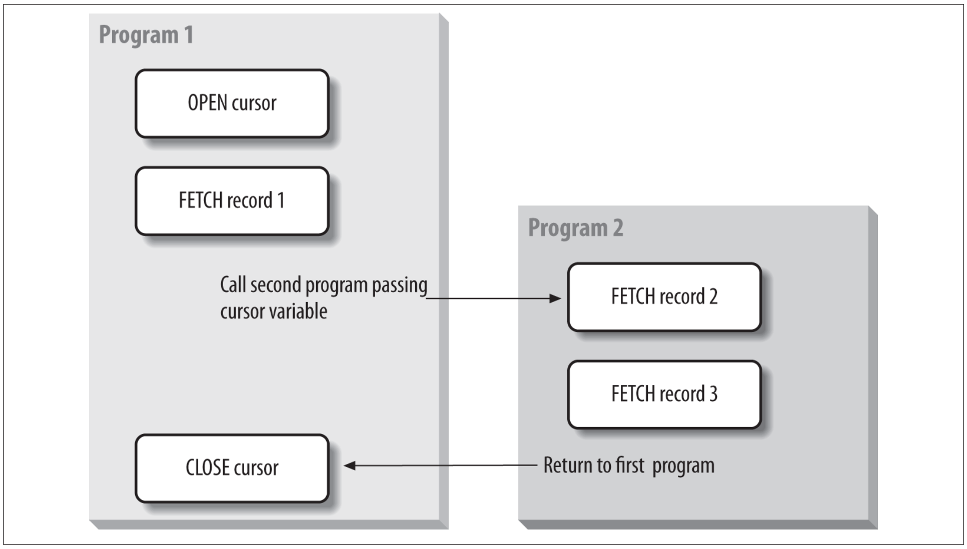


Figure 15-2. Referencing a cursor variable across two programs

Why Use Cursor Variables?

You can do all of the following with cursor variables:

- Associate them with different queries at different times in your program execution. In other words, a single cursor variable can be used to fetch from different result sets.
- Pass them as an argument to a procedure or function. You can, in essence, share the results of a cursor by passing a reference to that result set.
- Employ the full functionality of static PL/SQL cursors. You can OPEN, CLOSE, and FETCH with cursor variables within your PL/SQL programs. You can also reference the standard cursor attributes—%ISOPEN, %FOUND, %NOTFOUND, and %ROWCOUNT—for cursor variables.
- Assign the contents of one cursor (and its result set) to another cursor variable. Because the cursor variable is a variable, it can be used in assignment operations. There are restrictions on referencing this kind of variable, however, as I'll discuss later in this chapter.

Similarities to Static Cursors

One of the key design requirements for cursor variables was that, when possible, the semantics used to manage cursor objects would be the same as those of static cursors. While the declaration of a cursor variable and the syntax for opening it are enhanced, the following cursor operations for cursor variables are the same as for static cursors:

The CLOSE statement
: In the following example, I declare a REF CURSOR type and a cursor variable based on that type. Then I close the cursor variable using the same syntax as for a static cursor:

```
DECLARE
   TYPE var_cur_type IS REF CURSOR;
   var_cur var_cur_type;
BEGIN
   OPEN var_cur FOR ...
   ...
   CLOSE var_cur;
END;
```

Cursor attributes
: You can use any of the four cursor attributes with exactly the same syntax as for a static cursor. The rules governing the use of and values returned by those attributes match those of explicit cursors. If I have declared a cursor variable as in the previous example, I could use all the cursor attributes as follows:

```
var_cur%ISOPEN
var_cur%FOUND
var_cur%NOTFOUND
var_cur%ROWCOUNT
```

Fetching from the cursor variable
: You use the same FETCH syntax when fetching from a cursor variable into local PL/SQL data structures. There are, however, additional rules applied by PL/SQL to make sure that the data structures of the cursor variable's row (the set of values returned by the cursor object) match those of the data structures to the right of the INTO keyword. These rules are discussed in "Rules for Cursor Variables" on page 527.

Because the syntax for these aspects of cursor variables is the same as for the already familiar explicit cursors, the following sections will focus on features that are unique to cursor variables.

Declaring REF CURSOR Types

Just as with a collection or a programmer-defined record, you must perform two distinct declaration steps in order to create a cursor variable:

1. Create a referenced cursor type.

2. Declare the actual cursor variable based on that type.

The syntax for creating a referenced cursor type is as follows:

```
TYPE cursor_type_name IS REF CURSOR [ RETURN return_type ];
```

where *cursor_type_name* is the name of the type of cursor and *return_type* is the RETURN data specification for the cursor type. The *return_type* can be any of the data structures valid for a normal cursor RETURN clause, and you define it either using the %ROWTYPE attribute or by referencing a previously defined record type.

Notice that the RETURN clause is optional with the REF CURSOR type statement. Both of the following declarations are valid:

```
TYPE company_curtype IS REF CURSOR RETURN company%ROWTYPE;
TYPE generic_curtype IS REF CURSOR;
```

The first form of the REF CURSOR statement is called a *strong type* because it attaches a record type (or row type) to the cursor variable type at the moment of declaration. Any cursor variable declared using that type can only FETCH INTO data structures that match the specified record type. The advantage of a strong type is that the compiler can determine whether or not the developer has properly matched up the cursor variable's FETCH statements with its cursor object's query list.

The second form of the REF CURSOR statement, in which the RETURN clause is missing, is called a *weak type*. This cursor variable type is not associated with any record data structures. Cursor variables declared without the RETURN clause can be used in more flexible ways than the strong type. They can be used with any query or with any record type structure, and can vary even within the course of a single program.

Starting with Oracle9*i* Database, Oracle provides a predefined weak REF CURSOR type named SYS_REFCURSOR. You no longer need to define your own weak type; just use Oracle's:

```
DECLARE
   my_cursor SYS_REFCURSOR;
```

Declaring Cursor Variables

The syntax for declaring a cursor variable is:

```
cursor_name cursor_type_name;
```

where *cursor_name* is the name of the cursor, and *cursor_type_name* is the name of the type of cursor previously defined with a TYPE statement.

Here is an example of the creation of a cursor variable:

```
DECLARE
   /* Create a cursor type for sports cars. */
   TYPE sports_car_cur_type IS REF CURSOR RETURN car%ROWTYPE;
   /* Create a cursor variable for sports cars. */
   sports_car_cur sports_car_cur_type;
BEGIN
   ...
END;
```

It is important to distinguish between declaring a cursor variable and creating an actual cursor object—the result set identified by the cursor SQL statement. A constant is nothing more than a value, whereas a variable points to its value. Similarly, a static cursor acts as a constant, whereas a cursor variable references or points to a cursor object. These distinctions are shown in Figure 15-3. Notice that two different cursor variables in different programs are both referring to the same cursor object.

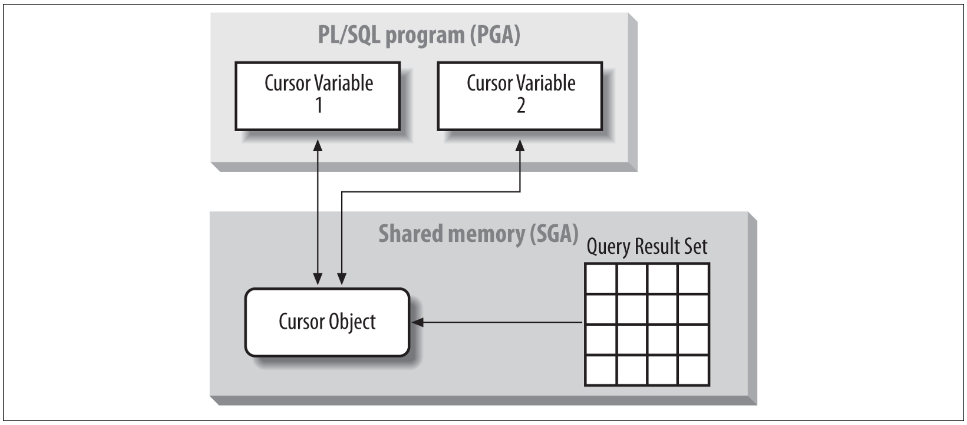


Figure 15-3. The referencing character of cursor variables

Declaring a cursor variable does not create a cursor object. You must use the OPEN FOR syntax to create a new cursor object and assign it to the variable.

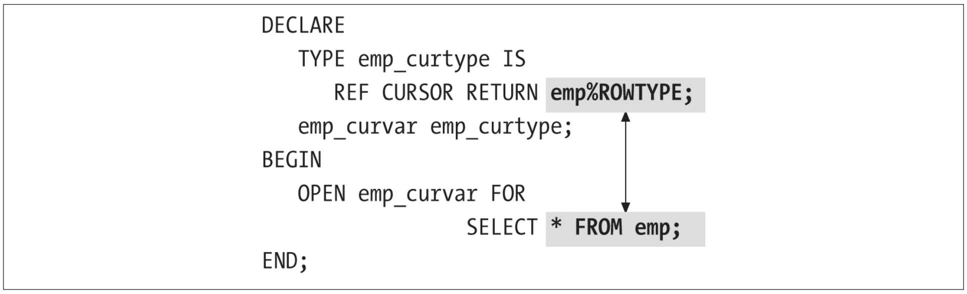
Opening Cursor Variables

You assign a value (the cursor object) to a cursor variable when you OPEN the cursor. So, the syntax for the traditional OPEN statement allows for cursor variables to accept a SELECT statement after the FOR clause, as follows:

```
OPEN cursor_name FOR select_statement;
```

where *cursor_name* is the name of a cursor variable, and *select_statement* is a SQL SELECT statement.

For strong REF CURSOR type cursor variables, the structure of the SELECT statement (the number and datatypes of the columns) must match or be compatible with the structure specified in the RETURN clause of the TYPE statement. Figure 15-4 shows an example of the kind of compatibility required. "Rules for Cursor Variables" on page 527 contains the full set of compatibility rules.



```
DECLARE
   TYPE emp_curtype IS
      REF CURSOR RETURN emp%ROWTYPE;
   emp_curvar emp_curtype;
BEGIN
   OPEN emp_curvar FOR
                 SELECT * FROM emp;
END;
```

Figure 15-4. Compatible REF CURSOR rowtype and select list

If *cursor_name* is a cursor variable defined with a weak REF CURSOR type, you can OPEN it for any query, with any structure. In the following example, I open (assign a value to) the cursor variable three times, with three different queries:

```
DECLARE
   TYPE emp_curtype IS REF CURSOR;
   emp_curvar emp_curtype;
BEGIN
   OPEN emp_curvar FOR SELECT * FROM employees;
   OPEN emp_curvar FOR SELECT employee_id FROM employees;
   OPEN emp_curvar FOR SELECT company_id, name FROM company;
END;
```

That last OPEN didn't even have anything to do with the employees table!

If the cursor variable has not yet been assigned to any cursor object, the OPEN FOR statement implicitly creates an object for the variable. If at the time of the OPEN the cursor variable is already pointing to a cursor object, OPEN FOR does not create a new object. Instead, it reuses the existing object and attaches a new query to that object. The cursor object is maintained separately from the cursor or query itself.

If you associate a new result set with a cursor variable that was previously used in an OPEN FOR statement, and you did not explicitly close that cursor variable, then the underlying cursor remains *open*. You should always explicitly close your cursor variables before repurposing them with another result set.

Fetching from Cursor Variables

As mentioned earlier, the syntax for a FETCH statement using a cursor variable is the same as that for static cursors:

```
FETCH cursor_variable_name INTO record_name;
FETCH cursor_variable_name INTO variable_name, variable_name ...;
```

When the cursor variable is declared with a strong REF CURSOR type, the PL/SQL compiler makes sure that the data structures listed after the INTO keyword are compatible with the structure of the query associated with the cursor variable.

If the cursor variable is of a weak REF CURSOR type, the PL/SQL compiler cannot perform the same kind of check it performs for a strong REF CURSOR type. Such a cursor variable can FETCH into any data structures because the REF CURSOR type is not identified with a rowtype at the time of declaration. At compile time, there is no way to know which cursor object (and associated SQL statement) will be assigned to that variable.

Consequently, the check for compatibility must happen at runtime, when the FETCH is about to be executed. At this point, if the query and the INTO clause do not structurally match, the PL/SQL runtime engine will raise the predefined ROWTYPE_MISMATCH exception. Note that PL/SQL will use implicit conversions if necessary and possible.

Handling the ROWTYPE_MISMATCH exception

You can trap the ROWTYPE_MISMATCH exception and then attempt to FETCH from the cursor variable using a different INTO clause. But even though you are executing the second FETCH statement in your program, you will still retrieve the first row in the result set of the cursor object's query. This functionality comes in handy for weak REF CURSOR types, which you can easily define using the predefined SYS_REFCURSOR type.

In the following example, a centralized real estate database stores information about properties in a variety of tables: one for homes, another for commercial properties, and so on. There is also a single, central table that stores addresses and building types (home, commercial, etc.). I use a single procedure to open a weak REF CURSOR variable for the appropriate table based on the street address. Each individual real estate office can then call that procedure to scan through the matching properties. Here are the steps:

1. Create the procedure. Notice that the mode of the cursor variable parameter is IN OUT:

```
/* File on web: rowtype_mismatch.sql */
PROCEDURE open_site_list
   (address_in IN VARCHAR2,
    site_cur_inout IN OUT SYS_REFCURSOR)
IS
   home_type CONSTANT PLS_INTEGER := 1;
   commercial_type CONSTANT PLS_INTEGER := 2;

   /* A static cursor to get building type. */
   CURSOR site_type_cur IS
```

```
         SELECT site_type FROM property_master
          WHERE address = address_in;
      site_type_rec site_type_cur%ROWTYPE;

   BEGIN
      /* Get the building type for this address. */
      OPEN site_type_cur;
      FETCH site_type_cur INTO site_type_rec;
      CLOSE site_type_cur;

      /* Now use the site type to select from the right table.*/
      IF site_type_rec.site_type =  home_type
      THEN
         /* Use the home properties table. */
         OPEN site_cur_inout FOR
            SELECT * FROM home_properties
             WHERE address LIKE '%' || address_in || '%';

      ELSIF site_type_rec.site_type =  commercial_type
      THEN
         /* Use the commercial properties table. */
         OPEN site_cur_inout FOR
            SELECT * FROM commercial_properties
             WHERE address LIKE '%' || address_in || '%';
      END IF;
   END open_site_list;
```

2. Now that I have my open procedure, I can use it to scan properties.

   In the following example, I pass in the address and then try to fetch from the cursor, assuming a home property. If the address actually identifies a commercial property, PL/SQL will raise the ROWTYPE_MISMATCH exception on account of the incompatible record structures. The exception section then fetches again, this time into a commercial building record, and the scan is complete:

```
   /* File on web: rowtype_mismatch.sql */
   DECLARE
      /* Declare a cursor variable. */
      building_curvar   sys_refcursor;

      address_string    property_master.address%TYPE;

      /* Define record structures for two different tables. */
      home_rec          home_properties%ROWTYPE;
      commercial_rec    commercial_properties%ROWTYPE;
   BEGIN
      /* Retrieve the address from a cookie or other source. */
      address_string := current_address ();

      /* Assign a query to the cursor variable based on the address. */
      open_site_list (address_string, building_curvar);

      /* Give it a try! Fetch a row into the home record. */
      FETCH building_curvar
      INTO home_rec;

      /* If I got here, the site was a home, so display it. */
      show_home_site (home_rec);
   EXCEPTION
      /* If the first record was not a home... */
      WHEN ROWTYPE_MISMATCH
      THEN
         /* Fetch that same 1st row into the commercial record. */
         FETCH building_curvar
         INTO commercial_rec;

         /* Show the commercial site info. */
         show_commercial_site (commercial_rec);
   END;
```

Rules for Cursor Variables

This section examines in more detail the rules and issues regarding the use of cursor variables in your programs. These include rowtype matching rules, cursor variable aliases, and scoping issues.

Remember that the cursor variable is a reference to a cursor object or query in the database. It is not the object itself. A cursor variable is said to refer to a given query if either of the following is true:

- An OPEN statement FOR that query was executed with the cursor variable.
- A cursor variable was assigned a value from another cursor variable that refers to that query.

You can perform assignment operations with cursor variables and also pass these variables as arguments to procedures and functions. To make it possible to perform such actions between cursor variables (and to bind a cursor variable to a parameter), the different cursor variables must follow a set of compile-time and runtime rowtype matching rules.

Compile-time rowtype matching rules

These are the rules that PL/SQL follows at compile time:

- Two cursor variables (including procedure parameters) are compatible for assignments and argument passing if any of the following are true:
  — Both variables (or parameters) are of a strong REF CURSOR type with the same *rowtype_name*.

    — Both variables (or parameters) are of a weak REF CURSOR type, regardless of the *rowtype_name*.
    — One variable (or parameter) is of any strong REF CURSOR type, and the other is of any weak REF CURSOR type.
- A cursor variable (or parameter) of a strong REF CURSOR type may be OPEN FOR a query that returns a rowtype that is structurally equal to the *rowtype_name* in the original type declaration.
- A cursor variable (or parameter) of a weak REF CURSOR type may be OPEN FOR any query. The FETCH from such a variable is allowed INTO any list of variables or record structure.

If either of the cursor variables is of a weak REF CURSOR type, then the PL/SQL compiler cannot really validate whether the two different cursor variables will be compatible. That will happen at runtime; the rules are covered in the next section.

Runtime rowtype matching rules

These are the rules that PL/SQL follows at runtime:

- A cursor variable (or parameter) of a weak REF CURSOR type may be made to refer to a query of any rowtype, regardless of the query or cursor object to which it may have referred earlier.
- A cursor variable (or parameter) of a strong REF CURSOR type may be made to refer only to a query that matches structurally the *rowtype_name* of the RETURN clause of the REF CURSOR type declaration.
- Two records (or lists of variables) are considered structurally matching with implicit conversions if both of the following are true:
    — The number of fields is the same in both records (or lists).
    — For each field in one record (or variable in one list), a corresponding field in the second list (or a variable in the second list) has the same PL/SQL datatype, or one that can be converted implicitly by PL/SQL to match the first.
- For a cursor variable (or parameter) used in a FETCH statement, the query associated with the cursor variable must structurally match (with implicit conversions) the record or list of variables of the INTO clause of the FETCH statement. This same rule is used for static cursors.

Cursor variable aliases

If you assign one cursor variable to another cursor variable, they become *aliases* for the same cursor object; i.e., they share the reference to the cursor object (the result set of the cursor's query). Any action taken against the cursor object through one variable is also available to and reflected in the other variable.

This anonymous block illustrates the way cursor aliases work:

```
1   DECLARE
2      TYPE curvar_type IS REF CURSOR;
3      curvar1 curvar_type;
4      curvar2 curvar_type;
5      story fairy_tales%ROWTYPE;
6   BEGIN
7      OPEN curvar1 FOR SELECT * FROM fairy_tales;
8      curvar2 := curvar1;
9      FETCH curvar1 INTO story;
10     FETCH curvar2 INTO story;
11     CLOSE curvar2;
12     FETCH curvar1 INTO story;
13  END;
```

The following table provides an explanation of the cursor variable actions.

Line(s)	Description
2–5	Declare my weak REF CURSOR type and cursor variables.
7	Create a cursor object and assigns it to curvar1 by opening a cursor for that cursor variable.
8	Assign that same cursor object to the second cursor variable, curvar2. (Now I have two cursor variables that can be used to manipulate the same result set!)
9	Fetch the first record using the curvar1 variable.
10	Fetch the second record using the curvar2 variable. (Notice that it doesn't matter which of the two variables you use. The pointer to the current record resides with the cursor object, not with any particular variable.)
11	Close the cursor object referencing curvar2.
12	Will raise the INVALID_CURSOR exception when I try to fetch again from the cursor object. (When I closed the cursor through curvar2, it also closed it as far as curvar1 was concerned.)

Any change of state in a cursor object will be seen through any cursor variable that is an alias for that cursor object.

Scope of cursor object

The scope of a cursor variable is the same as that of a static cursor: the PL/SQL block in which the variable is declared. The scope of the cursor object to which a cursor variable is assigned, however, is a different matter.

Once an OPEN FOR creates a cursor object, that cursor object remains accessible as long as at least one active cursor variable refers to it. This means that you can create a cursor object in one scope (PL/SQL block) and assign it to a cursor variable. Then, by assigning that cursor variable to another cursor variable with a different scope, you can ensure that the cursor object remains accessible even if the original cursor variable has gone out of scope.

In the following example, I use nested blocks to demonstrate how a cursor object can persist outside of the scope in which it was originally created:

```
DECLARE
   curvar1 SYS_REFCURSOR;
      do_you_get_it VARCHAR2(100);
BEGIN
   /*
   || Nested block which creates the cursor object and
   || assigns it to the curvar1 cursor variable.
   */
   DECLARE
      curvar2 SYS_REFCURSOR;
   BEGIN
      OPEN curvar2 FOR SELECT punch_line FROM joke;
      curvar1 := curvar2;
   END;
   /*
   || The curvar2 cursor variable is no longer active,
   || but "the baton" has been passed to curvar1, which
   || does exist in the enclosing block. I can therefore
   || fetch from the cursor object, through this other
   || cursor variable.
   */
   FETCH curvar1 INTO do_you_get_it;
   CLOSE curvar1;
END;
```

Passing Cursor Variables as Arguments

You can pass a cursor variable as an argument in a call to a procedure or a function. When you use a cursor variable in the parameter list of a program, you need to specify the mode of the parameter and the datatype (the REF CURSOR type).

Identifying the REF CURSOR type

In your program header, you must identify the REF CURSOR type of your cursor variable parameter. To do this, that cursor type must already be defined.

If you are creating a local module within another program (see Chapter 17 for information about local modules), you can define the cursor type in the same program. It will then be available for the parameter. This approach is shown here:

```
DECLARE
   /* Define the REF CURSOR type. */
   TYPE curvar_type IS REF CURSOR RETURN company%ROWTYPE;

   /* Reference it in the parameter list. */
   PROCEDURE open_query (curvar_out OUT curvar_type)
   IS
      local_cur curvar_type;
   BEGIN
      OPEN local_cur FOR SELECT * FROM company;
      curvar_out := local_cur;
   END;
BEGIN
   ...
END;
```

If you are creating a standalone procedure or function, then the only way you can reference a preexisting REF CURSOR type is by placing that TYPE statement in a package. All variables declared in the specification of a package act as globals within your session, so you can then reference this cursor type using the dot notation shown in the second example:

- Create the package with a REF CURSOR type declaration:

  ```
  PACKAGE company
  IS
     /* Define the REF CURSOR type. */
     TYPE curvar_type IS REF CURSOR RETURN company%ROWTYPE;
  END package;
  ```

- In a standalone procedure, reference the REF CURSOR type by prefacing the name of the cursor type with the name of the package:

  ```
  PROCEDURE open_company (curvar_out OUT company.curvar_type) IS
  BEGIN
     ...
  END;
  ```

Setting the parameter mode

Just like other parameters, a cursor variable argument can have one of the following three modes:

IN
: Can only be read by the program

OUT
: Can only be written to by the program

IN OUT
: Can be read or written to by the program

Remember that the value of a cursor variable is the reference to the cursor object, not the state of the cursor object. In other words, the value of a cursor variable does not change after you fetch from or close a cursor.

Only two operations, in fact, may change the value of a cursor variable (that is, the cursor object to which the variable points):

- An assignment to the cursor variable
- An OPEN FOR statement

If the cursor variable is already pointing to a cursor object, the OPEN FOR doesn't actually change the reference; it simply changes the query associated with the object.

The FETCH and CLOSE operations affect the state of the cursor object, but not the reference to the cursor object itself, which is the value of the cursor variable.

Here is an example of a program that has cursor variables as parameters:

```
PROCEDURE assign_curvar
   (old_curvar_in IN company.curvar_type,
    new_curvar_out OUT company.curvar_type)
IS
BEGIN
   new_curvar_out := old_curvar_in;
END;
```

This procedure copies the old company cursor variable to the new variable. The first parameter is an IN parameter because it appears only on the right side of the assignment. The second parameter must be an OUT (or IN OUT) parameter because its value is changed inside the procedure. Notice that the curvar_type is defined within the company package.

Cursor Variable Restrictions

Cursor variables are subject to the following restrictions (note that Oracle may remove some of these in future releases):

- Cursor variables cannot be declared in a package because they do not have a persistent state.
- You cannot use remote procedure calls (RPCs) to pass cursor variables from one server to another.
- If you pass a cursor variable as a bind variable or host variable to PL/SQL, you will not be able to fetch from it from within the server unless you also open it in that same server call.
- The query you associate with a cursor variable in an OPEN FOR statement cannot use the FOR UPDATE clause if you are running Oracle8*i* Database or earlier.
- You cannot test for cursor variable equality, inequality, or nullity using comparison operators.
- You cannot assign NULL to a cursor variable. Attempts to do so will result in a *PLS-00382: Expression is of wrong type* error message.

- Database columns cannot store cursor variable values. You will not be able to use REF CURSOR types to specify column types in CREATE TABLE statements.
- The elements in a nested table, associative array, or VARRAY cannot store the values of cursor variables. You will not be able to use REF CURSOR types to specify the element type of a collection.

Cursor Expressions

Oracle provides a powerful feature in the SQL language: the *cursor expression*. A cursor expression, denoted by the CURSOR operator, returns a nested cursor from within a query. Each row in the result set of this nested cursor can contain the usual range of values allowed in a SQL query; it can also contain other cursors as produced by sub-queries.

> The CURSOR syntax, although first introduced in Oracle8*i* Database SQL, was not available from within PL/SQL programs. This deficiency was corrected in Oracle9*i* Database Release 1; since then, SQL statements within a PL/SQL procedure or function have been able to take advantage of the CURSOR expression.

You can therefore use cursor expressions to return a large and complex set of related values retrieved from one or more tables. You can then process the cursor expression result set using nested loops that fetch from the rows of the result set, and then additional rows from any nested cursors within those rows.

Cursor expressions can get complicated, given how complex the queries and result sets can be. Nevertheless, it's good to know all the possible ways to retrieve data from the Oracle database.

You can use cursor expressions in any of the following:

- Explicit cursor declarations
- Dynamic SQL queries
- REF CURSOR declarations and variables

You cannot use a cursor expression in an implicit query.

The syntax for a cursor expression is very simple:

```
CURSOR (subquery)
```

The database opens the nested cursor defined by a cursor expression implicitly as soon as it fetches the row containing the cursor expression from the parent or outer cursor. This nested cursor is closed when:

- You explicitly close the cursor.
- The outer (parent) cursor is executed again, closed, or canceled.
- An exception is raised while fetching from the parent cursor. The nested cursor is closed along with the parent cursor.

Using Cursor Expressions

You can use a CURSOR expression in two different but very useful ways:

- To retrieve a subquery as a column in an outer query
- To transform a query into a result set that can be passed as an argument to a streaming or transformative function

Retrieving a subquery as a column

The following procedure demonstrates the use of nested CURSOR expressions to retrieve a subquery as a column in an outer query. The top-level query fetches just two pieces of data: the city location and a nested cursor containing departments in that city. This nested cursor, in turn, fetches a nested cursor with a CURSOR expression—in this case, one containing the names of all the employees in each department.

I could have performed this same retrieval with separate explicit cursors, opened and processed in a nested fashion. The CURSOR expression gives us the option of using a different approach, and one that can be much more concise and efficient, given that all the processing takes place in the SQL statement executor (which reduces context switching):

```
PROCEDURE emp_report (p_locid NUMBER)
IS
   TYPE refcursor IS REF CURSOR;

   -- The query returns only 2 columns, but the second column is
   -- a cursor that lets us traverse a set of related information.
   CURSOR all_in_one_cur is
      SELECT l.city,
             CURSOR (SELECT d.department_name,
                            CURSOR(SELECT e.last_name
                                     FROM employees e
                                    WHERE e.department_id =
                                                d.department_id)
                                  AS ename
                       FROM departments d
                      WHERE l.location_id = d.location_id) AS dname
        FROM locations l
       WHERE l.location_id = p_locid;
   departments_cur    refcursor;
   employees_cur   refcursor;

   v_city    locations.city%TYPE;
   v_dname   departments.department_name%TYPE;
   v_ename   employees.last_name%TYPE;
BEGIN
   OPEN all_in_one_cur;

   LOOP
      FETCH all_in_one_cur INTO v_city, departments_cur;
      EXIT WHEN all_in_one_cur%NOTFOUND;

      -- Now I can loop through departments and I do NOT need to
      -- explicitly open that cursor. Oracle did it for me.
      LOOP
         FETCH departments_cur INTO v_dname, employees_cur;
         EXIT WHEN departments_cur%NOTFOUND;

         -- Now I can loop through employees for that department.
         -- Again, I do not need to open the cursor explicitly.
         LOOP
            FETCH employees_cur INTO v_ename;
            EXIT WHEN employees_cur%NOTFOUND;
            DBMS_OUTPUT.put_line (
                  v_city
               || ' '
               || v_dname
               || ' '
               || v_ename
            );
         END LOOP;
      END LOOP;
   END LOOP;

   CLOSE all_in_one_cur;
END;
```

Implementing a streaming function with the CURSOR expression

Streaming functions, also known as *transformative functions*, allow you to transform data from one state to another without using any local data structures as intermediate staging points. Suppose, for example, that I need to take the data in StockTable and move it into TickerTable, pivoting one row in StockTable to two rows in TickerTable. Using the CURSOR expression and table functions, I can implement this solution as follows:

```
INSERT INTO TickerTable
   SELECT *
     FROM TABLE (StockPivot (CURSOR (SELECT * FROM StockTable)));
```

where the StockPivot function contains all the complex logic needed to perform the transformation. This technique is explained in depth in Chapter 17.

Restrictions on Cursor Expressions

There are a number of restrictions on the use of cursor expressions:

- You cannot use a cursor expression with an implicit cursor, because no mechanism is available to fetch the nested cursor INTO a PL/SQL data structure.
- Cursor expressions can appear only in the outermost select list of the query specification.
- You can place cursor expressions only in a SELECT statement that is not nested in any other query expression, except when it is defined as a subquery of the cursor expression itself.
- Cursor expressions cannot be used when declaring a view.
- You cannot perform BIND and EXECUTE operations on cursor expressions when using the CURSOR expression in dynamic SQL (see Chapter 16).