CHAPTER 5

Iterative Processing with Loops

This chapter explores the iterative control structures of PL/SQL, otherwise known as *loops*, which let you execute the same code repeatedly. It also describes the CONTINUE statement, introduced for loops in Oracle Database 11*g*. PL/SQL provides three different kinds of loop constructs:

- The simple or infinite loop
- The FOR loop (numeric and cursor)
- The WHILE loop

Each type of loop is designed for a specific purpose with its own nuances, rules for use, and guidelines for high-quality construction. As I explain each loop, I'll provide a table describing the following properties of the loop.

Property	Description
How the loop is terminated	A loop executes code repetitively. How do you make the loop stop executing its body?
When the test for termination takes place	Does the test for termination take place at the beginning or end of the loop? What are the consequences?
Reason to use this loop	What are the special factors you should consider to determine if this loop is right for your situation?

Loop Basics

Why are there three different kinds of loops? To provide you with the flexibility you need to write the most straightforward code to handle any particular situation. Most situations that require a loop could be written with any of the three loop constructs. If you do not pick the construct that is best suited for that particular requirement, however, you could end up having to write many additional lines of code. The resulting module will also be harder to understand and maintain.

Examples of Different Loops

To give you a feel for the way the different loops solve their problems in different ways, consider the following three procedures. In each case, the procedure makes a call to display_total_sales for a particular year, for each year number between the start and end argument values.

The simple loop

It's called simple for a reason: it starts simply with the LOOP keyword and ends with the END LOOP statement. The loop will terminate if you execute an EXIT, EXIT WHEN, or RETURN within the body of the loop (or if an exception is raised):

```
/* File on web: loop_examples.sql */
PROCEDURE display_multiple_years (
   start_year_in IN PLS_INTEGER
  ,end_year_in IN PLS_INTEGER
)
IS
   l_current_year PLS_INTEGER := start_year_in;
BEGIN
   LOOP
      EXIT WHEN l_current_year > end_year_in;
      display_total_sales (l_current_year);
      l_current_year :=  l_current_year + 1;
   END LOOP;
END display_multiple_years;
```

The FOR loop

Oracle offers a numeric and a cursor FOR loop. With the numeric FOR loop, you specify the start and end integer value and PL/SQL does the rest of the work for you, iterating through each intermediate value and then terminating the loop:

```
/* File on web: loop_examples.sql */
PROCEDURE display_multiple_years (
    start_year_in IN PLS_INTEGER
   ,end_year_in IN PLS_INTEGER
 )
 IS
 BEGIN
    FOR l_current_year IN start_year_in .. end_year_in
    LOOP
       display_total_sales (l_current_year);
    END LOOP;
 END display_multiple_years;
```

The cursor FOR loop has the same basic structure, but in this case you supply an explicit cursor or SELECT statement in place of the low-high integer range:

```
/* File on web: loop_examples.sql */
PROCEDURE display_multiple_years (
    start_year_in IN PLS_INTEGER
   ,end_year_in IN PLS_INTEGER
   )
   IS
   BEGIN
     FOR sales_rec IN (
        SELECT *
          FROM sales_data
         WHERE year BETWEEN start_year_in AND end_year_in)
     LOOP
        display_total_sales (sales_rec.year);
     END LOOP;
  END display_multiple_years;
```

The WHILE loop

The WHILE loop is very similar to the simple loop; a critical difference is that it checks the termination condition up front. It may not even execute its body a single time:

```
/* File on web: loop_examples.sql */
PROCEDURE display_multiple_years (
   start_year_in IN PLS_INTEGER
  ,end_year_in IN PLS_INTEGER
)
IS
   l_current_year PLS_INTEGER := start_year_in;
BEGIN
   WHILE (l_current_year <= end_year_in)
   LOOP
      display_total_sales (l_current_year);
      l_current_year :=  l_current_year + 1;
   END LOOP;
END display_multiple_years;
```

In this section, the FOR loop clearly requires the smallest amount of code. Yet I could use it in this case only because I knew that I would run the body of the loop a specific number of times. In many other situations, the number of times a loop must execute varies, so the FOR loop cannot be used.

Structure of PL/SQL Loops

While there are differences among the three loop constructs, every loop has two parts:

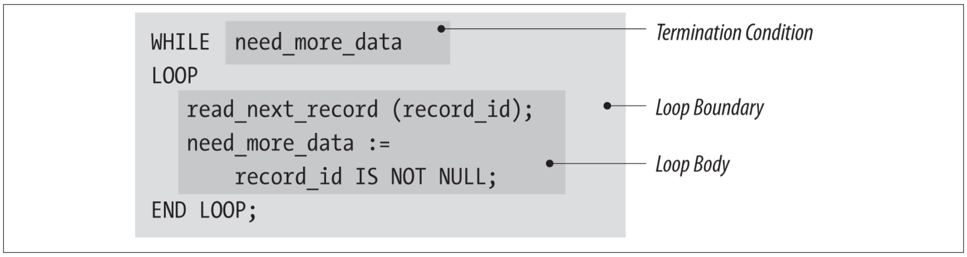
Loop boundary

This is composed of the reserved words that initiate the loop, the condition that causes the loop to terminate, and the END LOOP statement that ends the loop.

Loop body

This is the sequence of executable statements inside the loop boundary that execute on each iteration of the loop.

Figure 5-1 shows the boundary and body of a WHILE loop.



```
WHILE  need_more_data
LOOP
   read_next_record (record_id);
   need_more_data :=
       record_id IS NOT NULL;
END LOOP;
```

Figure 5-1. The boundary and body of the WHILE loop

In general, think of a loop much as you would a procedure or a function. The body of the loop is a black box, and the condition that causes loop termination is the interface to that black box. Code outside the loop should not have to know about the inner workings of the loop. Keep this in mind as you go through the different kinds of loops and examples in the rest of the chapter.

The Simple Loop

The structure of the simple loop is the most basic of all the loop constructs. It consists of the LOOP keyword, the body of executable code, and the END LOOP keywords, as shown here:

```
LOOP
  executable statement(s)
END LOOP;
```

The loop boundary consists solely of the LOOP and END LOOP reserved words. The body must consist of at least one executable statement. The following table summarizes the properties of the simple loop.

Property	Description
How the loop is terminated	The simple loop is terminated when an EXIT statement is executed in the body of the loop. If this statement is not executed, the simple loop becomes a true infinite loop.
When the test for termination takes place	The test takes place inside the body of the loop, and then only if an EXIT or EXIT WHEN statement is executed. Therefore, the body—or part of the body—of the simple loop always executes at least once.
Reason to use this loop	Use the simple loop when: • You are not sure how many times you want the loop to execute. • You want the loop to run at least once.

This loop is useful when you want to guarantee that the body (or at least part of the body) will execute at least one time. Because there is no condition associated with the loop boundary that determines whether or not it should execute, the body of the loop will always execute the first time.

The simple loop will terminate only when an EXIT (or its close cousin, EXIT WHEN) statement is executed in its body, or when an exception is raised (and goes unhandled) within the body of the loop.

Terminating a Simple Loop: EXIT and EXIT WHEN

Unless you want your loop to run forever, you can put an EXIT or EXIT WHEN statement within the body of the loop. The syntax for these statements is as follows:

```
EXIT;
EXIT WHEN condition;
```

where *condition* is a Boolean expression.

The following example demonstrates how the EXIT forces the loop to immediately halt execution and pass control to the next statement after the END LOOP statement. The account_balance procedure returns the amount of money remaining in the account specified by the account ID. If there is less than $1,000 left, the EXIT statement is executed and the loop is terminated. Otherwise, the program applies the balance to the outstanding orders for that account:

```
LOOP
   balance_remaining := account_balance (account_id);
   IF balance_remaining < 1000
   THEN
      EXIT;
   ELSE
      apply_balance (account_id, balance_remaining);
   END IF;
END LOOP;
```

You can use an EXIT statement only within a LOOP.

PL/SQL also offers the EXIT WHEN statement, which supports conditional termination of the loop. Essentially, the EXIT WHEN combines an IF-THEN statement with the EXIT statement. Using the same example, the EXIT WHEN changes the loop to:

```
LOOP
   /* Calculate the balance */
   balance_remaining := account_balance (account_id);

   /* Embed the IF logic into the EXIT statement */
   EXIT WHEN balance_remaining < 1000;

   /* Apply balance if still executing the loop */
   apply_balance (account_id, balance_remaining);
END LOOP;
```

Notice that the second form doesn't require an IF statement to determine when it should exit. Instead, that conditional logic is embedded inside the EXIT WHEN statement.

So when should you use EXIT WHEN, and when is the stripped-down EXIT more appropriate?

- EXIT WHEN is best used when there is a single conditional expression that determines whether or not a loop should terminate. The previous example demonstrates this scenario clearly.
- In situations with multiple conditions for exiting or when you need to set a "return value" coming out of the loop based on different conditions, you are probably better off using an IF or CASE statement, with EXIT statements in one or more of the clauses.

The following example demonstrates a preferred use of EXIT. It is taken from a function that determines if two files are equal (i.e., contain the same content):

```
   ...
   IF (end_of_file1 AND end_of_file2)
   THEN
      retval := TRUE;
      EXIT;
   ELSIF (checkline != againstline)
   THEN
      retval := FALSE;
      EXIT;
   ELSIF (end_of_file1 OR end_of_file2)
   THEN
      retval := FALSE;
      EXIT;
   END IF;
END LOOP;
```

Emulating a REPEAT UNTIL Loop

PL/SQL does not provide a REPEAT UNTIL loop in which the condition is tested after the body of the loop is executed and thus guarantees that the loop always executes at least once. You can, however, emulate a REPEAT UNTIL with a simple loop, as follows:

```
LOOP
   ... body of loop ...
   EXIT WHEN boolean_condition;
END LOOP;
```

where *boolean_condition* is a Boolean variable or an expression that evaluates to a Boolean value of TRUE or FALSE (or NULL).

The Intentionally Infinite Loop

Some programs, such as system monitoring tools, are not designed to be executed on demand but should always be running. In such cases, you may actually *want* to use an infinite loop:

```
LOOP
   data_gathering_procedure;
END LOOP;
```

Here, *data_gathering_procedure* goes out and, as you'd guess, gathers data about the system. As anyone who has accidentally run such an infinite loop can attest, it's likely that the loop will consume large portions of the CPU. The solution for this, in addition to ensuring that your data gathering is performed as efficiently as possible, is to pause between iterations:

```
LOOP
   data_gathering_procedure;
        DBMS_LOCK.sleep(10); -- do nothing for 10 seconds
END LOOP;
```

During the sleep period, the program uses virtually no cycles.

Terminating an Intentionally Infinite Loop

As a practical matter, there will be times when you really *do* want to terminate intentionally infinite loops. If you're just working on an anonymous block in SQL*Plus, typing the terminal interrupt sequence (usually Ctrl-C) will probably do the job. But real programs generally run as stored procedures, and even killing the process that submitted the program (such as SQL*Plus) won't stop the background task. Aha, you say, what about ALTER SYSTEM KILL SESSION? Nice idea, but in some versions of the Oracle database this command doesn't actually kill sessions that are stuck in a loop (go figure).

So how can you put an executing program to sleep—permanently?

You may have to resort to operating system–level tools such as *kill* in Unix/Linux and *orakill.exe* in Microsoft Windows. These commands require you to discover the system process ID of the Oracle "shadow task," which is not hard if you have privileges to read the V$SESSION and V$PROCESS views. But even if the inelegance isn't an issue for you, your conscience could bother you for another reason: if you're running in shared server mode, you will probably end up killing other sessions as well. The best solution that I've come up with is to insert into the loop a kind of "command interpreter" that uses the database's built-in interprocess communication, known as a *database pipe*:

```
DECLARE
   pipename CONSTANT VARCHAR2(12) := 'signaler';
   result INTEGER;
   pipebuf VARCHAR2(64);
BEGIN
```

```
      /* create private pipe with a known name */
      result := DBMS_PIPE.create_pipe(pipename);

      LOOP
         data_gathering_procedure;
         DBMS_LOCK.sleep(10);

         /* see if there is a message on the pipe */
         IF DBMS_PIPE.receive_message(pipename, 0) = 0
         THEN
            /* interpret the message and act accordingly */
            DBMS_PIPE.unpack_message(pipebuf);
            EXIT WHEN pipebuf = 'stop';
         END IF;
      END LOOP;
   END;
```

The DBMS_PIPE calls should have little impact on the overall CPU load.

A simple companion program can then kill the looping program by sending a "stop" message down the pipe:

```
   DECLARE
      pipename   VARCHAR2 (12) := 'signaler';
      result     INTEGER := DBMS_PIPE.create_pipe (pipename);
   BEGIN
      DBMS_PIPE.pack_message ('stop');
      result := DBMS_PIPE.send_message (pipename);
   END;
```

You can also send other commands down the pipe—for example, a command to increase or decrease the sleep interval. By the way, this example uses a private pipe, so the STOP message needs to be sent by the same user account that is running the infinite loop. Also note that the database's namespace for private pipes is global across all sessions that the current user is running. So, if you want to have more than one program running the infinite loop, you need some extra logic to (1) create pipe names that are unique across sessions, and (2) determine the correct pipe name(s) through which you want to send the STOP command.

The WHILE Loop

The WHILE loop is a conditional loop that continues to execute as long as the Boolean condition defined in the loop boundary evaluates to TRUE. Because the WHILE loop execution depends on a condition and is not fixed, you should use a WHILE loop if you don't know in advance the number of times a loop must execute.

Here is the general syntax for the WHILE loop:

```
WHILE condition
LOOP
```

```
      executable statement(s)
   END LOOP;
```

where *condition* is a Boolean variable or an expression that evaluates to a Boolean value of TRUE, FALSE, or NULL. Each time an iteration of the loop's body is executed, the condition is checked. If it evaluates to TRUE, then the body is executed. If it evaluates to FALSE or NULL, then the loop terminates, and control passes to the next executable statement following the END LOOP statement.

The following table summarizes the properties of the WHILE loop.

Property	Description
How the loop is terminated	The WHILE loop terminates when the Boolean expression in its boundary evaluates to FALSE or NULL.
When the test for termination takes place	The test for termination of a WHILE loop takes place in the loop boundary. This evaluation occurs prior to the first and each subsequent execution of the body. The WHILE loop, therefore, is not guaranteed to execute its loop even a single time.
Reason to use this loop	Use the WHILE loop when: • You are not sure how many times you must execute the loop body. • You will want to conditionally terminate the loop. • You don't have to execute the body at least one time.

The WHILE loop's condition is tested at the beginning of the loop's iteration, before the body of the loop is executed. There are two consequences to this preexecution test:

- All the information needed to evaluate the condition must be set before the loop is executed for the first time.
- It is possible that the WHILE loop will not execute even a single time.

Here is an example of a WHILE loop from the *datemgr.pkg* file available on the book's website. It shows a boundary condition consisting of a complex Boolean expression. The WHILE loop may stop either because I have run out of date masks to attempt a conversion, or because I have successfully performed a conversion (and date_converted is now TRUE):

```
   /* File on web: datemgr.pkg */
   WHILE mask_index <= mask_count AND NOT date_converted
   LOOP
      BEGIN
         /* Try to convert string using mask in table row */
         retval := TO_DATE (value_in, fmts (mask_index));
         date_converted := TRUE;
      EXCEPTION
         WHEN OTHERS
         THEN
            mask_index:= mask_index+ 1;
      END;
   END LOOP;
```

The Numeric FOR Loop

There are two kinds of PL/SQL FOR loops: the numeric FOR loop and the cursor FOR loop. The numeric FOR loop is the traditional and familiar "counted" loop. The number of iterations of the FOR loop is known when the loop starts; it is specified in the range scheme found between the FOR and LOOP keywords in the boundary.

The range scheme implicitly declares the loop index (if it has not already been declared), specifies the start and end points of the range, and optionally dictates the order in which the loop index proceeds (from lowest to highest or highest to lowest).

Here is the general syntax of the numeric FOR loop:

```
FOR loop index IN [REVERSE] lowest number .. highest number
LOOP
   executable statement(s)
END LOOP;
```

You must have at least one executable statement between the LOOP and END LOOP keywords.

The following table summarizes the properties of the numeric FOR loop.

Property	Description
How the loop is terminated	The numeric FOR loop terminates unconditionally when the number of times specified in its range scheme has been satisfied. You can also terminate the loop with an EXIT statement, but this is not recommended.
When the test for termination takes place	After each execution of the loop body, PL/SQL increments (or decrements if REVERSE is specified) the loop index and then checks its value. When it exceeds the upper bound of the range scheme, the loop terminates. If the lower bound is greater than the upper bound of the range scheme, the loop never executes its body.
Reason to use this loop	Use the numeric FOR loop when you want to execute a body of code a fixed number of times and do not want to halt that looping prematurely.

Rules for Numeric FOR Loops

Follow these rules when you use numeric FOR loops:

- Do not declare the loop index. PL/SQL automatically and implicitly declares it as a local variable with datatype INTEGER. The scope of this index is the loop itself; you cannot reference the loop index outside the loop.
- Expressions used in the range scheme (both for lowest and highest bounds) are evaluated once, when the loop starts. The range is not reevaluated during the exe-

cution of the loop. If you make changes within the loop to the variables that you used to determine the FOR loop range, those changes will have no effect.

- Never change the values of either the loop index or the range boundary from within the loop. This is an extremely bad programming practice. PL/SQL will either produce a compile error or ignore your instructions; in either case, you'll have problems.
- Use the REVERSE keyword to force the loop to decrement from the upper bound to the lower bound. You must still make sure that the first value in the range specification (the *lowest number* in *lowest number .. highest number*) is less than the second value. Do not reverse the order in which you specify these values when you use the REVERSE keyword.

Examples of Numeric FOR Loops

These examples demonstrate some variations of the numeric FOR loop syntax:

- The loop executes 10 times; loop_counter starts at 1 and ends at 10:

```
FOR loop_counter IN 1 .. 10
LOOP
   ... executable statements ...
END LOOP;
```

- The loop executes 10 times; loop_counter starts at 10 and ends at 1:

```
FOR loop_counter IN REVERSE 1 .. 10
LOOP
   ... executable statements ...
END LOOP;
```

- Here is a loop that doesn't execute even once. I specified REVERSE, so the loop index, loop_counter, will start at the highest value and end with the lowest. I then mistakenly concluded that I should switch the order in which I list the highest and lowest bounds:

```
FOR loop_counter IN REVERSE 10 .. 1
LOOP
   /* This loop body will never execute even once! */
   ... executable statements ...
END LOOP;
```

  Even when you specify a REVERSE direction, you must still list the lowest bound before the highest bound. If the first number is greater than the second number, the body of the loop will not execute at all. If the lowest and highest bounds have the same value, the loop will execute just once.

- The loop executes for a range determined by the values in the variable and expression:

```
FOR calc_index IN start_period_number ..
            LEAST (end_period_number, current_period)
LOOP
   ... executable statements ...
END LOOP;
```

In this example, the number of times the loop will execute is determined at runtime. The boundary values are evaluated once, before the loop executes, and then applied for the duration of loop execution.

Handling Nontrivial Increments

PL/SQL does not provide a "step" syntax whereby you can specify a particular loop index increment. In all variations of the PL/SQL numeric FOR loop, the loop index is always incremented or decremented by one.

If you have a loop body that you want executed for a nontrivial increment (something other than one), you will have to write some cute code. For example, what if you want your loop to execute only for even numbers between 1 and 100? You can make use of the numeric MOD function, as follows:

```
FOR loop_index IN 1 .. 100
LOOP
   IF MOD (loop_index, 2) = 0
   THEN
      /* We have an even number, so perform calculation */
      calc_values (loop_index);
   END IF;
END LOOP;
```

Or you can use simple multiplication inside a loop with half the iterations:

```
FOR even_number IN 1 .. 50
LOOP
   calc_values (even_number*2);
END LOOP;
```

In both cases, the calc_values procedure executes only for even numbers. In the first example, the FOR loop executes 100 times; in the second example, it executes only 50 times.

Whichever approach you decide to take, be sure to document this kind of technique clearly. You are, in essence, manipulating the numeric FOR loop to do something for which it is not designed. Comments would be very helpful for the maintenance programmer who has to understand why you would code something like that.

The Cursor FOR Loop

A cursor FOR loop is a loop that is associated with (and actually defined by) an explicit cursor or a SELECT statement incorporated directly within the loop boundary. Use the cursor FOR loop only if you need to fetch and process each and every record from a cursor, which is often the case with cursors.

The cursor FOR loop is one of my favorite PL/SQL features. It leverages fully the tight and effective integration of the procedural constructs with the power of the SQL database language. It reduces the volume of code you need to write to fetch data from a cursor. It greatly lessens the chance of introducing loop errors in your programming—and loops are one of the more error-prone parts of a program. Does this loop sound too good to be true? Well, it isn't—it's all true!

Here is the basic syntax of a cursor FOR loop:

```
FOR record IN { cursor_name | (explicit SELECT statement) }
LOOP
   executable statement(s)
END LOOP;
```

where *record* is a record declared implicitly by PL/SQL with the %ROWTYPE attribute against the cursor specified by *cursor_name*.

Don't declare a record explicitly with the same name as the loop index record. It is not needed (PL/SQL declares one for its use within the loop implicitly) and can lead to logic errors. For tips on accessing information about a cursor FOR loop's record outside or after loop execution, see "Obtaining Information About FOR Loop Execution" on page 126.

You can also embed a SELECT statement directly in the cursor FOR loop, as shown in this example:

```
FOR book_rec IN (SELECT * FROM books)
LOOP
   show_usage (book_rec);
END LOOP;
```

You should, however, avoid this formulation because it results in the embedding of SELECT statements in "unexpected" places in your code, making it more difficult to maintain and enhance your logic.

The following table summarizes the properties of the cursor FOR loop where *record* is a record declared implicitly by PL/SQL with the %ROWTYPE attribute against the cursor specified by *cursor_name*.

Property	Description
How the loop is terminated	The cursor FOR loop terminates unconditionally when all of the records in the associated cursor have been fetched. You can also terminate the loop with an EXIT statement, but this is not recommended.
When the test for termination takes place	After each execution of the loop body, PL/SQL performs another fetch. If the %NOTFOUND attribute of the cursor evaluates to TRUE, then the loop terminates. If the cursor returns no rows, then the loop never executes its body.
Reason to use this loop	Use the cursor FOR loop when you want to fetch and process every record in a cursor.

You should use a cursor FOR loop whenever you need to unconditionally fetch all rows from a cursor (i.e., there are no EXITs or EXIT WHENs inside the loop that cause early termination). Let's take a look at how you can use the cursor FOR loop to streamline your code and reduce opportunities for error.

Example of Cursor FOR Loops

Suppose I need to update the bills for all pets staying in my pet hotel, the Share-a-Din-Din Inn. The following example contains an anonymous block that uses a cursor, occupancy_cur, to select the room number and pet ID number for all occupants of the Inn. The procedure update_bill adds any new changes to that pet's room charges:

```
 1   DECLARE
 2      CURSOR occupancy_cur IS
 3         SELECT pet_id, room_number
 4           FROM occupancy WHERE occupied_dt = TRUNC (SYSDATE);
 5      occupancy_rec occupancy_cur%ROWTYPE;
 6   BEGIN
 7      OPEN occupancy_cur;
 8      LOOP
 9         FETCH occupancy_cur INTO occupancy_rec;
10         EXIT WHEN occupancy_cur%NOTFOUND;
11         update_bill
12             (occupancy_rec.pet_id, occupancy_rec.room_number);
13       END LOOP;
14       CLOSE occupancy_cur;
15    END;
```

This code leaves nothing to the imagination. In addition to defining the cursor (line 2), you must explicitly declare the record for the cursor (line 5), open the cursor (line 7), start up an infinite loop (line 8), fetch a row from the cursor set into the record (line 9), check for an end-of-data condition with the %NOTFOUND cursor attribute (line 10), and finally perform the update (line 11). When you are all done, you have to remember to close the cursor (line 14).

If I convert this PL/SQL block to use a cursor FOR loop, then I have:

```
DECLARE
   CURSOR occupancy_cur IS
```

```
         SELECT pet_id, room_number
           FROM occupancy WHERE occupied_dt = TRUNC (SYSDATE);
   BEGIN
      FOR occupancy_rec IN occupancy_cur
      LOOP
         update_bill (occupancy_rec.pet_id, occupancy_rec.room_number);
      END LOOP;
   END;
```

Here you see the beautiful simplicity of the cursor FOR loop! Gone is the declaration of the record. Gone are the OPEN, FETCH, and CLOSE statements. Gone is the need to check the %NOTFOUND attribute. Gone are the worries of getting everything right. Instead, you say to PL/SQL, in effect:

> You and I both know that I want each row, and I want to dump that row into a record that matches the cursor. Take care of that for me, will you?

And PL/SQL does take care of it, just the way any modern programming language should.

As with all other cursors, you can pass parameters to the cursor in a cursor FOR loop. If any of the columns in the select list of the cursor is an expression, remember that you must specify an alias for that expression in the select list. Within the loop, the only way to access a particular value in the cursor record is with the dot notation (*record_name.column_name*, as in occupancy_rec.room_number), so you need a column name associated with the expression.

For more information about working with cursors in PL/SQL, check out Chapter 15.

Loop Labels

You can give a name to a loop by using a label. (I introduced labels in Chapter 3.) A loop label in PL/SQL has the following format:

```
<<label_name>>
```

where *label_name* is the name of the label, and that loop label appears immediately before the LOOP statement:

```
<<all_emps>>
FOR emp_rec IN emp_cur
LOOP
   ...
END LOOP;
```

The label can also appear optionally after the END LOOP reserved words, as the following example demonstrates:

```
<<year_loop>>
WHILE year_number <= 1995
LOOP
```

```
      <<month_loop>>
      FOR month_number IN 1 .. 12
      LOOP
         ...
      END LOOP month_loop;
      year_number := year_number + 1;

   END LOOP year_loop;
```

The loop label is potentially useful in several ways:

- When you have written a loop with a large body (say, one that starts at line 50, ends on line 725, and has 16 nested loops inside it), use a loop label to tie the end of the loop back explicitly to its start. This visual tag will make it easier for a developer to maintain and debug the program. Without the loop label, it can be very difficult to keep track of which LOOP goes with which END LOOP.
- You can use the loop label to qualify the name of the loop indexing variable (either a record or a number). Again, this can be helpful for readability. Here is an example:

  ```
  <<year_loop>>
  FOR year_number IN 1800..1995
  LOOP
     <<month_loop>>
     FOR month_number IN 1 .. 12
     LOOP
        IF year_loop.year_number = 1900 THEN ... END IF;
     END LOOP month_loop;
  END LOOP year_loop;
  ```

- When you have nested loops, you can use the label both to improve readability and to increase control over the execution of your loops. You can, in fact, stop the execution of a specific named outer loop by adding a loop label after the EXIT keyword in the EXIT statement of a loop, as follows:

  ```
  EXIT loop_label;
  EXIT loop_label WHEN condition;
  ```

  While it is possible to use loop labels in this fashion, I recommend that you avoid it. It leads to very unstructured logic (quite similar to GOTOs) that is hard to debug. If you feel that you need to insert code like this, you should consider restructuring your loop, and possibly switching from a FOR loop to a simple or WHILE loop.

The CONTINUE Statement

Oracle Database 11*g* offers a new feature for loops: the CONTINUE statement. Use this statement to exit the current iteration of a loop, and immediately continue on to the *next* iteration of that loop. This statement comes in two forms, just like EXIT: the unconditional CONTINUE and the conditional CONTINUE WHEN.

Here is a simple example of using CONTINUE WHEN to skip over loop body execution for even numbers:

```
BEGIN
   FOR l_index IN 1 .. 10
   LOOP
      CONTINUE WHEN MOD (l_index, 2) = 0;
      DBMS_OUTPUT.PUT_LINE ('Loop index = ' || TO_CHAR (l_index));
   END LOOP;
END;
/
```

The output is:

```
Loop index = 1
Loop index = 3
Loop index = 5
Loop index = 7
Loop index = 9
```

Of course, you can achieve the same effect with an IF statement, but CONTINUE may offer a more elegant and straightforward way to express the logic you need to implement.

CONTINUE is likely to come in handy mostly when you need to perform "surgery" on existing code, make some very targeted changes, and then immediately exit the loop body to avoid side effects.

You can also use CONTINUE to terminate an inner loop and proceed immediately to the next iteration of an outer loop's body. To do this, you will need to give names to your loops using labels. Here is an example:

```
BEGIN
   <<outer>>
   FOR outer_index IN 1 .. 5
   LOOP
      DBMS_OUTPUT.PUT_LINE (
         'Outer index = ' || TO_CHAR (outer_index));

      <<inner>>
      FOR inner_index IN 1 .. 5
      LOOP
         DBMS_OUTPUT.PUT_LINE (
            '  Inner index = ' || TO_CHAR (inner_index));
         CONTINUE outer;
      END LOOP inner;
   END LOOP outer;
END;
/
```

The output is:

```
Outer index = 1
  Inner index = 1
Outer index = 2
  Inner index = 1
Outer index = 3
  Inner index = 1
Outer index = 4
  Inner index = 1
Outer index = 5
  Inner index = 1
```

Is CONTINUE as Bad as GOTO?

When I first learned about the CONTINUE statement, my instinctive reaction was that it represented another form of unstructured transfer of control, similar to GOTO, and should therefore be avoided whenever possible (I'd been doing just fine without it for years!). Charles Wetherell, a senior member of the PL/SQL development team, set me straight as follows:

> From a long time ago (the era of Dijkstra's "goto" letter), exit and continue were discussed and understood to be structured transfers of control. Indeed, exit was directly recognized in one of Knuth's major programming language papers as a way to leave politely from a computation that you needed to abandon.
>
> Böhm and Jacopini proved that any program that uses any arbitrary synchronous control element (think of loop or goto) could be rewritten using only while loops, if statements, and Boolean variables in a completely structured way. Furthermore, the transformation between the bad unstructured version and the good structured version of a program could be automated. That's the good news. The bad news is that the new "good" program might be exponentially larger than the old program because of the need to introduce many Booleans and the need to copy code into multiple if statement arms. In practice, real programs do not experience this exponential explosion. But one often sees "cut and paste" code copied to simulate the effects of continue and exit. "Cut and paste" causes maintenance headaches because if a change is needed, the programmer must remember to make a change in every copy of the pasted code.
>
> The continue statement is valuable because it makes code shorter, makes code easier to read, and reduces the need for Boolean variables whose exact meaning can be hard to decipher. The most common use is a loop where the exact processing that each item needs depends on detailed structural tests of the item. The skeleton of a loop might look like this; notice that it contains an exit to decide when enough items have been processed. Also notice that the last continue (after condition5) is not strictly necessary. But by putting a continue after each action, it is easy to add more actions in any order without breaking any other actions.

```
LOOP
   EXIT WHEN exit_condition_met;
   CONTINUE WHEN condition1;
   CONTINUE WHEN condition2;
   setup_steps_here;
```

```
      IF condition4 THEN
         action4_executed;
         CONTINUE;
      END IF;

      IF condition5 THEN
         action5_executed;
         CONTINUE; -- Not strictly required.
      END IF;
   END LOOP;
```

Without continue, I would have to implement the loop body like this:

```
   LOOP
      EXIT WHEN exit_condition_met;

      IF condition1
      THEN
         NULL;
      ELSIF condition2
      THEN
         NULL;
      ELSE
         setup_steps_here;

         IF condition4 THEN
            action4_executed;
         ELSIF condition5 THEN
            action5_executed;
         END IF;
      END IF;
   END LOOP;
```

Even with this simple example, continue avoids numerous elsif clauses, reduces nesting, and shows clearly which Boolean tests (and associated processing) are on the same level. In particular, the nesting depth is much less when continue is used. PL/SQL programmers can definitely write better code once they understand and use continue correctly.

Tips for Iterative Processing

Loops are very powerful and useful constructs, but they are structures that you should use with care. Performance issues within a program often are traced back to loops, and any problem within a loop is magnified by its repeated execution. The logic determining when to stop a loop can be very complex. This section offers some tips on how to write loops that are clean, easy to understand, and easy to maintain.

Use Understandable Names for Loop Indexes

Software programmers should not have to make Sherlock Holmes–like deductions about the meaning of the start and end range values of the innermost FOR loops in order to understand their purpose. Use names that self-document the purposes of variables and loops. That way, other people will understand your code, and you will remember what your own code does when you review it three months later.

How would you like to try to understand—much less maintain—code that looks like this?

```
FOR i IN start_id .. end_id
LOOP
   FOR j IN 1 .. 7
   LOOP
      FOR k IN 1 .. 24
      LOOP
         build_schedule (i, j, k);
      END LOOP;
   END LOOP;
END LOOP;
```

It is hard to imagine that someone would write code based on such generic integer variable names (right out of Algebra 101), yet it happens all the time. The habits we pick up in our earliest days of programming have an incredible half-life. Unless you are constantly vigilant, you will find yourself writing the most abominable code. In the preceding case, the solution is simple—use variable names for the loop indexes that are meaningful and therefore self-documenting:

```
FOR focus_account IN start_id .. end_id
LOOP
   FOR day_in_week IN 1 .. 7
   LOOP
      FOR month_in_biyear IN 1 .. 24
      LOOP
         build_schedule (focus_account, day_in_week, month_in_biyear);
      END LOOP;
   END LOOP;
END LOOP;
```

Now that I have provided descriptive names for those index variables, I discover that the innermost loop actually spanned two sets of 12 months (12 × 2 = 24).

The Proper Way to Say Goodbye

One important and fundamental principle in structured programming is "one way in, one way out"; that is, a program should have a single point of entry and a single point of exit. A single point of entry is not an issue with PL/SQL: no matter what kind of loop you are using, there is always only one entry point into the loop—the first executable statement following the LOOP keyword. It is quite possible, however, to construct loops that have multiple exit paths. Avoid this practice. Having multiple ways of terminating a loop results in code that is much harder to debug and maintain.

In particular, you should follow these guidelines for loop termination:

- Do not use EXIT or EXIT WHEN statements within FOR and WHILE loops. You should use a FOR loop only when you want to iterate through all the values (integer or record) specified in the range. An EXIT inside a FOR loop disrupts this process and subverts the intent of that structure. A WHILE loop, on the other hand, specifies its termination condition in the WHILE statement itself.
- Do not use the RETURN or GOTO statements within a loop—again, these cause the premature, unstructured termination of the loop. It can be tempting to use these constructs because in the short run they appear to reduce the amount of time spent writing code. In the long run, however, you (or the person left to clean up your mess) will spend more time trying to understand, enhance, and fix your code over time.

Let's look at an example of loop termination issues with the cursor FOR loop. As you have seen, the cursor FOR loop offers many advantages when you want to loop through all of the records returned by a cursor. This type of loop is not appropriate, however, when you need to apply conditions to each fetched record to determine if you should halt execution of the loop. Suppose that you need to scan through each record from a cursor and stop when a total accumulation of a column (like the number of pets) exceeds a maximum, as shown in the following code. Although you can do this with a cursor FOR loop by issuing an EXIT statement inside the loop, it's an inappropriate use of this construct:

```
 1   DECLARE
 2      CURSOR occupancy_cur IS
 3         SELECT pet_id, room_number
 4           FROM occupancy WHERE occupied_dt = TRUNC (SYSDATE);
 5      pet_count INTEGER := 0;
 6   BEGIN
 7      FOR occupancy_rec IN occupancy_cur
 8      LOOP
 9         update_bill
10             (occupancy_rec.pet_id, occupancy_rec.room_number);
11          pet_count := pet_count + 1;
12          EXIT WHEN pet_count >= pets_global.max_pets;
13       END LOOP;
14    END;
```

The FOR loop explicitly states: "I am going to execute the body of this loop *n* times" (where *n* is a number in a numeric FOR loop, or the number of records in a cursor FOR loop). An EXIT inside the FOR loop (line 12) short-circuits this logic. The result is code that's difficult to follow and debug.

If you need to terminate a loop based on information fetched by the cursor FOR loop, you should use a WHILE loop or a simple loop in its place. Then the structure of the code will more clearly state your intentions.

Obtaining Information About FOR Loop Execution

FOR loops are handy and concise constructs. They handle lots of the “administrative work” in a program; this is especially true of cursor FOR loops. There is, however, a tradeoff: by letting the database do so much of the work for you, you have limited access to information about the end results of the loop after it has been terminated.

Suppose that I want to know how many records I processed in a cursor FOR loop and then execute some logic based on that value. It would be awfully convenient to write code like this:

```
BEGIN
   FOR book_rec IN books_cur (author_in => 'FEUERSTEIN,STEVEN')
   LOOP
      ... process data ...
   END LOOP;
   IF books_cur%ROWCOUNT > 10 THEN ...
```

but if I try it, I get the runtime error *ORA-01001: invalid cursor*. This makes sense, because the cursor is implicitly opened and closed by the database. So how can you get this information from a loop that is closed? You need to declare a variable in the block housing that FOR loop, and then set its value inside the FOR loop so that you can obtain the necessary information about the FOR loop after it has closed. This technique is shown here:

```
DECLARE
   book_count PLS_INTEGER := 0;
BEGIN
   FOR book_rec IN books_cur (author_in => 'FEUERSTEIN,STEVEN')
   LOOP
      ... process data ...
      book_count := books_cur%ROWCOUNT;
   END LOOP;
   IF book_count > 10 THEN ...
```

SQL Statement as Loop

You actually can think of a SQL statement like SELECT as a loop. After all, such a statement specifies an action to be taken on a set of data; the SQL engine then “loops through” the data set and applies the action. In some cases, you will have a choice between using a PL/SQL loop and a SQL statement to do the same or similar work. Let’s look at an example and then draw some conclusions about how you can decide which approach to take.

I need to write a program to move the information for pets who have checked out of the pet hotel from the occupancy table to the occupancy_history table. As a seasoned PL/SQL developer, I immediately settle on a cursor FOR loop. For each record fetched (implicitly) from the cursor (representing a pet who has checked out), the body of the loop first inserts a record into the occupancy_history table and then deletes the record from the occupancy table:

```
DECLARE
   CURSOR checked_out_cur IS
      SELECT pet_id, name, checkout_date
        FROM occupancy WHERE  checkout_date IS NOT NULL;
BEGIN
   FOR checked_out_rec IN checked_out_cur
   LOOP
      INSERT INTO occupancy_history (pet_id, name, checkout_date)
         VALUES (checked_out_rec.pet_id, checked_out_rec.name,
                 checked_out_rec.checkout_date);
      DELETE FROM occupancy WHERE pet_id = checked_out_rec.pet_id;
   END LOOP;
END;
```

This code does the trick. But was it necessary to do it this way? I can express precisely the same logic and get the same result with nothing more than an INSERT-SELECT FROM followed by a DELETE, as shown here:

```
BEGIN
   INSERT INTO occupancy_history (pet_id, NAME, checkout_date)
      SELECT pet_id, NAME, checkout_date
        FROM occupancy WHERE checkout_date IS NOT NULL;
   DELETE FROM occupancy WHERE checkout_date IS NOT NULL;
END;
```

What are the advantages to this approach? I have written less code, and my code will run more efficiently because I have reduced the number of context switches (moving back and forth between the PL/SQL and SQL execution engines). I execute just a single INSERT and a single DELETE.

There are, however, disadvantages to the 100% SQL approach. SQL statements are generally all-or-nothing propositions. In other words, if any one of those individual rows from occupancy_history fails, then the entire INSERT fails; no records are inserted or deleted. Also, the WHERE clause had to be coded twice. Although not a significant factor in this example, it may well be when substantially more complex queries are involved. The initial cursor FOR loop thus obviated the need to potentially maintain complex logic in multiple places.

PL/SQL offers more flexibility as well. Suppose, for example, that I want to transfer as many of the rows as possible, and simply write a message to the error log for any transfers of individual rows that fail. In this case, I really do need to rely on the cursor FOR loop, but with the added functionality of an exception section:

```
BEGIN
   FOR checked_out_rec IN checked_out_cur
   LOOP
      BEGIN
         INSERT INTO occupancy_history ...
```

```
            DELETE FROM occupancy ...
         EXCEPTION
            WHEN OTHERS THEN
               log_checkout_error (checked_out_rec);
         END;
      END LOOP;
   END;
   ;
```

PL/SQL offers the ability to access and process a single row at a time, and to take action (and, perhaps, complex procedural logic based on the contents of that specific record). When that's what you need, use a blend of PL/SQL and SQL. If, on the other hand, your requirements allow you to use native SQL, you will find that you can use less code and that it will run more efficiently.

You can continue past errors in SQL statements in two other ways: (1) use the LOG ERRORS clause with inserts, updates, and deletes in Oracle Database 10*g* Release 2 and later; and (2) use the SAVE EXCEPTIONS clause in your FORALL statements. See Chapter 21 for more details.