[Appendix A] Appendix: PL/SQL Exercises

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

A. Appendix: PL/SQL Exercises

Contents:

The exercises included in this appendix are designed to enhance your ability to write well−structured PL/SQL programs, and also to identify problems with existing code. I recommend that you test out your baseline PL/SQL skills on these exercises before you explore Parts III through V of this book, where you will learn how to apply your skills to building robust and reusable packages.

For solutions to these exercises, see Section A.2, "Solutions" later in this appendix.

A.1 Exercises

The exercises are arranged by topic:

Conditional logic
Loops
Exception handling
Cursors
Builtin functions
Builtin packages
Modules
Module evaluation

A.1.1 Conditional Logic

1. Rewrite the following IF statements so that you do *not* use an IF statement to set the value of **no_revenue**. What is the difference between these two statements? How does that difference affect your answer?

```
IF total_sales <= 0
THEN
   no_revenue := TRUE;
ELSE
   no_revenue := FALSE;
END IF;

IF total_sales <= 0
THEN
   no_revenue := TRUE;
ELSIF total_sales > 0
THEN
   no_revenue := FALSE;
END IF;
```

2. Rewrite the following IF statement to work as efficiently as possible under all conditions, given the following information: the **calc_totals** numeric function takes three minutes to return its value, while the **overdue_balance** Boolean function returns TRUE/FALSE in less than a second.

```
IF calc_totals (1994, company_id_in => 1005) AND
   NOT overdue_balance (company_id_in => 1005)
```

```
THEN
   display_sales_figures (1005);
ELSE
   contact_vendor;
END IF;
```

3. Rewrite the following IF statement to get rid of unnecessary nested IFs:

```
IF salary < 10000
THEN
   bonus := 2000;
ELSE
   IF salary < 20000
   THEN
      bonus := 1500;
   ELSE
      IF salary < 40000
      THEN
         bonus := 1000;
      ELSE
         bonus := 500;
      END IF;
   END IF;
END IF;
```

4. Which procedure will *never* be executed in this IF statement?

```
IF (order_date > SYSDATE) AND order_total >= min_order_total
THEN
   fill_order (order_id, 'HIGH PRIORITY');
ELSIF (order_date < SYSDATE) OR
      (order_date = SYSDATE)
THEN
   fill_order (order_id, 'LOW PRIORITY');
ELSIF order_date <= SYSDATE AND order_total < min_order_total
THEN
   queue_order_for_addtl_parts (order_id);
ELSIF order_total = 0
THEN
   disp_message (' No items have been placed in this order!');
END IF;
```

A.1.2 Loops

1. How many times does the following loop execute?

```
FOR year_index IN REVERSE 12 .. 1
LOOP
   calc_sales (year_index);
END LOOP:
```

2. Select the type of loop (FOR, WHILE, simple) appropriate to meet each of the following requirements:

   a. Set the status of each company whose company IDs are stored in a PL/SQL table to closed.

   b.

For each of twenty years in the loan−processing cycle, calculate the outstanding loan balance for the specified customer. If the customer is a preferred vendor, stop the calculations after twelve years.

c.
Display the name and address of each employee returned by the cursor.

d.
Scan through the list of employees in the PL/SQL table, keeping count of all salaries greater than $50,000. Don't even start the scan, though, if the table is empty or if today is a Saturday or if the first employee in the PL/SQL table is the president of the company.

3.
Identify the problems with (or areas for improvement in) the following loops. How would you change the loop to improve it?

a.

```
FOR i IN 1 .. 100
LOOP
   calc_totals (i);
   IF i > 75
   THEN
      EXIT;
   END IF;
END LOOP;
```

b.

```
OPEN emp_cur;
FETCH emp_cur INTO emp_rec;
WHILE emp_cur%FOUND
LOOP
   calc_totals (emp_rec.salary);
   FETCH emp_cur INTO emp_rec;
   EXIT WHEN emp_rec.salary > 100000;
END LOOP;
CLOSE emp_cur;
```

c.

```
FOR a_counter IN lo_val .. hi_val
LOOP
   IF a_counter > lo_val * 2
   THEN
      hi_val := lo_val;
   END IF;
END LOOP;
```

d.

```
DECLARE
   CURSOR emp_cur IS SELECT salary FROM emp;
   emp_rec emp_cur%ROWTYPE
BEGIN
   OPEN emp_cur;
   LOOP
      FETCH emp_cur INTO emp_rec;
      EXIT WHEN emp_cur%NOTFOUND;
      calc_totals (emp_rec.salary);
   END LOOP;
   CLOSE emp_cur;
END;
```

e.

```
WHILE no_more_data
LOOP
   read_next_line (text);
   no_more_data := text IS NULL;
   EXIT WHEN no_more_data;
END LOOP;
```

f.

```
FOR month_index IN 1 .. 12
LOOP
   UPDATE monthly_sales
      SET pct_of_sales = 100
    WHERE company_id = 10006
```

```
            AND month_number = month_index;
      END LOOP;
```

g.

```
      DECLARE
         CURSOR emp_cur IS SELECT ... ;
      BEGIN
         FOR emp_rec IN emp_cur
         LOOP
            calc_totals (emp_rec.salary);
         END LOOP;
         IF emp_rec.salary < 10000
         THEN
            DBMS_OUTPUT.PUT_LINE ('Give ''em a raise!');
         END IF;
         CLOSE emp_cur;
      END;
```

h.

```
      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);
         END LOOP;
      END;
```

4. How many times does the following WHILE loop execute?

```
DECLARE
   end_of_analysis BOOLEAN := FALSE;
   CURSOR analysis_cursor IS SELECT ...;
   analysis_rec analysis_cursor%ROWTYPE;
   next_analysis_step NUMBER;
   PROCEDURE get_next_record (step_out OUT NUMBER) IS
   BEGIN
      FETCH analysis_cursor INTO analysis_rec;
      IF analysis_rec.status = 'FINAL'
      THEN
         step_out := 1;
      ELSE
         step_out := 0;
      END IF;
   END;
BEGIN
   OPEN analysis_cursor;
   WHILE NOT end_of_analysis
   LOOP
      get_next_record (next_analysis_step);
      IF analysis_cursor%NOTFOUND AND
         next_analysis_step IS NULL
      THEN
         end_of_analysis := TRUE;
      ELSE
         perform_analysis;
      END IF;
   END LOOP;
END;
```

5. Rewrite the following loop so that you do not use a loop at all.

```
FOR i IN 1 .. 2
```

```
LOOP
   IF i = 1
   THEN
      give_bonus (president_id, 2000000);
   ELSIF i = 2
   THEN
      give_bonus (ceo_id, 5000000);
   END IF;
END LOOP;
```

6. What statement would you remove from this block? Why?

```
DECLARE
   CURSOR emp_cur IS
      SELECT ename, deptno, empno
        FROM emp
       WHERE sal < 2500;
   emp_rec emp_cur%ROWTYPE;
BEGIN
   FOR emp_rec IN emp_cur
   LOOP
      give_raise (emp_rec.empno, 10000);
   END LOOP;
END;
```

A.1.3 Exception Handling

1. In each of the following PL/SQL blocks, a VALUE_ERROR exception is raised (usually by an attempt to place too large a value into a local variable). Identify which exception handler (if any −− the exception could also go unhandled) will handle the exception by writing down the message that will be displayed by the call to PUT_LINE in the exception handler. Explain your choice.

a.

```
DECLARE
   string_of_5_chars VARCHAR2(5);
BEGIN
   string_of_5_chars := 'Steven';
END;
```

b.

```
DECLARE
   string_of_5_chars VARCHAR2(5);
BEGIN
   BEGIN
      string_of_5_chars := 'Steven';
   EXCEPTION
      WHEN VALUE_ERROR
      THEN
         DBMS_OUTPUT.PUT_LINE ('Inner block');
   END;
EXCEPTION
   WHEN VALUE_ERROR
   THEN
      DBMS_OUTPUT.PUT_LINE ('Outer block');
END;
```

c.

```
DECLARE
   string_of_5_chars VARCHAR2(5) := 'Eli';
BEGIN
   BEGIN
      string_of_5_chars := 'Steven';
   EXCEPTION
      WHEN VALUE_ERROR
      THEN
         DBMS_OUTPUT.PUT_LINE ('Inner block');
   END;
```

```
EXCEPTION
   WHEN VALUE_ERROR
   THEN DBMS_OUTPUT.PUT_LINE ('Outer block');
END;
```

d.

```
DECLARE
   string_of_5_chars VARCHAR2(5) := 'Eli';
BEGIN
   DECLARE
      string_of_3_chars VARCHAR2(3) := 'Chris';
   BEGIN
      string_of_5_chars := 'Veva';
   EXCEPTION
      WHEN VALUE_ERROR
      THEN DBMS_OUTPUT.PUT_LINE ('Inner block');
   END;
EXCEPTION
   WHEN VALUE_ERROR
   THEN DBMS_OUTPUT.PUT_LINE ('Outer block');
END;
```

e.

```
DECLARE
   string_of_5_chars VARCHAR2(5);
BEGIN
   BEGIN
      string_of_5_chars := 'Steven';
   EXCEPTION
      WHEN VALUE_ERROR
      THEN
         RAISE NO_DATA_FOUND;
      WHEN NO_DATA_FOUND
      THEN
         DBMS_OUTPUT.PUT_LINE ('Inner block');
   END;
EXCEPTION
   WHEN NO_DATA_FOUND
   THEN
      DBMS_OUTPUT.PUT_LINE ('Outer block');
END;
```

2.

Write a PL/SQL block that allows all of the following SQL DML statements to execute, even if any of the others fail:

```
UPDATE emp SET empno = 100 WHERE empno > 5000;
DELETE FROM dept WHERE deptno = 10;
DELETE FROM emp WHERE deptno = 10;
```

3.

Write a PL/SQL block that handles by name the following Oracle error:

```
ORA−1014: ORACLE shutdown in progress.
```

The exception handler should, in turn, raise a VALUE_ERROR exception. Hint: use the EXCEPTION INIT pragma.

4.

When the following block is executed, which of the two messages shown below are displayed? Explain your choice.

Message from Exception Handler	Output from Unhandled Exception
Predefined or programmer−defined?	Error at line 1: ORA−1403: no data found ORA−6512: at line 5

```
DECLARE
```

```
   d VARCHAR2(1);
   /* Create exception with a predefined name. */
   no_data_found EXCEPTION;
BEGIN
   SELECT dummy INTO d FROM dual WHERE 1=2;
   IF d IS NULL
   THEN
      /*
      || Raise my own exception, not the predefined
      || STANDARD exception of the same name.
      */
      RAISE no_data_found;
   END IF;
EXCEPTION
   /* This handler only responds to the RAISE statement. */
   WHEN no_data_found
   THEN
      DBMS_OUTPUT.PUT_LINE ('Predefined or programmer−defined?');
END;
```

5. I create the **`getval`** package as shown below. I then call DBMS_OUTPUT.PUT_LINE to display the value returned by the **`getval.get`** function. What is displayed on the screen?

```
CREATE OR REPLACE PACKAGE getval
IS
   FUNCTION get RETURN VARCHAR2;
END getval;
/
CREATE OR REPLACE PACKAGE BODY getval
IS
   v VARCHAR2(1) := 'abc';
   FUNCTION get RETURN VARCHAR2 IS
   BEGIN
      RETURN v;
   END;
BEGIN
   NULL;
EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE ('Trapped!');
END getval;
/
```

A.1.4 Cursors

1. What cursor−related statements are missing from the following block?

```
DECLARE
   CURSOR emp_cur IS SELECT * FROM emp;
BEGIN
   OPEN emp_cur;
   FETCH emp_cur INTO emp_rec;
END;
```

2. What statement should be *removed* from the following block?

```
DECLARE
   CURSOR emp_cur IS SELECT * FROM emp;
   emp_rec emp_cur%ROWTYPE;
BEGIN
   FOR emp_rec IN emp_cur
   LOOP
      give_raise (emp_rec.empno);
   END LOOP;
END;
```

3. Name the cursor attribute (along with the cursor name) you would use (if any) for each of the following requirements:

   a. If the FETCH did not return any records from the **company_cur** cursor, exit the loop.

   b. If the number of rows deleted exceeded 100, notify the manager.

   c. If the **emp_cur** cursor is already open, fetch the next record. Otherwise, open the cursor.

   d. If the FETCH returns a row from the **sales_cur** cursor, display the total sales information.

   e. I use an implicit cursor SELECT statement to obtain the latest date of sales for my store number 45067. If no data is fetched or returned by the SELECT, display a warning.

4. What message is displayed in the following block if the SELECT statement does not return a row?

```
PROCEDURE display_dname (emp_in IN INTEGER) IS
   department# dept.deptno%TYPE := NULL;
BEGIN
   SELECT deptno INTO department#
     FROM emp
    WHERE empno = emp_in;
   IF department# IS NULL
   THEN
      DBMS_OUTPUT.PUT_LINE ('Dept is not found!');
   ELSE
      DBMS_OUTPUT.PUT_LINE ('Dept is ' || TO_CHAR (department#));
   END IF;
EXCEPTION
   WHEN NO_DATA_FOUND
   THEN
      DBMS_OUTPUT.PUT_LINE ('No data found');
END;
```

5. What message is displayed in the following block if there are no employees in department 15?

```
PROCEDURE display_dept_count
IS
   total_count INTEGER := 0;
BEGIN
   SELECT COUNT(*) INTO total_count
     FROM emp
    WHERE deptno = 15;
   IF total_count = 0
   THEN
      DBMS_OUTPUT.PUT_LINE ('No employees in department!');
   ELSE
      DBMS_OUTPUT.PUT_LINE
```

```
         ('Count of employees in dept 15 = ' || TO_CHAR (total_count));
   END IF;
EXCEPTION
   WHEN NO_DATA_FOUND
   THEN
      DBMS_OUTPUT.PUT_LINE ('No data found');
END;
```

6. If you fetch past the last record in a cursor's result set, what will happen?

7. How would you change the SELECT statement in the following block's cursor so that the block can display the sum of salaries in each department?

```
DECLARE
   CURSOR tot_cur IS
      SELECT deptno, SUM (sal)
        FROM emp
       GROUP BY deptno;
BEGIN
   FOR tot_rec IN tot_cur
   LOOP
      DBMS_OUTPUT.PUT_LINE
         ('Total is: ' || tot_rec.total_sales);
   END LOOP;
END;
```

8. Rewrite the following block to use a cursor parameter. Then rewrite to use a local module, as well as a cursor parameter.

```
DECLARE
   CURSOR dept10_cur IS
      SELECT dname, SUM (sal) total_sales
        FROM emp
       WHERE deptno = 10;
   dept10_rec dept10_cur%ROWTYPE;
   CURSOR dept20_cur IS
      SELECT dname, SUM (sal)
        FROM emp
       WHERE deptno = 20;
   dept20_rec dept20_cur%ROWTYPE;
BEGIN
   OPEN dept10_cur;
   FETCH dept10_cur INTO dept10_rec;
   DBMS_OUTPUT.PUT_LINE
      ('Total for department 10 is: ' || tot_rec.total_sales);
   CLOSE dept10_cur;
   OPEN dept20_cur;
   FETCH dept20_cur INTO dept20_rec;
   DBMS_OUTPUT.PUT_LINE
      ('Total for department 20 is: ' || tot_rec.total_sales);
   CLOSE dept20_cur;
END;
```

9. Place the following cursor inside a package, declaring the cursor as a *public* element (in the specification). The SELECT statement contains all of the columns in the **`emp`** table, in the same order.

```
CURSOR emp_cur (dept_in IN INTEGER) IS
   SELECT empno, ename, job, mgr, hiredate, sal, comm, deptno
     FROM emp
```

```
        WHERE deptno = dept_in;
```

A.1.5 Builtin Functions

1. Identify the appropriate builtin to use for each of the following requirements:

Requirement	Builtin
Calculate the number of days until the end of the month.	
Capitalize the first character in a word and lowercase the rest of the word.	
Convert a date to a string.	
Convert a number to a string.	
Convert a string to a date.	
Convert a string to lower case.	
Determine the length of a string.	
Determine the place of a character in the collating sequence of the character set used by the database.	
Extract the last four characters in a string.	
Extract the word found between the first and second _ delimiters in a string.	
Fill out a number in a string with leading zeroes.	
Find the last blank in a string.	
Find the Saturday nearest to the last day in March 1992.	
Find the third S in a string	
Get the first day in the month for a specified date.	
How many months are between **date1** and **date2**?	
I store all my names in uppercase in the database, but want to display them in reports in upper and lowercase.	
If it is High Noon in New York, what time is it in Calcutta?	
Remove a certain prefix from a string (for example, change **std_company_id** to **company_id**).	
Replace all instances of _ with a #.	
Return the error message associated with a SQL error code.	
Return the largest integer less than a specified value.	
Review all new hires on the first Wednesday after they'd been working for three months.	
Strip all leading numeric digits from a string.	
What is the current date and time?	
What is the date of the last day in the month?	

2. What portion of the string "Curious George deserves what he gets!" (assigned to variable **curious_george**) is returned by each of the following calls to SUBSTR:

```
1234567890123456789012345678901234567
Curious George deserves what he gets!
```

SUBSTR Usage	Returns
`SUBSTR (curious_george, −1)`	
`SUBSTR (curious_george, 1, 7)`	
`SUBSTR (curious_george, 9, 6)`	
`SUBSTR (curious_george, −8, 2)`	

`SUBSTR (curious_george,`
`INSTR (curious_george, −1, ' ') + 1)`	

`SUBSTR (curious_george,`
`INSTR (curious_george, ' ', −1, 3) + 1,`
`LENGTH ('cute'))`	

`SUBSTR (curious_george, −1 * LENGTH (curious_george))`	

A.1.6 Builtin Packages

1. What program would you use to calculate the elapsed time of your PL/SQL code execution? To what degree of accuracy can you obtain these timings?

2. What would you call to make your PL/SQL program pause for a specified number of seconds? What other techniques can you think of which would have this same effect?

3. What package can you use to determine if the current session has issued a COMMIT? How would you go about obtaining this information?

4. What do you see when you execute the following statements in SQL*Plus (assuming that you have already called SET SERVEROUTPUT ON):

```
SQL> execute DBMS_OUTPUT.PUT_LINE (100);
SQL> execute DBMS_OUTPUT.PUT_LINE ('     Five spaces in');
SQL> execute DBMS_OUTPUT.PUT_LINE (NULL);
SQL> execute DBMS_OUTPUT.PUT_LINE (SYSDATE < SYSDATE − 5);
SQL> execute DBMS_OUTPUT.PUT_LINE (TRANSLATE ('abc', NULL));
SQL> execute DBMS_OUTPUT.PUT_LINE (RPAD ('abc', 500, 'def'));
```

5. When an error occurs in your program, you want to be able to see which program is currently executing. What builtin packaged function would you call to get this information? If the current program is a procedure named **`calc_totals`** in the analysis package, what would you see when you call the builtin function?

6. You want to build a utility for DBAs that would allow them to create an index from within a PL/SQL program. Which package would you use? Which programs inside that package would be needed?

7.

You need to run a stored procedure named **`update_data`** every Sunday at 4 AM to perform a set of batch processes. Which builtin package would you use to perform this task? You will need to pass a string to the **`submit`** program to tell it how often to run **`update_data`**. What would that string be?

A.1.7 Modules

1. In each of the following modules, identify changes you would make to improve their structure, performance, or functionality.

   a.
   ```
   FUNCTION status_desc (status_cd_in IN VARCHAR2) RETURN VARCHAR2
   IS
   BEGIN
      IF    status_cd_in = 'C' THEN RETURN 'CLOSED';
      ELSIF status_cd_in = 'O' THEN RETURN 'OPEN';
      ELSIF status_cd_in = 'A' THEN RETURN 'ACTIVE';
      ELSIF status_cd_in = 'I' THEN RETURN 'INACTIVE';
      END IF;
   END;
   ```
   b.
   ```
   FUNCTION status_desc
               (status_cd_in IN VARCHAR2, status_dt_out OUT DATE)
   RETURN VARCHAR2
   IS
   BEGIN
      ... /* same function as above */
   END;
   ```
   c.
   ```
   FUNCTION company_name (company_id_in IN company.company_id%TYPE)
      RETURN VARCHAR2
   IS
      cname company.company_id%TYPE;
      found_it EXCEPTION;
   BEGIN
      SELECT name INTO cname FROM company
       WHERE company_id = company_id_in;
      RAISE found_it;
   EXCEPTION
      WHEN NO_DATA_FOUND
      THEN
          RETURN NULL;
      WHEN found_it
       THEN
          RETURN cname;
   END;
   ```
   d.
   ```
   PROCEDURE compute_net (company_id IN INTEGER)
   IS
      balance_remaining NUMBER := annual_sales (company_id);
   BEGIN
      FOR month_index IN 1 .. 12
      LOOP
         IF balance_remaining <= 0
         THEN
            RETURN 0;
         ELSE
            balance_remaining := debt (company_id, month_index);
         END IF;
      END LOOP;
   END;
   ```
2. Given the header for **`calc_profit`** below, which of the following calls to **`calc_profit`** are valid:

   ```
   PROCEDURE calc_profit
      (company_id_in IN NUMBER,
   ```

```
          profit_out OUT NUMBER
          fiscal_year_in IN NUMBER,
          profit_type_in IN VARCHAR2 := 'NET_PROFITS',
          division_in IN VARCHAR2 := 'ALL_DIVISIONS')
```

Call to calc_profit	Good/Bad? Why?

`calc_profit`
`   (1005, profit_level, 1995, 'ALL', 'FINANCE');`	

`calc_profit`
`   (new_company, profit_level);`	

`calc_profit`
`   (company_id_in => 32, fiscal_year_in => 1995,`
`    profit_out => big_number);`	

`calc_profit`
`   (company_id_in => 32, fiscal_year_in => 1995,`
`    profit_out => 1000);`	

3. Suppose that I have a general utility that displays the contents of a PL/SQL table of dates. The header for this procedure is:

   ```
   PROCEDURE dispdates
      (table_in IN PLVtab.date_table,
       num_rows_in IN INTEGER,
       header_in IN VARCHAR2 := NULL);
   ```

   where **`PLVtab.date_table`** is a predefined table TYPE stored in the PLVtab package. Notice that the default value for the header is NULL, which means that no header is displayed with the table contents.

   Here is an example of a call to this program:

   ```
   dispdates (birthdays, bday_count, 'List of Birthdays');
   ```

   Now suppose that you had to use **`dispdates`** to satisfy the following requirement: "Display the list of company start dates stored in the date table without any header." I can think of two ways do this:

   ```
   dispdates (company_list, num_companies);
   ```

   and:

   ```
   dispdates (company_list, num_companies, NULL);
   ```

   Which of these implementations would you choose and why? Is there any reason to choose one over the other?

A.1.8 Module Evaluation: Foreign Key Lookup

I have found that there are two ways to improve your skills in module construction:

1. Write lots of procedures and functions.
2. Critique someone else's efforts.

Certainly, there is no substitute for doing the work yourself. I find, on the other hand, that when I have the opportunity to look at another developer's work, a different kind of dynamic sets in. I am not sure that it speaks well of my personality, but I find it a whole lot easier to find the weaknesses in someone else's programs than in my own.

So assuming that everyone in the world in the same as me (a scary thought, but one I must entertain as a possibility), I offer a function for you to evaluate that I built myself long ago that does foreign−key lookups. No holds barred. No one to insult. See just how many problems you can find in the **`getkey_clrtyp`**. You might even try to rewrite the program to suit your tastes −− and then evaluate that!

We spend way too much of our time writing software to perform foreign key lookups. And in many situations, the interface we offer to our users to support easy access to foreign key information is inadequate. The approach I like to take is to hide the foreign keys themselves (users rarely need to know, after all, that the ID number for Acme Flooring, Inc. is 2765). Instead, I let the user type in as much of the name as she wants. I then see if there if there are any matches for that string. If there are no matches, I prompt for another entry. If there is just one match, I return the full name and the ID to the host application. If there are more than one match, I display a list.

The **`getkey_clrtyp`** function encapsulates this logic. The function itself returns a numeric code as follows:

0 = No match	1 = Unique	2 = Duplicate

It also returns through the parameter list the full name of the caller type and the numeric foreign key value. This function has a number of weaknesses in its design. See how many you can identify.

```
FUNCTION GETKEY_CLRTYP
(NAME_INOUT IN OUT VARCHAR2, NU_INOUT OUT NUMBER)
RETURN NUMBER IS
   CURSOR CLRTYP_CUR IS
      SELECT TYP_NU, TYPE_DS
        FROM CALLER_TYPE
       WHERE TYPE_DS LIKE NAME_INOUT || '%';
   CLRTYP_REC CLRTYP_CUR%ROWTYPE;
   NEXT_REC CLRTYP_CUR%ROWTYPE;
   TYP_NU VARCHAR2(10) := NULL;
   RETVAL NUMBER := NULL;
BEGIN
IF NAME_INOUT IS NOT NULL
THEN
OPEN CLRTYP_CUR;
   FETCH CLRTYP_CUR INTO CLRTYP_REC;
   IF CLRTYP_CUR%NOTFOUND
   THEN RETURN 0; ELSE
   FETCH CLRTYP_CUR INTO NEXT_REC;
   IF CLRTYP_CUR%NOTFOUND
   THEN RETVAL := 1;
   ELSE RETVAL := 2;
   END IF;
   NU_INOUT := CLRTYP_REC.TYP_NU;
   NAME_INOUT := CLRTYP_REC.TYP_DS;
   END IF;
CLOSE CLRTYP_CUR;
RETURN RETVAL;
END IF;
END GETKEY_CLRTYP;
```

PREVIOUS
VI. Testing Your Knowledge

HOME
BOOK INDEX

NEXT
A.2 Solutions

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

A.2 Solutions

This section contains the answers to the exercises shown earlier in this appendix.

A.2.1 Conditional Logic

1. *Rewrite the following IF statements so that you do not use the IF statement to set the value of no_revenue. What is the difference between the two statements?*

   The first IF statement can be simplified to:

   ```
   no_revenue := NVL (total_sales, 1) <= 0;
   ```

   I use NVL to make sure that **no_revenue** is set to FALSE, as would happen in the original IF statement. Without using NVL, I will set **no_revenue** to NULL if **total_sales** is NULL.

   The second statement is a bit more complicated, again due to the complexities of handling NULL values. If **total_sales** is NULL, the IF statement does not assign a value to **no_revenue** at all. NULL is never less than or equal to any number. So I still need an IF statement, but not (strictly speaking!) to assign a value to **no_revenue**:

   ```
   IF total_sales IS NOT NULL
   THEN
      no_revenue := total_sales <= 0;
   END IF;
   ```

2. *Rewrite the following IF statement to work as efficiently as possible under all conditions, given the following information: the* **calc_totals** *numeric function takes 3 minutes to return its value, while the* **overdue_balance** *Boolean function returns TRUE/FALSE in less than a second.*

   ```
   IF NOT overdue_balance (company_id_in => 1005)
   THEN
      IF calc_totals (1994, company_id_in => 1005)
      THEN
         display_sales_figures (1005);
      ELSE
         contact_vendor
      END IF;
   ELSE
      contact_vendor;
   END IF;
   ```

3. *Rewrite the following IF statement to get rid of unnecessary nested IFs:*

   ```
   IF salary < 10000
   ```

```
THEN
   bonus := 2000;
ELSIF salary < 20000
THEN
   bonus := 1500;
ELSIF salary < 40000
THEN
   bonus := 1000;
ELSE
   bonus := 500;
END IF;
```

4. Which procedure will never be executed in this IF statement?

   The call to **`queue_order_for_addtl_parts`** will never run since the previous ELSIF clause will always be true first.

A.2.2 Loops

1. *How many times does the following loop execute?*

   Not a single time. The first number in the range scheme must *always* be the smaller value.

2. *Select the type of loop (FOR, WHILE, simple) appropriate to meet each of the following requirements:*

   a. Numeric FOR loop.

   b. Simple or WHILE loop. The main thing is to not use a FOR loop since there is a conditional exit.

   c. Display the name and address of each employee returned by the cursor. Cursor FOR loop.

   d. WHILE loop, since there are conditions under which you do not want the loop body to execute even a single time.

3. *Identify the problems with (or areas for improvement in) the following loops. How would you change the loop to improve it?*

   a. Do not use a generic loop index name (i). In addition, the conditional EXIT from the FOR loop should be removed. Instead, use a FOR loop that loops from 1 to 76 (can you see why?).

   b. This loop relies on two diferent FETCH statements. Better off using a simple loop and just a single FETCH inside the loop. In addition, you should not EXIT from inside a WHILE loop. You should instead rely on the loop boundary condition.

   c.

Never attempt to change the values used in the range scheme. It will not actually affect the execution of the loop, since the range scheme is evaluated only once, at the time the loop begins. Such an assignment remains, however, a very bad programming practice.

d.
First, this program will not compile, since **`emp_rec`** record has not been defined. Second, you don't really need to declare that record because you should instead use instead a cursor FOR loop to reduce code volume.

e.
Do not use EXIT WHEN inside WHILE loop. Should only rely on changes in loop boundary condition.

f.
You should not use a PL/SQL loop at all. Instead employ straight SQL as follows:

```
UPDATE monthly_sales
   SET pct_of_sales = 100
 WHERE company_id = 10006
   AND month_number BETWEEN 1 AND 12;
```

g.
Never declare the cursor loop index (**`emp_rec`**). The reference to **`emp_rec.salary`** after the loop references the still−null record declared in the block, not the record filled inside the loop (which has terminated and erased that record). Also, the final CLOSE will attempt to close a closed cursor.

h.
Do not use a PL/SQL loop. Instead, INSERT directly from the SELECT statement.

```
INSERT INTO occupancy_history (pet_id, name, checkout_date)
   SELECT pet_id, name, checkout_date
     FROM occupancy
    WHERE checkout_date IS NOT NULL;
```

4.
How many times does the following WHILE loop execute?

An infinite number of times. This is an infinite WHILE loop. The local module called inside the loop never returns NULL for **`step_out`**, so **`next_analysis_step`** is never NULL, so the loop never terminates.

5.
Rewrite the following loop so that you do not use a loop at all.

This is a "phony loop." You don't need the loop or the IF statement. Just execute the code sequentially.

```
give_bonus (president_id, 2000000);
give_bonus (ceo_id, 5000000);
```

6.
What statement would you remove from this block? Why?

Remove the declaration of the **`emp_rec`** record. The cursor FOR loop implicitly declares a record of the right structure for you. In fact, the **`emp_rec`** record declared right after the cursor is never used in the block.

A.2.3 Exception Handling

1. *In each of the following PL/SQL blocks, a VALUE_ERROR exception is raised (usually by an attempt to place too large a value into a local variable). Identify which exception handler (if any −− the exception could also go unhandled) will handle the exception by writing down the message that will be displayed by the call to PUT_LINE in the exception handler. Explain your choice.*

   a. VALUE_ERROR is raised because "Steven" has six characters and the maximum length of string is five characters.

   b. Exception is unhandled. There is no exception section.

   c. "Inner block" is displayed.

   d. "Inner block" is displayed.

   e. "Outer block" is displayed.

   f. "Outer block" is displayed. The inner NO_DATA_FOUND handler does not come into play.

2. *Write a PL/SQL block that allows all of the following SQL DML statements to execute, even if the any of the others fail:*

```
BEGIN
   BEGIN
      UPDATE emp SET empno = 100 WHERE empno > 5000;
   EXCEPTION
      WHEN OTHERS THEN NULL;
   END;
   BEGIN
      DELETE FROM dept WHERE deptno = 10;
   EXCEPTION
      WHEN OTHERS THEN NULL;
   END;
   BEGIN
      DELETE FROM emp WHERE deptno = 10;
   EXCEPTION
      WHEN OTHERS THEN NULL;
   END;
END;
```

3. *Write a PL/SQL block that handles by name the following Oracle error:*

```
ORA−1014: ORACLE shutdown in progress.
```

   The exception handler should handle the error by propagating the exception by in turn raising a VALUE_ERROR exception. Hint: use the EXCEPTION INIT pragma.

```
DECLARE
```

```
   shutting_down EXCEPTION;
   PRAGRA EXCEPTION INIT (shutting_down, 1014);

BEGIN

   ... code ...
EXCEPTION
   WHEN shutting_down
   THEN
      ... handler ...
END;
```

4.

When the following block is executed, which of these two messages are displayed?

The ORA−1403 error message is displayed. The exception, in other words, goes unhandled. Since I have defined a local exception with same name as system predefined, that identifier overrides the system exception in this block. So when the SELECT statement raised NO_DATA_FOUND and PL/SQL moves to the exception section, it does not find a match.

You could make sure that the system exception is handled by changing the handler as follows:

```
WHEN SYSTEM.NO_DATA_FOUND
THEN
   ...
```

By qualifying the name of the exception, you tell PL/SQL which one you want to handle.

5.

I create the getval package as shown below. I then call DBMS_OUTPUT.PUT_LINE to display the value returned by the **`getval.get`** *function. What is displayed on the screen?*

```
ERROR at line 1:
ORA−06502: PL/SQL: numeric or value error
ORA−06512: at line 2
```

The error, in other words, goes unhandled. When I first reference the **`getval.get`** function, the package data is instantiated for my session. It then attempts to declare the private **`v`** variable. The default value for the variable is, unfortunately, too large for the variable, so PL/SQL raises the VALUE_ERROR exception. The exception section of the package only can handle exceptions raised in the initialization section of the package, so this error is unhandled and quite unhandleable from within the package itself.

A.2.4 Cursors

1.

What cursor−related statements are missing from the following block?

Missing a CLOSE **statement and a declaration of the `emp_rec` record.**

2.

What statement should be removed from the following block?

Remove the declaration of **`emp_rec`**.

3.

Name the cursor attribute (along with the cursor name) you would use (if any) for each of the following requirements:

`company_cur%NOTFOUND`

b. **`SQL%ROWCOUNT`**

c. **`emp_cur%ISOPEN`**

d. **`sales_cur%FOUND`**

e. No cursor attribute can be used. Instead, rely on an exception handler for NO_DATA_FOUND.

4. *What message is displayed in the following block if the SELECT statement does not return a row?*

   "No data found." If an implicit cursor does not return any rows, PL/SQL raises the NO_DATA_FOUND exception.

5. *What message is displayed in the following block if there are no employees in department 15?*

   "No employees in department!" is displayed. This SELECT statement does not find any rows, but since it is a group operation, SQL returns a single value of 0.

6. *If you fetch past the last record in a cursor's result set, what will happen?*

   Nothing. PL/SQL does not raise any errors, nor does it return any data into the record or variable list of the FETCH statement.

7. *How would you change the SELECT statement in the following block's cursor so that the block can display the sum of salaries in each department?*

   Add a column alias "total sales" right after the SUM (SAL) expression.

8. Rewrite the following block to use a cursor parameter. Then rewrite to use a local module, as well as a cursor parameter.

   a. With cursor parameter:

```
DECLARE
   CURSOR dept_cur (dept_in IN emp.deptno%TYPE) IS
      SELECT dname, SUM (sal) total_sales
        FROM emp
       WHERE deptno = dept_in;
   dept_rec dept_cur%ROWTYPE;
BEGIN
   OPEN dept_cur (10);
   FETCH dept_cur INTO dept_rec;
   DBMS_OUTPUT.PUT_LINE
```

```
                ('Total for department 10 is: ' || tot_rec.total_sales);
             CLOSE dept_cur;
             OPEN dept_cur;
             FETCH dept_cur INTO dept_rec;
             DBMS_OUTPUT.PUT_LINE
                ('Total for department 20 is: ' || tot_rec.total_sales);
             CLOSE dept_cur;
          END;
```

b.
With local module and cursor parameter:

```
DECLARE
   CURSOR dept_cur (dept_in IN emp.deptno%TYPE) IS
      SELECT dname, SUM (sal) total_sales
        FROM emp
       WHERE deptno = dept_in;
   dept_rec dept_cur%ROWTYPE;
   PROCEDURE display_dept (dept_in IN emp.deptno%TYPE) IS
   BEGIN
      OPEN dept_cur (dept_in);
      FETCH dept_cur INTO dept_rec;
      DBMS_OUTPUT.PUT_LINE
         ('Total for department ' || TO_CHAR (dept_in) ||
          ' is: ' || tot_rec.total_sales);
      CLOSE dept_cur;
   END;
BEGIN
   display_dept (10);
   display_dept (20);
END;
```

9.
Place the following cursor inside a package, declaring the cursor as a public element (in the specification). The SELECT statement contains all of the columns in the **emp** *table, in the same order.*

```
PACKAGE emp
IS
   CURSOR emp_cur (dept_in IN INTEGER) RETURN emp%ROWTYPE;
END emp;
PACKAGE emp
IS
   CURSOR emp_cur (dept_in IN INTEGER) RETURN emp%ROWTYPE
   IS
      SELECT empno, ename, job, mgr, hiredate, sal, comm, deptno
        FROM emp
       WHERE deptno = dept_in;
END emp;
```

A.2.5 Builtin Functions

1.
Identify the appropriate builtin to use for each of the following requirements:

Requirement	Builtin
Calculate the number of days until the end of the month.	LAST_DAY SYSDATE
Capitalize the first character in a word and lower−case the rest of the word.	INITCAP
Convert a date to a string.	TO_CHAR

Convert a number to a string.	TO_CHAR
Convert a string to a date.	TO_DATE
Convert a string to lower case.	LOWER
Determine the length of a string.	LENGTH
Determine the place of a character in the collating sequence of the character set used by the database.	ASCII
Extract the last four characters in a string.	SUBSTR
Extract the word found between the first and second _ delimiters in a string.	INSTR and SUBSTR
Fill out a number in a string with leading zeroes.	LPAD
Find the last blank in a string.	INSTR
Find the Saturday nearest to the last day in March 1992.	ROUND
Find the third S in a string	UPPER and INSTR
Get the first day in the month for a specified date.	TRUNC
How many months are between **date1** and **date2**?	MONTHS_BETWEEN
I store all my names in uppercase in the database, but want to display them in reports in upper− and lowercase.	INITCAP
If it is High Noon in New York, what time is it in Calcutta?	NEW_TIME
Remove a certain prefix from a string (for example, change **std_company_id** to **company_id**).	LTRIM or (better yet) SUBSTR
Replace all instances of _ with a #.	REPLACE
Return the error message associated with a SQL error code.	SQLERRM
Return the largest integer less than a specified value.	FLOOR or (TRUNC and DECODE)
Review all new hires on the first Wednesday after they'd been working for three months.	NEXT_DAY ADD_MONTHS
Strip all leading numeric digits from a string.	LTRIM
What is the current date and time?	SYSDATE
What is the date of the last day in the month?	LAST_DAY

2. What portion of the string "Curious George deserves what he gets!" (assigned to variable **curious_george**) is returned by each of the following calls to SUBSTR:

SUBSTR Usage	Returns
SUBSTR (curious_george, −1)	!
SUBSTR (curious_george, 1, 7)	Curious
SUBSTR (curious_george, 9 6)	George
SUBSTR (curious_george, −8, 2)	he
SUBSTR (curious_george, INSTR (curious_george, −1, ' ') + 1)	gets!
SUBSTR (curious_george, INSTR (curious_george, −1, ' ', 3) + 1,	what

`LENGTH ('cute'))`	
`SUBSTR (curious_george, −1 * LENGTH (curious_george))`	entire string

A.2.6 Builtin Packages

1. What program would you use to calculate the elapsed time of your PL/SQL code execution? To what degree of accuracy can you obtain these timings?

   The DBMS_UTILITY.GET_TIME function returns the number of hundredths of seconds that have elapsed since the *last* call to DBMS_UTILITY.GET_TIME. So if you compare consecutive calls to this builtin, you have the elapsed time down to the hundredth of a second.

2. *What would you call to make your PL/SQL program pause for a specified number of seconds? What other techniques can you think of which would have this same effect?*

   The DBMS_LOCK.SLEEP procedure will put your PL/SQL program to sleep for the number of seconds you pass to it. You can also make a PL/SQL program pause by calling a number of the DBMS_PIPE builtins, such as RECEIVE_MESSAGE and SEND_MESSAGE.

3. *What package can you use to determine if the current session has issued a COMMIT? How would you go about obtaining this information?*

   The DBMS_LOCK allows you to determine if a COMMIT has occurred. You use the REQUEST procedure to request a lock specifying TRUE for **`release_on_commit`**. Then later in your program you can request this same lock. If you can get the lock, a commit has been performed.

4. *What do you see when you execute the following statements in SQL*Plus (assuming that you have already called SET SERVEROUTPUT ON)?*

   I will show the output from each of these calls and then explain them afterwards.

   ```
   SQL> execute DBMS_OUTPUT.PUT_LINE (100);
   100
   SQL> execute DBMS_OUTPUT.PUT_LINE ('     Five spaces in');
   Five spaces in
   SQL> execute DBMS_OUTPUT.PUT_LINE (NULL);
   ERROR at line 1:
   ORA−06550: line 1, column 7:
   PLS−00307: too many declarations of 'PUT_LINE' match this call
   SQL> execute DBMS_OUTPUT.PUT_LINE (SYSDATE < SYSDATE − 5);
   ERROR at line 1:
   ORA−06550: line 1, column 7:
   PLS−00306: wrong number or types of arguments in call to 'PUT_LINE'
   SQL> execute DBMS_OUTPUT.PUT_LINE (TRANSLATE ('abc', 'a', NULL));
   SQL> execute DBMS_OUTPUT.PUT_LINE (RPAD ('abc', 500, 'def'));
   ERROR at line 1:
   ORA−20000: ORU−10028: line length overflow, limit of 255 bytes per line
   ```

   The first answer of "100" verifies that the DBMS_OUTPUT.PUT_LINE procedure (DOPL for short) puts a line of information to the screen or standard out from inside your PL/SQL program. In the second answer, you notice that the text is *not* five spaces in. That is because DOPL automatically LTRIMs your text on display in SQL*Plus. When I tried to display NULL, DBMS_OUTPUT could not figure out which of the overloaded versions of PUT_LINE to call because NULL does not have a datatype.

When I tried to display the value returned by SYSDATE < SYSDATE − 5, DOPL raised an exception because it is not overloaded for Boolean values.

When I tried to display the output from the TRANSLATE function, *nothing happened*! This non−event was caused by two factors: first, when you specify NULL for the replacement character set in TRANSLATE, that builtin returns NULL. Second, when you try to display a NULL string (which is different from the NULL literal) or blank line, DOPL simply ignores your request and does nothing.

When I attempted to display the string "abc" right−padded to a length of 500 with the string "def", I was reminded that DOPL cannot handle pieces of data with more than 255 bytes.

5. *When an error occurs in your program, you want to be able to see which program is currently executing. What builtin packaged function would you call to get this information? If the current program is a procedure named* **`calc_totals`** *in the analysis package, what would you see when you call the builtin function?*

   The DBMS_UTILITY.FORMAT_CALL_STACK returns the current execution stack in PL/SQL. If the current program is analysis.calc_totals, however, the string returned by FORMAT_CALL_STACK only tells you that you are executing analysis. It does not know which program *inside* the package you are running.

6. *You want to build a utility for DBAs that would allow them to create an index from within a PL/SQL program. Which package would you use? Which programs inside that package would be needed?*

   To perform SQL DDL inside PL/SQL, you use the DBMS_SQL package. With this package, you dynamically construct the string to create the index and then call the following elements of DBMS_SQL: OPEN_CURSOR to allocate memory for the dynamic SQL; PARSE to parse the statement; and CLOSE_CURSOR to close the cursor. Since you are working with DDL, a parse also executes and commits.

7. *You need to run a stored procedure named* **`update_data`** *every Sunday at 4 AM to perform a set of batch processes. Which builtin package would you use to perform this task?*

   You need to pass a string to the submit program to tell it how often to run **`update_data`**. What would that string be? The DBMS_JOB package allows you to queue up stored procedures to be executed on a regular or one−time basis. You would call the SUBMIT program of DBMS_JOB. When you call SUBMIT, you pass it a string (which will be executed as dynamic PL/SQL) defining the next time the program will be executed. SYSDATE stands for now and this is the string which means "every Sunday at 4 AM":

   ```
   'NEXT_DAY (TRUNC (SYSDATE), ''SUNDAY'') + 4/24'
   ```

A.2.7 Modules

1. *In each of the following modules, identify changes you would make to improve their structure, performance or functionality.*

   a.

Use single RETURN statement at end of function to return value. Put in assertion routine or other form of check to handle situation when invalid (unhandled) status code is passed to routine.

b.
Either remove the OUT parameter or change the function to a procedure.

c.
Do not use a RAISE statement to handle successful completion of function. Consider using explicit cursor rather than implicit cursor.

d.
Do not issue a RETURN from inside a loop. Also, do not issue a RETURN from inside a procedure.

2.
Given the header for **`calc_profit`** *below, which of the following calls to* **`calc_profit`** *are valid:*

Call to calc_profit	Good/Bad? Why?

`calc_profit`
`(1005, profit_level, 1995, 'ALL', 'FINANCE');`
	Great

`calc_profit`
`(new_company, profit_level);`
	Bad. Must supply value for fiscal year.

`calc_profit`
`(company_id_in => 32, fiscal_year_in => 1995,`
`profit_out => big_number);`
	Good. All three mandatory params are present, even if not in right order.

`calc_profit`
`(company_id_in => 32, division_in => 'ACCTG',`
`profit_out => 1000);`
	Bad. The actual **`profit_out`** argument must be a variable.

3.
Suppose you had to use **`dispdates`** *to satisfy the following requirement: "Display the list of company start dates stored in the date table without any header." I can think of two ways do this:*

```
dispdates (company_list, num_companies);
```

and

```
dispdates (company_list, num_companies, NULL);
```

Which of these implementations would you choose and why? Is there any reason to choose one over the other?

It would be tempting to take the first approach. It is less typing. The second form is, however, the correct solution. The reason is this: you were asked to display a list without a header −− with, in other words, a NULL header. You were not asked to display a list with the default header. If you were asked to use the default value, then you can and should simply rely on the default. If you were asked to skip the header, then you should explicitly request a NULL header when you call dispdates. That way, if the default value ever changes, your code is not affected and the format of the display does not change.

A.2.8 Module Evaluation: Foreign Key Lookup

You were asked to evaluate a function that performs a foreign key lookup. My evaluation follows. You will undoubtedly have found other problems as well.

There are many, many problems with the **`getkey_clrtyp`** function:

- It is ugly. Everything is in uppercase, which makes the code hard to read. Indentation in the body of the function is inconsistent. All the code between the BEGIN and END statements should be indented. Then within an IF statement, all code should be indented another level. It indents unevenly and also uses different formats for IF statement indentation within this single program. This poor formatting makes it even more difficult to understand the logical flow of the program.

- There are side−effects with IN OUT parameters. A function should never have OUT or IN OUT parameters. The point of a function is to return a value through its RETURN clause. If you need to return multiple values, you can either return a composite data structure (a PL/SQL table or record) or change the function to a procedure.

- The **`nu_inout`** argument does not need to be IN OUT, only OUT. The **`nu_inout`** argument is only referenced on the left side of an assignment operator. It is, therefore, simply an OUT parameter.

- No value is returned if **`name_inout`** is NULL. If the **`name_inout`** argument is NULL, then this function never executes a RETURN statement. This raises a runtime error and is a fatal flaw in a function's design. You should instead use an approach in which the last line of your function issues the single RETURN for that function.

- The cursor is not closed if no record is found. This is a stylistic as opposed to functional weakness. The cursor *will* be closed when the function terminates, since it is declared and opened locally. If the cursor were based in a package, on the other hand, it would stay open until closed explicitly when the session ends. Always closing cursors is a good habit to develop −− and you can't go wrong.

- "Magic values" of 0, 1, and 2 are used as return values of the function. The return values of this function are obscure and poorly designed. There is no way to know by simply glancing at the function what these return values signify. It would be even more difficult for a user of the function to use **`getkey_clrtyp`** properly simply by looking at the header of the function. You should always avoid these kinds of "magic values" and literals in your code. If specific values have special meanings, you are much better off defining these as constants in a package and then referencing those constants both inside and outside the function (see the recoding of the function below for an example of this approach).

- The function name does not describe the value returned. The name **`getkey_clrtyp`** describes in the most general terms the objective of the function, but it in no way indicates what kind of value is being returned by the function. Since a function encapsulates a returned value, the name of the function should describe the value.

- There are multiple RETURN statements. A very fundamental rule for structured programming is that there should be one way in to a program and one way out of the program. In PL/SQL terms, this means that you should only have one RETURN statement in the executable section of the function. When you have more than one RETURN, the code is more difficult to understand, debug and enhance.[1]

> [1] You should also have a RETURN statement for each exception handler, but that is a separate issue and in no way contradicts this structured programming rule.

- There is an unused variable. I declare the **`typ_nu`** variable, but then never use it in the program. You are much better off without such clutter.

Did you find any other problems? I would not be the least bit surprised. Every time I have gone over this program in a class, the students have uncovered additional areas for improvement.

A.2.8.1 A rewrite of the getkey_clrtyp function

The following version of **`getkey_clrtyp`** incorporates many of the comments in the previous section. Notice that it is no longer even a function; I have changed it to a procedure so that I can take in and return as many values as needed.

```
PROCEDURE getkey_clrtyp
   (name_inout IN OUT VARCHAR2,
     nu_inout IN OUT NUMBER,
     get_status_out OUT INTEGER)
IS
   CURSOR clrtyp_cur IS
      SELECT typ_nu, type_ds
        FROM caller_type
       WHERE type_ds LIKE name_inout || '%';

   clrtyp_rec clrtyp_cur%ROWTYPE;
   next_rec clrtyp_cur%ROWTYPE;
   retval NUMBER := NULL;
BEGIN
   IF name_inout IS NULL
   THEN
      get_status_out := get.nullname;

   ELSE

      OPEN clrtyp_cur; FETCH ...;
      IF clrtyp_cur%NOTFOUND
      THEN
         get_status_out := get.notfound;
      ELSE
         FETCH clrtyp_cur INTO next_rec;
         IF clrtyp_cur%NOTFOUND
         THEN
            get_status_out := get.unique_match;
         ELSE
            get_status_out := get.dup_match;
         END IF;
         nu_inout := clrtyp_rec.cllr_typ_nu;
         name_inout := clrtyp_rec.cllr_typ_ds;
      END IF;
      CLOSE clrtyp_cur;
   END IF;
END getkey_clrtyp;
```

PREVIOUS HOME

A.1 Exercises BOOK INDEX

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8*i* Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

1. PL/SQL Packages

Contents:

1.1 What Is a PL/SQL Package?

A package is a collection of PL/SQL elements that are "packaged" or grouped together within a special BEGIN−END syntax, a kind of "meta−block." Here is a partial list of the kinds of elements you can place in a package:

- Cursors
- Variables (scalars, records, tables, etc.) and constants
- Exception names and pragmas for associating an error number with an exception
- PL/SQL table and record TYPE statements
- Procedures and functions

Packages are among the least understood and most underutilized features of PL/SQL. That's a shame because the package structure is also one of the most useful constructs for building well−designed PL/SQL−based applications. Packages provide a structure to organize your modules and other PL/SQL elements. They encourage proper structured programming techniques in an environment that often befuddles the implementation of structured programming. When you place a program unit into a package you automatically create a "context" for that program. By collecting related PL/SQL elements in a package, you express that relationship *in the very structure of the code itself*. Packages are often called "the poor man's objects" because they support some, but not all, object−oriented rules.

The PL/SQL package is a deceptively simple, yet powerful construct. It consists of up to two distinct parts: the specification and the body.

- The *package specification*, which defines the public interface (API) of the package: those elements that can be referenced outside of the package.
- The *package body*, which contains the implementation of the package and elements of the package you want to keep hidden from view.

In just a few hours you can learn the basic elements of package syntax and rules; there's not all that much to it. You can spend weeks and months, however, uncovering all the nuances and implications of the package structure.

Oracle Corporation itself uses the package construct to define and extend the PL/SQL language. In fact, the most basic operators of the PL/SQL language, such as the + and LIKE operators and the INSTR function, are all defined in a special package called STANDARD.[1] Packages will, without doubt, be the preferred method of delivering new functionality in PL/SQL in the coming decade. Just consider PL/SQL packages in the Oracle Web Agent: these add−ons provide a powerful interface between World Wide Web pages/HTML and the Oracle database, allowing you to construct Oracle−aware WWW pages more easily.

[1] If Oracle believes that packages are the way to go when it comes to building both fundamental and complex programs, don't you think that you could benefit from the same?

PREVIOUS | HOME | NEXT
I. Working With Packages | BOOK INDEX | 1.2 What Are the Types and Layers of Packages?

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8*i* Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

Copyright (c) 2000 O'Reilly Associates. All rights reserved.

PREVIOUS Chapter 1
PL/SQL Packages NEXT

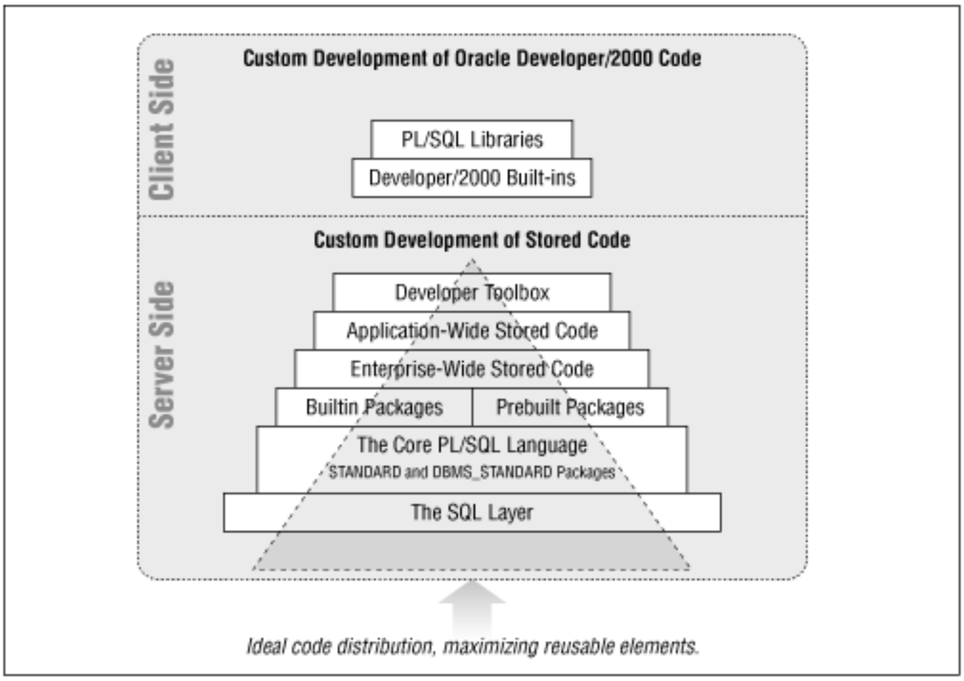
1.2 What Are the Types and Layers of Packages?

Although any PL/SQL package must have the same structure and follow the same rules, there are different types of packages that will play different roles in your application.

Types of Package	Description
Builtin	A builtin package is provided to you by Oracle Corporation, built right into the PL/SQL language as installed. (A builtin package could reside in the database as stored code or could instead be imbedded in a client tool, such as Oracle Developer/2000.)
Prebuilt	Prebuilt packages are libraries of packages that are built by third parties or other developers that are installed in and used by your PL/SQL environment.
Build−Your−Own (BYO)	The very best kind of package: the one you build yourself, or is built by your application development team.

Figure 1.1 shows how these different types of packages interact as layers of PL/SQL code and packages that are available to you. We describe the types further in "Types of Packages" later in this chapter.

Figure 1.1: The layers of PL/SQL code and packages


Client Side
Custom Development of Oracle Developer/2000 Code
PL/SQL Libraries
Developer/2000 Built-ins
Server Side
Custom Development of Stored Code
Developer Toolbox
Application-Wide Stored Code
Enterprise-Wide Stored Code
Builtin Packages
Prebuilt Packages
The Core PL/SQL Language
STANDARD and DBMS_STANDARD Packages
The SQL Layer
Ideal code distribution, maximizing reusable elements.

The lowest layer of code upon which you build your PL/SQL applications is the SQL language. PL/SQL was designed explicitly as a procedural language extension to SQL; most of your PL/SQL programs will function as an interface between a user of some kind and the database.

The next layer of code is the core PL/SQL language. At the center of the PL/SQL universe, we have the STANDARD and DBMS_STANDARD packages, which define the basic elements of the language. When you execute the TO_CHAR function or even use the LIKE operator, you are actually calling elements of the STANDARD package. Since these two packages are the default in PL/SQL, however, you do not have to explicitly reference the package name.

Many PL/SQL developers make use of only these two layers of code, but as you can tell from Figure 1.1, there is much more for you to take advantage of in PL/SQL, in terms of both builtin functionality and code reusability. Oracle Corporation provides a vast suite of builtin packages which extend the PL/SQL language itself and which, from a layering standpoint, sit directly above the core language elements. These packages have either the DBMS_ or UTIL_ prefix on their names. At the same level −− because they are used in precisely the same manner −− are prebuilt packages (and other kinds of prebuilt stored code, such as procedures and functions). PL/Vision is an example of prebuilt code: a library of packages. Once they have been installed and access has been granted to them, all of these packages are available for your use.

The next three layers of code represent the different kinds of reusable code you can and should build into your own environment. The enterprise−wide stored code is very similar to prebuilts: packages and other program units that are shared throughout an entire enterprise (i.e., the corporation). The application−wide stored code is the body of PL/SQL programs that are reused throughout a particular application. The developer's toolbox is the individual developer's set of code that he uses to enhance his own development efforts.

The enterprise−wide, application−wide, and developer toolbox packages are all examples of "build−your−own" packages.

All of those different, reusable layers of code (colored in white) exist to help developers build their custom applications as rapidly as possible. In Figure 1.1, the custom code is represented by the gray areas. Notice that the gray areas are in contact with all layers of the reusable code. This makes sense because your own code will undoubtedly include SQL statements, calls to the builtin functions and PL/SQL operators, and so on, right up through the layers. While it is important to be able to access those lower levels of code, however, you should always leverage programs from the *highest level* possible. This principle is illustrated by the dashed−line triangle in the figure; I call it the "iceberg" approach to writing PL/SQL code.

1.2.1 The Iceberg Approach to Coding

The tip of the iceberg appears in the custom code section. This portion of the triangle represents our own program, for example, procedure **`calc`**, which performs calculations for an order entry system. Only a small portion of the triangle resides in the custom code area. That is because it makes use of all of the other layers of reusable PL/SQL code, keeping the custom code to an absolute minimum. The triangle broadens as it descends through the layers because each higher−level program typically utilizes many elements in the layers below it.

If you take fullest possible advantage of the many layers of reusable code in PL/SQL, your own custom programs will resemble an iceberg floating heavily in the sea. A person looking at your code will wonder at its brevity and clarity. He or she will wonder: how does this little program get the job done? The answer is that the custom program is just the tip of the iceberg. The visible portion is just a hint at the bulk of code below the surface, powerful and solid.

1.2.2 The Client−Side Layers

There are also a few layers on the client side of the equation. If you also use Oracle Developer/2000, you can also take advantage of the fact that PL/SQL is available as a client−side language inside that tool suite. (The PL/SQL of Oracle Developer/2000 Release 1 is, however, only PL/SQL Release 1.1 and this can cause many complications. See *Oracle PL/SQL Programming, Appendix B*, for more information. Release 2 of Oracle Developer/2000 will support PL/SQL, simplifying all of our lives greatly.) In this case, you can make use of call programs from the builtin packages of Oracle Developer/2000. These packages offer access to functionality specific to the client−side environment, such as OLE and DDE.

With Oracle Developer/2000 you can also construct PL/SQL *libraries*, which can be shared across different Oracle Developer/2000 modules. Finally, you can also build your own client−side packages. These BYO client−side packages can execute code from an Oracle Developer/2000 builtin package, a BYO server−side package, prebuilt packages like PL/Vision, and server−side builtin packages.

Now, that is a lot of code to choose from. Of course, you have to be able to figure out what is available and how to use it −− and that is just the kind of challenge PL/Vision tries to address for you.

Before diving into PL/Vision, though, Part I will bring you up to speed on the structure, features, and best practices of PL/SQL packages.

PREVIOUS
1.1 What Is a PL/SQL Package?

HOME
BOOK INDEX

NEXT
1.3 What Are the Benefits of Packages?

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8*i* Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

1.3 What Are the Benefits of Packages?

Before exploring the architecture of packages and how best to build them, let's look at some of the most important benefits of the package.

1.3.1 Enforced Information Hiding

When you build a package, you decide which of the package elements are public (can be referenced outside of the package) and which are private (available only within the package itself). You also can restrict access to the package to only the specification. In this way, you use the package to hide the implementation details of your programs. This is most important when you want to isolate the most volatile aspects of your application, such as platform dependencies, frequently changing data structures, and temporary workarounds.

1.3.2 Object−Oriented Design

While PL/SQL does not yet offer full object−oriented capabilities, packages do offer the ability to follow many object−oriented design principles. The package gives developers very tight control over how the modules and data structures inside the package can be accessed.

You can, therefore, embed all the rules about your entities (whether they are database tables or memory−based structures), and access to them, in the package. Since this is the only way to work with that entity, you have in essence created an abstracted and encapsulated object.

1.3.3 Top−Down Design

A package's specification can be written before its body. You can, in other words, design the *interface* to the code hidden in the package (the modules, their names, and their parameters) before you have actually implemented the modules themselves. This feature dovetails nicely with top−down design, in which you move from high−level requirements to functional decompositions to module calls.

Of course, you can design the names of standalone modules just as you can the names of packages and their modules. The big difference with the package specification is that you can compile it even without its body or implementation. Furthermore and most remarkably, programs that call packaged modules also compile successfully −− even if nothing more than the specification has been defined.

1.3.4 Object Persistence

PL/SQL packages offer the ability to implement *global data* in your application environment. Global data is information that persists across application components; it isn't just local to the current module. If you have designed screens with SQL*Forms or Oracle Forms, you are probably familiar with its GLOBAL variables, which allow you to pass information between screens. Those globals have many limitations (e.g., GLOBAL variables are always represented as fixed−length CHAR variables with a length of 254), but they sure can be useful. Package−based data gets around all these limitations.

Objects declared in a package specification (that is, visible to anyone with execute authority on that package) act as global data for all PL/SQL objects in the application. If you have access to the package, you can modify package variables in one module and then reference those changed variables in another module. This data persists for the duration of a user session (connection to the database).

And your global data doesn't consist merely of scalar data like numbers. If, for example, a packaged procedure opens a cursor, that cursor remains open and is available to other packaged routines throughout the session. You do not have to explicitly define the cursor in each program. You can open it in one module and fetch it in another module.

Finally, package variables can carry data across the boundaries of transactions, since they are tied to the session itself and not to a transaction.

1.3.5 Guaranteeing Transaction Integrity

The RDBMs and SQL language give you the ability to tightly control access to, and changes in, any particular table. With the GRANT command you can, for example, make sure that only certain roles and users have the ability to perform an UPDATE on a given table. But this GRANT statement cannot make sure that the UPDATEs performed by a user or application that affect multiple tables conform to all complex business rules.

In a typical banking transaction, for example, you might need to transfer funds from account A to account B. The balance of account B must be incremented, and that of account A decremented. Table access is necessary, but not sufficient, to guarantee that both of these steps are always performed by all programmers who write code to perform a transfer. With stored code in general, and packages in particular, you can guarantee that a funds transfer either completes successfully or is completely rolled back −− regardless of who executes the process.

The secret to achieving this level of transaction integrity is the *execute authority* concept. Instead of granting the authority to update a table to a role or user, you grant privileges to that role/user only to *execute a procedure*. This procedure controls and provides access to the underlying data structures. The procedure is owned by a separate Oracle RDBMs account, which, in turn, is granted the actual update privileges on those tables needed to perform the transaction. The procedure therefore becomes the gatekeeper for the transfer transaction. The only way a program (whether it is an Oracle Forms application or a Pro*C executable) can execute the transfer is through the procedure, thus guaranteeing transaction integrity.

1.3.6 Performance Improvement

When an object in a package is referenced for the first time, the entire package (already compiled and validated) is loaded into memory (the System Global Area, or SGA, of the RDBMs). All other package elements are thereby made immediately available for future calls to the package. The PL/SQL runtime engine does not have to keep retrieving program elements or data from disk each time a new object is referenced.

This feature is especially important in a distributed execution environment. You may reference packages from different databases across a local−area or even a wide−area network. You want to minimize the network traffic involved in executing your code.

Packages also offer performance advantages on the development side, with a potential impact on overall database performance. The Oracle RDBMs automatically tracks the validity of all program objects (procedures, functions, packages) stored in the database. It determines what other objects that program is dependent on, such as tables. If a dependent object changes (for example, a table's structure changes), then all programs that rely on that object are flagged as invalid. The database then automatically recompiles these invalid programs when they are referenced next.

You can limit the need for recompiles by placing functions and procedures inside packages. If program A calls packaged module B, it does so through the package's specification. As long as the specification of a packaged module does not change, any program that calls the module is not flagged as invalid and will not have to be recompiled.

This brief review of the benefits of packages should help focus your interest on this fascinating and powerful element of the PL/SQL language.

PREVIOUS | HOME | NEXT
1.2 What Are the Types and Layers of Packages? | BOOK INDEX | 1.4 Using Packages

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8*i* Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

1.4 Using Packages

Whether you are referencing an element in a builtin package, prebuilt package, or build−your−own package, the syntax is the same. One thing to remember is that a package itself is not any kind of executable piece of code. Instead, it is a repository for code that is executed or otherwise used. When you use a package, you actually execute or make reference to an element *in* a package. To use a package you must know what is defined and available inside the package. This information is contained in the package specification.

1.4.1 The Package Specification

The package specification contains the definition or specification of all elements in the package that may be referenced outside of the package. These are called the *public* elements of the package. Here is a very simple package specification consisting of two procedures:

```
PACKAGE sp_timer
IS
   PROCEDURE capture;
   PROCEDURE show_elapsed;
END sp_timer;
```

(The **`sp_timer`** package is an early version of the PLVtmr package, used to time PL/SQL code execution.) What do you learn by looking at this specification? That you can call either the **`capture`** or the **`show_elapsed`** procedures of **`sp_timer`** −− and that is it.

The package specification contains all the code needed for a developer to understand how to call the objects in the package. A developer should never have to examine the code behind the specification (which is the body) in order to understand how to use and benefit from the package.

Here is a more generic representation of the syntax for a package specification:

```
PACKAGE package_name
IS
   [ declarations of variables and types ]
   [ headers of cursors ]
   [ headers of procedures and functions ]
END [ package_name ];
```

You can declare variables and include headers of both cursors and modules (and only the specifications). You must have at least one declaration or header statement in the package specification.

Notice that the package specification has its own BEGIN−END block syntax. This establishes a named context for all the elements of the package and allows them to exist outside of any particular PL/SQL block, such as a procedure or function.

Now let's take a look at a more complex package specification and use that as a springboard to learn how to execute and reference package−based PL/SQL code. Example 1.1 shows the specification for the **`pets_r_us`** package, which is used by veterinarians to keep track of their patients and to determine when a pet needs another visit.

Example 1.1: The specification of the pets_r_us package

```
PACKAGE pets_r_us
IS
   max_pets_in_facility CONSTANT INTEGER := 120;
   pet_is_sick EXCEPTION;
   next_appointment DATE := SYSDATE;
   CURSOR onepet_cur (pet_id_in IN INTEGER) RETURN pet%ROWTYPE;
   CURSOR allpets_cur IS SELECT pet_id, name, owner FROM pet;
   FUNCTION next_pet_shots (pet_id_in IN NUMBER) RETURN DATE;
   PROCEDURE set_schedule (pet_id_in IN NUMBER);
END pets_r_us;
```

The **`pets_r_us`** package specification shown in Example 1.1 declares a constant, an exception, a variable, two cursors, a function, and a procedure. The constant informs us that the package restricts the number of pets allowed in the facility to 120. The **`pets_r_us`** package also provides an exception that can be used throughout the application to indicate that a pet is sick. It offers a predefined variable to hold the date of the next appointment and initializes that variable to today's date.

The code in this package might look odd to you; only the headers are present for the function and procedure. The executable code for these modules is, in fact, hidden in the package body (explored later in this chapter). A package specification never contains executable statements; you should not have to see this code in order to understand how to call the program.

Notice the difference between the two cursors. The first cursor, **`onepet_cur`**, takes a single parameter (primary key for a pet) and returns a record with the same structure as the pet table. The SELECT statement for this query is not, however, present. Instead, the query is hidden in the package body (the SQL is, after all, the implementation of the cursor) and the RETURN clause is used. In the second cursor, the RETURN clause is replaced by the actual query for the cursor. You can take either approach to cursors in packages.

1.4.2 Referencing Package Elements

A package owns its elements, just as a table owns its columns. An individual element of a package only makes sense, in fact, in the context of the package. Consequently, you use the same dot notation employed in "table.column" syntax for "package.element". Let's take a look at this practice by calling elements of the **`pets_r_us`** package.

In the following IF statement, I check to see if I am allowed to handle more than 100 pets in the facility:

```
IF pets_r_us.max_pets_in_facility > 100
   THEN
      ...
   END IF;
```

In this exception section, I check for and handle the situation of a sick pet:

```
EXCEPTION
   WHEN pets_r_us.pet_is_sick
   THEN
      ...
END;
```

I can open the cursor defined in a package by prefixing the package name to the cursor and passing any required arguments:

```
OPEN pets_r_us.onepet_cur (1305);
```

In the following statement, I assign (to the package variable **`next_appointment`**) the date for the next shot for a pet identified by an Oracle Forms host variable (indicated by the use of the **:** before the **block.item** name):

```
:pets_r_us.next_appointment
      := pets_r_us.next_pet_shots (:pet_master.pet_id);
```

And if you forget to qualify a package element with its package name? The compiler will try to find an unqualified element (table, standalone procedure, etc.) with the same name and characteristics. Failing that, your code will not compile.

1.4.2.1 Unqualified package references

There is one exception to the rule of qualifying a package element with its package name. Inside the body of a package, you do not need to qualify references to other elements of that package. PL/SQL will automatically resolve your reference within the scope of the package; the package is the "current" context. Suppose, for example, that the **set_schedule** procedure of **pets_r_us** (defined in the package specification) references the **max_pets_in_facility** constant. Such a reference would be *unqualified*, as shown below in the partial implementation of **set_schedule** (found in the package body):

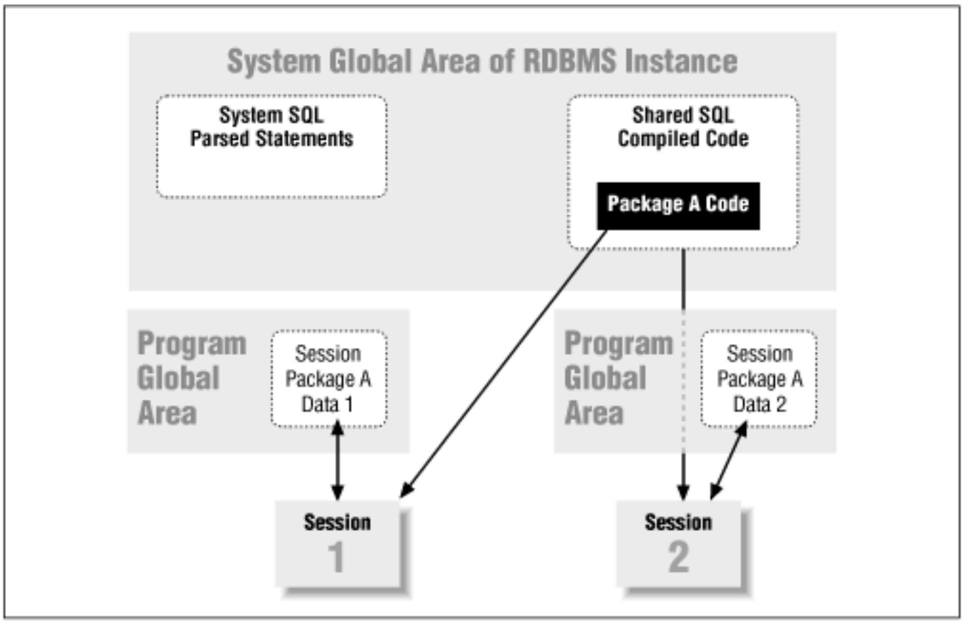
```
PROCEDURE set_schedule (pet_id_in IN NUMBER)
IS
   total_pets NUMBER := pet_analysis.current_load;
BEGIN
   ...
   IF total_pets < max_pets_in_facility
   THEN
      ...
   END IF;
END;
```

There is no need to preface the "maximum pets" constant with **pets_r_us**. There is a need, on the other hand, to prefix the reference to the **current_load** function of the **pet_analysis** package.

1.4.3 The Memory−Resident Architecture of Packages

To use packages most effectively, you must understand the architecture of these constructs within an Oracle Server instance. Figure 1.2 shows how the different elements of shared memory are employed to support both package code and data.

Figure 1.2: The architecture of packages in shared memory



Before exploring the relationships in Figure 1.2, keep these basic principles in mind:

- The compiled code for stored objects (procedures, functions, and packages) is shared by all users of the instance with execute authority on that code.
- Each Oracle session has its own copy of the in−memory data defined within stored objects.
- The Oracle Server applies a least−recently used (LRU) algorithm to maintaining compiled code in shared memory.

When a user executes a stored program or references a package−based data structure, the PL/SQL runtime engine first must make sure that the compiled version of that code is available in the System Global Area or SGA of the Oracle instance. If the code is present in the SGA, it is then executed for that user. If the code is not present, the Oracle Server reads the compiled code from disk and loads it into the shared memory area. At that point the code is available to all users with execute authority on that code.

So if session 1 is the first account in the Oracle instance to reference package A, session 1 will cause the compiled code for A to be loaded into shared memory. When session 2 references an element in package A, that code is already present in shared memory and is re−used.

A user's relationship to data structures defined in stored code, particularly package data, is very different from that of the compiled code. While the same compiled code is shared, each user gets her own version of the data. This process is clear for procedures and functions. Any data declared in the declaration section of these programs is instantiated, manipulated, and then, on the termination of that program, erased. Every time a user calls that procedure or function, she gets her own local versions of the data.

The situation with packages is the same as that with stored code, but is less obvious at first glance. Data declared at the package level (defined outside of any particular procedure or function in the package) persist for as long as a session is active −− but those data are specific to a single Oracle session or connection. Each Oracle session is assigned its own private PL/SQL area, which contains a copy of the package data. This private PL/SQL area is maintained by the PL/SQL runtime engine for as long as your session is running. When session 1 references package A, session 1 instantiates her own version of the data structures used by A. When session 2 calls a program in A or accesses a data structure defined by A, session 2 gets her own copy of that data. Any changes made to the memory−based package data in session 1 is not affected by and does not affect the data in session 2.

> ***NOTE:*** If you are running a multithreaded Oracle Server, then Figure 1.2 changes slightly. With the multithreaded architecture, the program global areas for each user are also stored within the SGA of the Oracle instance.

1.4.3.1 Managing packages in shared memory

When a package is loaded into shared memory, a contiguous amount of memory is required to hold the package (the same is true for any piece of stored code). So if you have a large package, you may have to tune your shared pool in the SGA to accommodate this package. (The shared pool is the area in the SGA devoted to shared SQL and PL/SQL statements.) You can get more space for your stored code by increasing the value of the SHARED_POOL_SIZE parameter in the database instance initialization file.[2]

[2] If the Oracle Server is having trouble fitting your stored code into memory, you will get ORA−04031 errors: out of shared memory.

The Oracle Server uses a least−recently used (LRU) algorithm to decide which items in the shared pool will remain present. If your package is flushed out of memory and is then needed by another program, the compiled code of the package will have to be read again from disk. Contiguous memory will also need to be available at that point.

If you know that you will want to use a large package or standalone program intermittently throughout application execution and do not want to have the code flushed out of memory, you can use the DBMS_SHARED_POOL package to pin your code into memory. The KEEP procedure of this package exempts the specified program or package from the LRU algorithm.

To pin the **`config`** package into shared memory, for example, you would execute this statement:

```
DBMS_SHARED_POOL.KEEP ('config');
```

You can also *unpin* a program with the UNKEEP program. The DBMS_SHARED_POOL package is not installed by default when you create an Oracle Server instance. You will need to execute (usually from within the SYS account) the *dbmspool.sql* script in the *admin* subdirectory of your particular version of the server. For example, on Windows95 and Oracle 7.2, you would issue this command in SQL*Plus:

```
SQL> @c:\orawin95\rdbms72\admin\dbmspool
```

You should only pin programs if absolutely necessary and unavoidable (you cannot, for instance, further expand the overall size of the SGA and the shared pool). Why? In answer, I quote from the above−mentioned *dbmspool.sql* file about KEEP:

```
--WARNING:  This procedure may not be supported in the future when
--and if automatic mechanisms are implemented to make this unnecessary.
```

You can calculate the size of a package or any other piece of stored code by executing queries against the USER_OBJECT_SIZE data dictionary view. This view contains information about the size of the source code, the size of the parsed code, and the size of the compiled code. The SQL statement below will display the names and sizes for any stored code larger than the specified SQL*Plus parameter:

```
SELECT name, type, source_size, parsed_size, code_size
  FROM user_object_size
 WHERE code_size > &1
 ORDER BY code_size DESC
/
```

1.4.4 Access to Package Elements

One of the most valuable aspects of a package is its ability to truly enforce information hiding. With a package you can not only modularize your secrets behind a procedural interface, you can keep those parts of your application completely private.

An element of a package, whether it is a variable or a module, can be either public or private:

Public

An element is public if it is defined in the specification. A public element can be referenced directly from other programs and PL/SQL blocks. The package specification is, in a sense, the gatekeeper for the package. It determines the package elements to which a developer may have access.

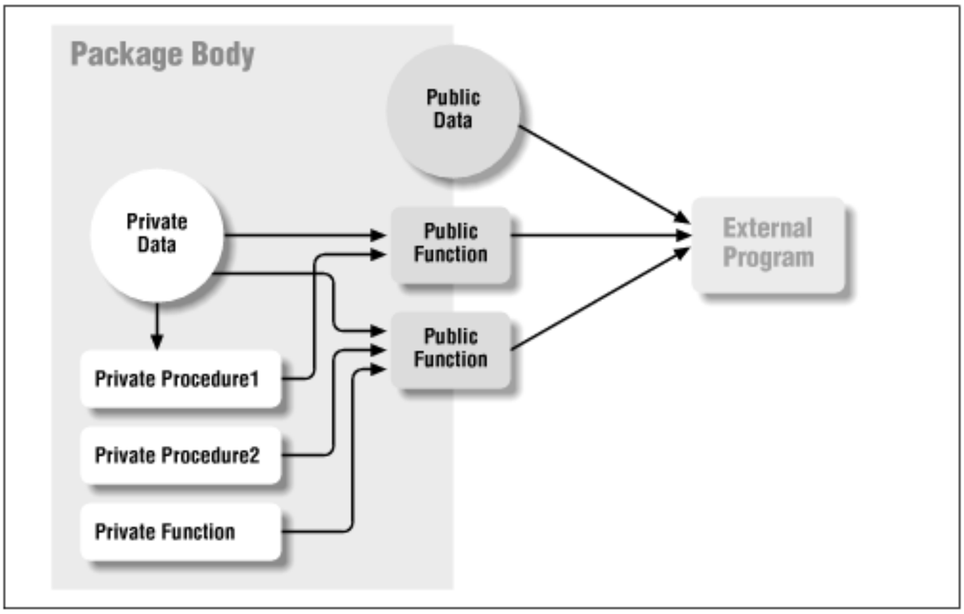
Private

An element is private if it is defined only in the body of the package, but does not appear in the specification. A private element cannot be referenced outside of the package. Any other element of the package may, however, reference and use a private element.

The distinction between public and private elements gives PL/SQL developers unprecedented control over their data structures and programs. Figure 1.3 shows a Booch[3] diagram for the package that displays private and public package elements, and very neatly portrays the way these two kinds of elements interact.

[3] This diagram is named after Grady Booch, who pioneered many of the ideas of the package, particularly in the context of object−oriented design.

Figure 1.3: A Booch diagram for a package



In Figure 1.3, all of the boxes that lie completely inside the box are private elements, defined only within the body of the package. Boxes that lie on the boundary of the box are public elements, defined in the package specification and implemented (if programs) in the package body. An external program can make direct references only to those package elements that lie on the boundary. But any package element, whether wholly inside the boundary or straddling that line, can reference any other package element.

A boundary in the package delineates that which is publicly available and that which is private or hidden from view. It has particularly important and valuable consequences for data structures defined in a package.

1.4.4.1 Public and private data

Whether a variable is declared in the specification or body, it *does* function as a global piece of data. Once the package is instantiated in your session, data declared in the package persist for the duration of the session. A variable will retain its value until it is changed. That value will be available to any program that has access to the data. The kind of access depends on whether the variable is defined in the package specification or in the body.

To understand the consequences of public (specification−declared) data and private (body−declared) data in packages, consider the following simple package. In **`downsize`**, **`hire_date`** is a public variable and **`fire_date`** is a private variable.

```
PACKAGE downsize
IS
   v_hire_date DATE;
END;

PACKAGE BODY downsize
IS
   v_fire_date DATE;
END;
```

Since **`v_hire_date`** is defined in the package specification, I can directly reference that variable in my own code outside of the *downsize* package, as follows:

1. Read the value of the **`hire_date`** variable:

   ```
   last_hired := downsize.v_hire_date;
   ```

2. Change the value of the **`hire_date`** variable to ten days in the future:

   ```
   downsize.v_hire_date := SYSDATE + 10;
   ```

If I try to access **`v_fire_date`** in the same way, however, my code will not compile. It is hidden behind the public boundary of the package. Its value is maintained in my private global area since it is in a package, but the only programs that can reference it are those defined within the package itself, either in the body or the specification.

The next chapter covers the implications of public global data and contains recommendations on how to safeguard your application and data.

1.4.4.2 Switching between public and private

When you first create a package, your decision about which elements of a package are public and which private is not cast in stone. You can, in fact, switch a public element to private and vice versa at any time.

If you find that a private element program or cursor should instead be made public, simply add the header of that element to the package specification and recompile. It will then be visible outside of the package. Notice that you do not need to make any changes at all to the package body.

If you want to make a private variable accessible directly from outside the package, you will need to remove the declaration of that data structure from the body and paste it into the specification. You cannot declare the same element in both the body and the specification.

If you do make a public element private, you will need to remember that any program that referenced that element will no longer compile successfully.

PREVIOUS | HOME | NEXT
1.3 What Are the Benefits of Packages? | BOOK INDEX | 1.5 Types of Packages

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8*i* Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

1.5 Types of Packages

The different types of packages are determined by who wrote them and by where they lay in the layers of PL/SQL code. As we mentioned earlier, the lowest−level and therefore most broadly available packages are the builtin packages, provided by Oracle Corporation. The next level of packages are the prebuilt packages, written by a third party and made available to you for inclusion in your applications. Finally, there are packages you build yourself.

1.5.1 Builtin Packages

Table 1.1 shows a partial list of builtin packages provided by Oracle Corporation. Unless otherwise noted, these packages are available in PL/SQL Release 2.1 and beyond. Most of these packages are installed by default when you create a database instance. In some cases, you may have to grant execute privileges on specific packages (such as DBMS_LOCK and DBMS_SQL) in order to make them available to your user community.

Table 1.1: Some of the Builtin Packages Stored in the Oracle Database

Package Name	Description
DBMS_ALERT	Provides support for notification of database events on an asynchronous basis. Registers a process with an alert and then waits for a signal from that alert.
DBMS_DDL	Provides a programmatic access to some of the SQL DDL statements.
DBMS_JOB	Used to submit and manage regularly scheduled jobs for execution inside the database.
DBMS_LOCK	Allows users to create their own locks using the Oracle Lock Management (OLM) services in the database.
DBMS_MAIL	Offers an interface to Oracle Office (previously known as Oracle Mail).
DBMS_OUTPUT	Displays output from PL/SQL programs to the terminal. The "lowest common denominator" debugger mechanism for PL/SQL code.
DBMS_PIPE	Allows communication between different Oracle sessions through a pipe in the RDBMs shared memory. One of the few ways to share memory−resident data between Oracle sessions.
DBMS_SESSION	Provides a programmatic interface to several SQL ALTER SESSION commands and other session−level commands.
DBMS_SNAPSHOT	A programmatic interface through which you can manage snapshots and purge snapshot logs. You might use modules in this package to build scripts to automate maintenance of snapshots.
DBMS_SQL	Full support for dynamic SQL within PL/SQL. Dynamic SQL means SQL statements that are not prewritten into your programs. They are, instead,

	constructed at runtime as character strings and then passed to the SQL Engine for execution. (PL/SQL Release 2.1 only)
DBMS_TRANSACTION	A programmatic interface to a number of the SQL transaction statements, such as the SET TRANSACTION command.
DBMS_UTILITY	The "miscellaneous" package. Contains various useful utilities, such as GET_TIME, which calculates elapsed time to the hundredth of a second, and FORMAT_CALL_STACK, which returns the current execution stack in the PL/SQL runtime engine.
UTL_FILE	Allows PL/SQL programs to read from and write to operating system files. (PL/SQL Release 2.3 only)

All of the packages in Table 1.1 are stored in the database and can be executed by both client and server−based PL/SQL programs. In addition to these packages, many of the development tools, like Oracle Forms, offer their own specific package extensions as well, such as packages to manage OLE2 objects and DDE communication.

It is no longer sufficient for a developer to become familiar simply with the basic PL/SQL functions like TO_CHAR and ROUND. Those functions have now become simply the inner layer of useful functionality. Oracle Corporation has built upon them, and you should do the same. (To take full advantage of the Oracle technology as it blasts its way to the 21st century, you must be aware of these packages and how they can help you.)

Builtin packages can and should revolutionize the code you write. With the last few releases of the Oracle Corporation's CDE tools, Oracle Server, and PL/SQL itself, the software vendor has shifted course. As Oracle developed the code it needed to implement new features, it no longer hid that code from the rest of the world. Instead, Oracle has exposed that code −− invariably structured as one or more packages −− so that all developers can also take advantage of those same techniques it employs. The next section gives you an example of this process.

1.5.1.1 Leveraging builtin packages

Oracle Corporation called the Oracle7 Server Version 7.1 the "Parallel Everything" server. It offered parallel query, parallel index updates, and many other features that take advantage of the symmetric multiprocessors readily available today. The parallelization of the RDBMs is an important advance in raw performance, but Oracle Corporation didn't stop there. It also "made public" (i.e., available to outside developers) a package of procedures and functions −− DBMS_PIPE −− used by its developers to support these parallel operations.

The DBMS_PIPE package provides a means to communicate between different Oracle processes directly through the SGA, outside of any particular data transaction. When the RDBMs receives a query request, it can determine whether any of the individual components of the query can be processed independently. If so, the RDBMs issues a call to DBMS_PIPE.SEND_MESSAGE to send the various query components to waiting processes in order to execute those chunks of SQL −− simultaneously.

Now here's the really exciting part: the advantages of DBMS_PIPE are not confined to the Oracle RDBMs. You also can use DBMS_PIPE in all sorts of new and creative ways. You can parallelize your own programs. You can communicate between a client program in Oracle Forms and a server−based process, without having to commit any data. You can build a debugger for your server−side PL/SQL programs.

The DBMS_PIPE package is just one of many such mind− and functionality−expanding new resources. Do you need to issue your own locks? Do you need to detect whether another process in your current session has committed data? Use the DBMS_LOCK package. Do you want to issue messages from within your PL/SQL programs to help trace and debug your program? Check out the DBMS_OUTPUT package. Would you like to schedule jobs within the RDBMs itself? Explore the DBMS_JOB package. The list goes on and on, and is constantly growing. With the Oracle−supplied packages, you have at your disposal many of the same tools used by the internal Oracle product developers. With these tools, you can do things never before possible!

Chapter 15 of *Oracle PL/SQL Programming* provides details on many of the stored packages of the Oracle Server.

> ***NOTE:*** Each time you install a new version of the Oracle Server, you should peruse the *dbms*.sql* and *utl*.sql* files (usually found in an operating system−specific variant of the *rdbmsN/admin* directory, where *N* is the release number, as in: *rdbms73/admin*). See what is new in terms of both entirely new builtin packages and also changes to existing builtin packages. Don't rely solely on reading the New Features section in the Oracle Server documentation.

1.5.2 Prebuilt Packages

Prebuilt packages are the newest type of package in the PL/SQL development arena, and in many ways offer the most promise to PL/SQL developers. "Prebuilt" (my own terminology) refers to a package that is designed, built, and tested by a third party and then made available to you, either as free shareware or as a licensed product.

Prebuilt packages will most likely come in two forms: miscellaneous utilities and libraries. A utility package might be a single package that supplies functionality in a specific area, such as a package that makes it easier to work with the job queue of the Oracle Server (interfacing with DBMS_JOB, in other words). A package library is, on the other hand, a coherent set of packages that work together and offer an entire layer of reusable code.

PL/Vision is an example of a package library and is, to my knowledge, one of the first −− if not the first −− such library to be made available to PL/SQL developers. I hope the time will come when third−party PL/SQL developers regularly publish prebuilt packages, whether standalone utilities or libraries, to help the entire PL/SQL community. In the meantime, PL/Vision and this book will offer PL/SQL developers an extensive set of prebuilt utilities and functionality to enhance their development environments.

1.5.3 Build−Your−Own Packages

The third type of package is the build−your−own (BYO) package. This is a package whose specification and body you write yourself. You decide what programs and data are publicly available and how those programs should be called.

The BYO package is, of course, the most common kind of package. You will, I hope, create many, many packages during your PL/SQL development career. Some of those packages might even evolve into prebuilt packages used by other developers. Some will remain at the core of your very business−specific applications.

However your package is used, it is critical that you learn how to build packages that can be easily debugged, maintained, and enhanced. The rest of this chapter explores aspects of package syntax and features. The next chapter, Chapter 2, *Best Practices for Packages*, provides advice about how best to design and build your packages.

PREVIOUS | HOME | NEXT
1.4 Using Packages | BOOK INDEX | 1.6 Building Packages

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

1.6 Building Packages

You will want to take maximum possible advantage of builtin and prebuilt packages. Soon, however, you will be building your own packages. This section explores syntax and issues related to package construction.

Remember that a package consists of up to two distinct parts: the specification and the body. The specification and body are completely distinct objects. You can write and compile the specification independently of the body. When you create and replace stored packages in the database, you perform this action separately for each specification and each body. The package specification describes the different elements of the package that can be called or referenced. The package body contains the implementation or executable code for the elements of the specification.

This chapter has already introduced to you the package specification. Before turning our attention to the package body, let's examine the circumstances under which you should consider building your own package.

1.6.1 When Should You Build a Package?

When should you build a package, instead of just creating a set of standalone procedures and functions? Anyone who has attended any of my classes or presentations could probably guess my answer (or is it my dogma): Always! You should always build a package! A package is the answer to all of your problems in PL/SQL!

I do realize that my readers deserve a more reasoned and nuanced answer. So I'll review the reasons for building a package. I believe you will want to create a package whenever you find yourself in the following kinds of situations.

1.6.1.1 Clean up a bewildering mass of standalone functions and procedures

Do you have dozens, perhaps hundreds, of standalone functions and procedures stored in the database? Do you wonder if there is overlap between these programs? Do you have a sense that you have lost control of your development/stored code environment? The package construct can help bring everything into focus and get you and your code organized.

The package is the closest thing in PL/SQL (prior to PL/SQL Release 3) to an object. A package bundles together different −− usually related −− PL/SQL elements. Rather than have separate, standalone programs to maintain the contents of a particular table, to return information from that table and so on, you can place all of those programs within the context of the package. This simple transfer of code within the boundaries of a package makes it easier for you to manage all of your code. It will also be much, much easier for others to understand and maintain the code you have written.

So if you are new to packages and really do not believe that you have a grip on what is actually out there in the database, it would be a good time to analyze your stored code and reorganize it into a set of packages.

1.6.1.2 Maintain control over your data structures

The data structures defined inside the body of packages are shared by (or accessible to) all elements of the package, but cannot be referenced outside of the package. This means that you can control tightly any access to your data. I recommend, in fact, that you never define variables and data structures in the specification of the package.

This advice applies not only to in−memory data defined inside the package, but also to table−based data in the Oracle Server. You can set up your environment so that users cannot access database tables directly for read or write purposes. You then create a package that maintains the data in the table. The owner of the package is given access to the underlying table. Finally, you grant execute authority on the package to the end users. Now all access to the table goes through the package. As a result, you can now use the PL/SQL language to apply business rules to all user−initiated transactions. This approach is particularly important when you use a third−party frontend tool like Gupta Corporation's SQL*Windows to access the Oracle database. You want to do everything you can to avoid hard−coding your data structures (in the form of SELECTs, INSERTs, and so on) into client−side code, whether in Oracle Developer/2000 or in SQL*Windows.

1.6.1.3 Need global data structures for your PL/SQL programs

When you declare a variable in the specification of a package, that variable becomes, for all intents and purposes, a global variable within a given user session. It can be accessed by any PL/SQL program, regardless of where the variable was first referenced and assigned a value, as long as that program is run under the same session. (Note that you can make use of the DBMS_PIPE package to make data available across different Oracle sessions.)

If you declare a data structure inside the body of the package, that structure is still "global" −− but only within the package. It also *persists* for the duration of your session; in other words, the variable (whether scalar or composite) maintains its value until it is changed or until you disconnect.

If you need to define and then access persistent, global data in your session, the PL/SQL package is the only way to accomplish this particular trick. Common scenarios requiring global data include:

- Configuration information for a session or even for an application. What is my printer name? What is the latest date allowed for entry of new orders?
- Memory−based lists of information. Rather than having to constantly go back to the database to obtain and display a list of options, you can load these values into a PL/SQL table and then display those values. You will use more memory, but improve the performance of your application.
- Running totals and other accumulated or derived values. Your application might perform lots of what−if analysis on current information in the database and new information entered by users. Package−based data structures can be used to keep track of derived, analytical values until they are ready to be saved to the database or discarded when the session ends.

1.6.1.4 Remove hard−coded literals from your application

We all know that you should never, or hardly ever, put hard−coded literal values in your application code.

One of the most common causes for program failure is, I believe, the undying belief by programmers that a particular value will never change and so can be hard−coded into a system. Why do we developers make this mistake again and again? I believe that it is a subconscious reaction to the fundamental uncertainty in our lives. Our end users change their minds about their specifications, requirements, and user interface preferences on a daily −− if not hourly −− basis. We can find no comfort, no security, from our users.

And what about our own IS management? CIOs and their architectures teams seem to always be discovering the next "silver bullet" methodology or application development tool −− and pulling the proverbial carpet out from under us.

So when you read in your application specifications that C will represent a "closed account," you are swept up by an irresistible urge to bet the house on that particular value not changing. But of course it will −− and your code will suffer for it.

You are much better off establishing a package of constants that holds all the application−wide and/or system−wide constants that you will be using. Then when you feel the need to hard−code a literal into your program, instead switch over to your constants package (I talk more about this in the "simultaneous package construction" best practice, described in the next chapter). Deposit the literal value there and give it a name with a CONSTANT declaration. From then on, reference the value by package constant *name*. If the value ever must be changed, you change it in the constants package specification, recompile all dependent programs, and you are done!

As an alternative, you can also define constants in the packages to which they relate. I have found that I will usually create a single, application−level package of constants if I am reviewing and cleaning up an existing application.

1.6.1.5 Isolate volatile areas of functionality

If you know that an aspect of your application −− or even the technology on which your application is based −− is volatile and bound to change, build a package around that functionality to protect your application from that volatility.

One of the most volatile areas of an application is its underlying data structures. Many developers tend to think of the database −− all those tables, primary keys, constraints and so forth −− as the bedrock of the application. Now, it is true that these data structures are the foundation upon which most code is built. But it is most definitely not true that this foundation is unchanging. The entity−relationship diagram (ERD) maps the real world to the relational world.

The real world −− at least as perceived by our users −− is a moving target. New or changed tables, modified columns, evolving relationships: these are an everyday element of our work. They are also one of the most complicated aspects of our development. It is not unusual, for instance, to require triple and quadruple joins of tables to retrieve the most basic information about an entity.

So, yes, you have to build your applications on top of these data structures, but you do not have to do so in a blind and short−sighted manner. There are two approaches you can take to deal with the complexity and volatility:

Option 1. Train all of your developers about all the subtleties of your organization's data. You will, of course, need to keep training them as your data structures change over time.

Option 2. Hide as many of these subtleties as you can, allowing your developers to work at a higher level of abstraction.

The optimal choice should be clear. While it is an admirable goal to educate everyone about every aspect of the work, it is simply not practical. You can use a package to encapsulate the triple joins and present a unified front to a developer. They can skip the complex SQL and apply the prebuilt package code to their needs much more quickly. Assume, in other words, that your structures will change. Protect your code (and your sanity) accordingly. Here are some recommendations to follow that employ the package to improve the robustness of your application:

1. *Avoid implicit queries.* When you write an implicit query, you place a SELECT statement directly in your program. You fetch data from the database directly into local PL/SQL variables. When you take this approach, you essentially hard−code your data structures into your programs. What happens when that two−table join turns into a six−table join? You have to find all those implicit queries and fix them.

2. *Set as a general rule that developers do not write DML statements directly in their own applications.* Again, if everyone is writing whatever SQL they find appropriate to their individual circumstances, you will end up with an application in which your entity relationships are distributed throughout your code. How do you maintain and upgrade such code to match your changing database? Instead...

3. *Consolidate all of your SQL statements into one or more packages.* Provide programmatic access to the SQL, or offer explicit cursors defined in the package. With this approach, you anticipate developer needs concerning the SQL layer. Will developers need to update the `emp` table with a new salary? Provide a procedure that does this. Do you need to query employees by decreasing salaries? Create an explicit cursor with this structure and let developers open and fetch from the cursor.

If you succeed in predefining and consolidating your SQL statements behind the package interface, you will have gone a long way towards making your application change−proof. This approach takes up−front planning and lots of discipline. You might even want to build scripts to query the contents of the USER_SOURCE data dictionary view to verify that DML statements do not appear outside of your SQL packages. In the long run, however, you will achieve higher productivity and higher code quality. Individual developers are liberated from writing complex, bug−prone SQL and instead concentrate on the user interface −− or whatever is the task at hand.

1.6.1.6 Hide weaknesses to facilitate upgrades and fixes.

Every release of every piece of software comes with bugs, undocumented "features," and functionality that lacks, shall we say, a certain polish. You can whine about this situation, but sooner or later you have to deal with it. This usually means that you have to use elements of the language that are substandard. You will figure out the workaround or whatever compensation is necessary to keep the development process moving forward. At this point, you have two choices of how to apply this workaround:

1. Whenever you encounter the problem area, you code the workaround directly in the program.

2. You build a package that contains the workaround. The package specification provides the solution without revealing the nature of the workaround. It is hidden inside the body of the package.

In the first approach, which is almost always the route chosen, you hard−code the (usually temporary) drawback of the language directly in multiple places in your programs. When the upgrade to PL/SQL (or whatever language you are using) arrives at your installation, you have to hunt down every place you put the workaround and replace it with the new, fixed functionality.

With the second (package−based) approach, the code for the workaround is in one place only. When the patch tape or upgrade arrives, you make the change only inside the body of the package. All the code that called the package element to apply the workaround will now call that same package element, but this time use the new, fixed version.

If you build yourself a layer of code with a package that hides the implementation of workarounds, you can then easily and rapidly apply upgrades that completely obviate the need for the workaround.

1.6.1.7 An example

To illustrate this technique, consider the task of clearing or emptying PL/SQL tables. Prior to Release 2.3 of PL/SQL, the only way to delete all rows from a PL/SQL table was to assign an empty table to the populated table. There is, in other words, no DELETE function or operator for PL/SQL tables. The PLVtab package of PL/Vision makes it as easy as possible for you to use PL/SQL tables by predefined table types in the package. To similarly ease the task of emptying tables, PLVtab provides an empty table for each table TYPE defined in the package. The **PLVprsps.init_table** procedure below shows how these elements are put to use in PL/Vision to delete all rows from a PL/SQL table:

```
PROCEDURE init_table
   (tokens_out IN OUT PLVtab.vc2000_table,
    num_tokens_out IN OUT INTEGER)
IS
BEGIN
   tokens_out := PLVtab.empty_vc2000;
   num_tokens_out := 0;
END;
```

The problem with this approach is that it exposes the implementation of my workaround and makes it very difficult to upgrade this code to PL/SQL Release 2.3.

You see, PL/SQL Release 2.3 provides a DELETE method for PL/SQL tables. Rather than assigning an empty table to delete all rows from **tokens_out**, I could simply issue this statement:

```
tokens_out.DELETE;
```

I can certainly make this substitution in the **init_table** program, but in how many other places are these empty tables utilized? Taking a different approach within PLVtab would have left me in a much stronger position. Suppose that instead of providing a set of predefined empty tables, I built a series of *procedures* to empty the tables. The procedure to empty the VARCHAR2(2000) tables would look like this:

```
PROCEDURE empty (table_inout IN OUT PLVtab.vc2000_table) IS
BEGIN
   tokens_inout := PLVtab.empty_vc2000;
END;
```

and the **init_table** procedure would change to this:

```
PROCEDURE init_table
   (tokens_out IN OUT PLVtab.vc2000_table,
    num_tokens_out IN OUT INTEGER)
IS
BEGIN
   PLVtab.empty (tokens_out);
   num_tokens_out := 0;
END;
```

Notice that there is no mention of the empty table outside of PLVtab. So when Release 2.3 comes along, I can change the empty procedure of PLVtab as follows:

```
PROCEDURE empty (table_inout IN OUT PLVtab.vc2000_table) IS

BEGIN
   tokens_inout.DELETE;
```

```
      END;
```

Like magic, without making any change to the **PLVprsps.init_table** procedure, it is now using the new capabilities of the PL/SQL language. This is just one very simple example of how you can and should use packages to make feasible the upgrade to new features and fixes to bugs.

There are many situations in PL/SQL development that cry out for a package. I hope the previous sections will help raise a flag as you proceed through your application development projects. Do you find yourself dealing repeatedly with some weakness in PL/SQL or another element of the Oracle product set? Are you writing the same complex SQL statement again and again in your code? Stop! Take the time required to bundle the logic, the flaws, the relationships into a package. The payoff will come instantaneously and continuously; the investment will never be regretted.

Once you have decided to build your package and constructed the interface (specification) for that package, you will then embark on constructing the guts of the package: its body.

1.6.2 The Package Body

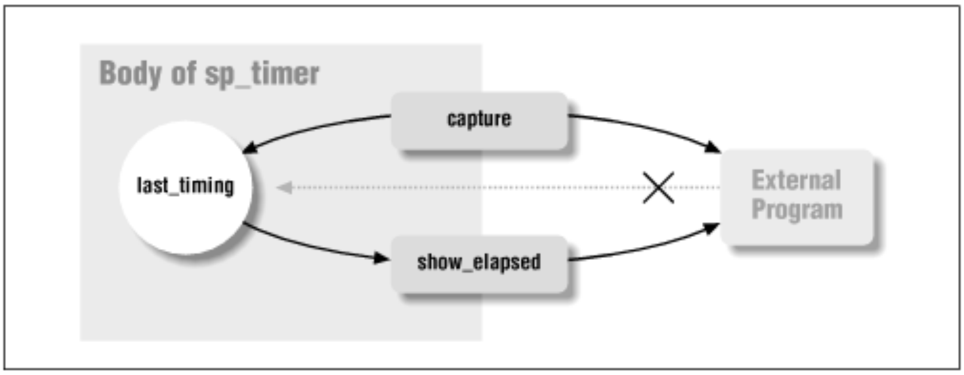
The body of the package contains all the code behind the package specification: the implementation of the modules, cursors, and other elements. Whereas package specifications tend to be short and to the point ("here are the programs you can run and the data you can access"), package bodies can easily grow to intimidating length and complexity. Several coding recommendations presented in the next chapter address organizing your package bodies.

The packages of PL/Vision offer many examples of package bodies. Let's take a look now at a relatively simple package body. The package body shown below illustrates the code required to implement the specification of the **sp_timer** package shown earlier.

```
PACKAGE BODY sp_timer
IS
   last_timing NUMBER := NULL;

   PROCEDURE capture IS
   BEGIN
      last_timing := DBMS_UTILITY.GET_TIME;
   END;

   PROCEDURE show_elapsed IS
   BEGIN
      DBMS_OUTPUT.PUT_LINE
        (DBMS_UTILITY.GET_TIME - last_timing);
   END;
END sp_timer;
```

Notice that the body contains a declaration of a local variable, **last_timing**. This variable does not appear in the specification; instead, it is referenced within **capture** (which sets the value of **last_timing**) and in **show_elapsed** (which references the variable). Consequently, the only programs that can directly reference the **last_timing** variable are **capture** and **show_elapsed**, as shown in Figure 1.4.

Figure 1.4: A Booch diagram for the sp_timer package



The body of the package resembles a standalone module's declaration section. It contains both the declarations of variables and the definitions of all package modules. The package body may also contain an execution section, which is called the *initialization section* because it is run only once, to initialize the package. (This aspect of packages is discussed in the next section.)

1.6.2.1 Package body syntax

The general syntax for the package body is shown below:

```
PACKAGE BODY package_name
IS
   [ declarations of variables and types ]
   [ header and SELECT statement of cursors ]
   [ header and body of modules ]
[ BEGIN
     executable statements ]
[ EXCEPTION
      exception handlers ]
END [ package_name ];
```

In the body you can declare other variables, but you do not repeat the declarations in the specification. The body contains the full implementation of cursors and modules. In the case of a cursor, the package body contains both the header and the SQL statement for the cursor. In the case of a module, the package body contains both the header and body of the module.

The BEGIN keyword indicates the presence of an execution or initialization section for the package. This section can also optionally include an exception section.

As with a procedure, function, and package specification, you can add the name of the package, as a label, after the END keyword in both the specification and package.

1.6.3 The Initialization Section

The first time your application makes a reference to a package element, the entire package (in precompiled form) is loaded into the System Global Area of the database instance, making all objects immediately available in memory. All package data structures are defined and default values are assigned. You can supplement this automatic instantiation of the package code with the execution of startup code for the package. This initialization code is contained in the optional initialization section of the package body.

The initialization section consists of all statements following the BEGIN statement through the END statement for the entire package body. It is called the initialization section because the statements in this part of the package are executed only once, the first time an object in the package is referenced (to name a few possibilities, when a program is called, a cursor is opened, or a variable is used in an assignment). The initialization section initializes the package; it is commonly used to set values for variables declared and referenced in the package.

The initialization section is a powerful mechanism: PL/SQL detects automatically when this code should be run. You do not have to explicitly execute the statements, and you can be sure they are run only once. There is, however, a downside to use of the initialization section. It can be difficult to trace actions triggered automatically by the package ("Now where does that variable get set?" "How did that record get inserted into that table? I don't see it in any of *my* code!"). It can also be difficult for less experienced developers to locate and be aware of the initialization code.

You should only use the initialization section when you cannot rely on the normal initialization mechanisms (such as setting a default value when a variable is declared), as explored below.

1.6.3.1 When to use the initialization section

Use the initialization section only when you need to set the initial values of package elements using rules and complex logic that cannot be handled in the default value syntax for variables. You do not, for example, need an initialization section to set the value of the constant **earliest_date** to today's date. Instead, simply declare the variable with a default value.

The following package body contains unnecessary initialization code:

```
PACKAGE config
IS
   earliest_date DATE;
BEGIN
   earliest_date := SYSDATE;
END config;
```

This code should be replaced with a much simpler and more direct default assignment as follows:

```
PACKAGE config
IS
   earliest_date DATE := SYSDATE;
END config;
```

Suppose, on the other hand, that you wanted to use PLVlst to maintain a list of companies that the user has selected for financial analysis. The analysis is performed by a package you built called **compcalc** (COMPany CALCulation). You want to make sure that the list is defined and available whenever any program in **compcalc** is used, but you don't want the list refreshed or created anytime after the start of a user session. This is the perfect opportunity for an initialization section. The following package shows you how to achieve this effect:

```
PACKAGE BODY compcalc
IS
   c_list CHAR(8) := 'compcalc';

   PROCEDURE total_sales
   IS
   BEGIN
      FOR list_ind IN 1 .. PLVlst.nitems (c_list)
      LOOP
         calc_sales (PLVlst.getitem (c_list, list_ind));
      END LOOP;
   END;

BEGIN
   PLVlst.make (c_list);
END;
```

The package body declares a constant containing the name of the list. This constant is then referenced throughout the package to avoid hard−coding of literals. The **total_sales** procedure calculates the sales for each item in the list. The initialization section at the bottom of the package consists of a single line of code that makes the list for use in the package. The **`PLVlst.make`** procedure is called only once in a user's session; the list is then available for the duration of the session.

You will also want to use an initialization section when you need to handle exceptions that might arise when initializing values.

1.6.3.2 The exception section

As noted earlier, the initialization section of a package can also have its own exception section. The ability to handle exceptions is one of the most important characteristics of this section of the package.

Remember that the exception handlers in this section will only trap exceptions that occur in the initialization section itself. If an exception is raised in one of the programs defined in the package or if an exception is raised in the process of instantiating data in the package, the initialization section exception handlers will not be able to trap those exceptions.

To understand this flow, consider the following package specification and body:

```
PACKAGE going
IS
   PROCEDURE bad;
END going;

PACKAGE BODY going
IS
   fast VARCHAR2(3) := 'fast';

   PROCEDURE bad IS
   BEGIN
      DBMS_OUTPUT.PUT_LINE (fast);
   END;
BEGIN
   NULL;
EXCEPTION
   WHEN VALUE_ERROR
   THEN
      DBMS_OUTPUT.PUT_LINE ('too late');
END going;
```

This package contains a single public procedure, **`going.bad`**, and a single private variable, **`fast`**. When any program tries to execute **`going.bad`**, the PL/SQL runtime engine loads the package into shared memory and instantiates all package data. When it tries to assign the value of **`fast`** to the variable **`going.bad`**, however, the runtime engine will raise the VALUE_ERROR exception. The literal value has four characters, but the **`fast`** variable is restricted to three characters.

It would appear at first glance that the exception handler for VALUE_ERROR at the bottom of the package body would trap this exception and display "too late". Instead, the exception will go unhandled.

Let's now see how to use the initialization section to your advantage when initializing values in a package. Consider the simple package body below:

```
PACKAGE BODY analysis
IS
   best_salesperson INTEGER := sales_list (1);
END analysis;
```

This package sets the default value for the best salesperson ID to the value in row 1 of the **`sales_list`** PL/SQL table. If for any reason this row has not yet been defined, the PL/SQL runtime engine will raise a NO_DATA_FOUND exception simply for trying to *reference* the **`analysis.best_salesperson`** variable.

If you move the assignment to an initialization section, on the other hand, you can handle gracefully the scenario in which row 1 in the table has not been set:

```
PACKAGE BODY analysis
IS
   best_salesperson INTEGER;
BEGIN
   best_salesperson := sales_list (1);
EXCEPTION
   WHEN NO_DATA_FOUND
   THEN
      best_salesperson := NULL;
END analysis;
```

1.6.4 Working with Large Packages

There is a limit to the size of any PL/SQL program, whether it is a procedure or a package. The restriction on code size is determined by the limitation of the size of the PL/SQL parse tree, which is built for compilation purposes. In PL/SQL Release 2.2 and earlier, the number of nodes in the parse tree is limited to 16K. In Release 7.3, the upper limit is raised to 32K. A node in the parse tree typically consists of a keyword, application identifier, or operator.

Whatever the current limit, someone will always come up against it. And even if you don't actually threaten to exceed the theoretical limit, you might easily come up against a practical limit to the size of the package you can deal with. For example, do you really want to endure a 10−minute compile every time you have to make a small change to your monster package?

When PL/SQL developers ask me what they should do about their packages that are 10,000 lines long and causing all sorts of problems, I tell them: shorten your package or turn it into multiple packages. There, that was easy! And sometimes it really is that easy. Most programs, and certainly most packages, contain redundancies that needlessly increase code volume. A very careful review will almost always uncover ways to reduce the size of a program through fanatical modularization. In addition, few developers properly break apart their packages into distinct areas of functionality. Most large packages I have analyzed should have been broken up into multiple packages, regardless of the original size so that functionality would be more accessible and reuseable.

Yet in other situations, it really is difficult to shrink the size of one's package. Many real world tables have dozens, if not hundreds, of columns. Any Data Manipulation Language (DML) statement on such a table requires many bytes of code. And that 25,000−line package might really just contain code related to one specific area of functionality. To break that package into more than one package means requiring the user of those packages to deal with different package names. Why does procedure A reside in package X while function B can be found only in package Y?

There is, however, a technique you can use to break up a large package into multiple packages, while still maintaining the appearance of a single package for your users. You can create a cover package[4] that offers all the elements of the package under a single name. This cover package is, however, nothing more than a pass−through to other packages that contain the application logic.

[4] Note that the initial idea for this cover technique came from John M. Beresniewicz.

To see how this cover package technique works, consider the three packages defined below:

```
CREATE OR REPLACE PACKAGE forreal1
IS
   PROCEDURE proc1;
```

```
END forreal1;
/
CREATE OR REPLACE PACKAGE BODY forreal1
IS
   PROCEDURE proc1 IS BEGIN DBMS_OUTPUT.PUT_LINE ('for real 1'); End;
END forreal1;
/
CREATE OR REPLACE PACKAGE forreal2
IS
   PROCEDURE proc1;
END forreal2;
/
CREATE OR REPLACE PACKAGE BODY forreal2
IS
   PROCEDURE proc2 IS BEGIN DBMS_OUTPUT.PUT_LINE ('for real 2'); End;
END forreal2;
/

CREATE OR REPLACE PACKAGE cover
IS

   PROCEDURE proc1;
   PROCEDURE proc2;
END cover;
/
CREATE OR REPLACE PACKAGE BODY cover
IS
   PROCEDURE proc1 IS BEGIN forreal1.proc1; END;
   PROCEDURE proc2 IS BEGIN forreal2.proc2; END;
END cover;
/
```

The **`cover`** package contains two procedures. But if you look at the implementation of those procedures in **`cover`**, you see that all they do is call the "for real" version of those procedures in their respective "for real" packages, **`forreal1`** and **`forreal2`**. When a developer calls the cover procedures, she doesn't know that she is actually calling the underlying "for real" procedures.

```
SQL> exec cover.proc1
for real 1
SQL> exec cover.proc2
for real 2
```

This redirection or passthrough in and of itself is not a major breakthrough. What makes this cover layer of code so useful is that you can set up access to these packages so that a developer can *only* execute the cover package and never even know about the underlying packages. This preserves the integrity of existing applications (written way back when all the code managed to fit in a single package) and protects the underlying code from being accessed improperly.

Suppose, for example, that the cover and "for real" packages are created in the APPOWNER account. I can then grant access to the **`cover`** package to SCOTT as follows:

```
SQL> GRANT EXECUTE ON cover TO scott;
```

I can also create a synonym for cover:

```
SQL> CREATE PUBLIC SYNONYM cover FOR appowner.cover;
```

Now a developer working in the SCOTT account can execute the **`cover`** procedures, but cannot execute the "for real" package−based procedures. All anyone knows about is the cover −− and if you name the packages differently, you don't even know you are working with a cover!

Sure, it would be better to be able to keep all related code in a single package. But at least with the cover technique developers using your software don't have to know about the smoke−filled back room manipulations.

1.6.5 Calling Packaged Functions in SQL

As of PL/SQL Release 2.1, you can call stored functions like **`total_comp`** anywhere in a SQL statement where an expression is allowed, including the SELECT, WHERE, START WITH, GROUP BY, HAVING, ORDER BY, SET, and VALUES clauses. (Since stored procedures are in and of themselves PL/SQL executable statements, they cannot be embedded in a SQL statement.)

Suppose, for example, that you need to calculate and use an employee's total compensation both in native SQL and in your forms. The computation itself is straightforward enough:

```
Total compenstation = salary + bonus
```

My SQL statement would include this formula:

```
SELECT employee_name, salary + NVL (bonus, 0)
  FROM employee;
```

while my Post−Query trigger in my Oracle Forms application might employ the following PL/SQL code:

```
:employee.total_comp := :employee.salary + NVL (:employee.bonus, 0);
```

In this case, the calculation is very simple, but the fact remains that if, for any reason, you need to change the total compensation formula (different kinds of bonuses, for example), you would then have to change all of these hard−coded calculations both in the SQL statements and in the frontend application components.

A far better approach is to create a function that returns the total compensation:

```
FUNCTION total_comp
   (salary_in IN employee.salary%TYPE, bonus_in IN employee.bonus%TYPE)
   RETURN NUMBER
IS
BEGIN
   RETURN salary_in + NVL (bonus_in, 0);
END;
```

Then I can replace the formulas in my code as follows:

```
SELECT employee_name, total_comp (salary, bonus)
  FROM employee;

:employee.total_comp := total_comp (:employee.salary, :employee.bonus);
```

You can use one of your own functions just as you would a builtin SQL function such as TO_DATE or SUBSTR or LENGTH. (*Chapter 19*, of *Oracle PL/SQL Programming*, offers many examples and details about how to use stored functions in this way.) There are, for example, many restrictions on functions called in SQL, most notably that:

- The function may not execute INSERTs, UPDATEs, or DELETEs. You can't change data while you are executing the function in your SQL statement.
- The function may not execute any builtin packaged programs. You cannot, in other words, use DBMS_SQL or DBMS_PIPE in a function and then call it in SQL.

In the remainder of this chapter, I take a look at the steps you need to take to make package−based functions callable in your SQL statements. Specifically, you will need to make use of the RESTRICT_REFERENCES pragma.

1.6.5.1 RESTRICT_REFERENCES pragma

A stored function can exist as a standalone function or as a function in a package. For standalone functions, the Oracle Server automatically determines whether it is callable in SQL. It will, for example, reject your SQL statement if it uses a function that issues an UPDATE statement. The situation with packaged functions is a bit more complicated.

As noted earlier, the specification and body of a package are distinct; a specification can exist even before its body. For this and other reasons, the Oracle Server cannot automatically determine (when you execute your SQL) that a packaged function is valid for SQL execution. Instead, you must state explicitly the "purity level" of a function in a package with the RESTRICT_REFERENCES pragma. The Oracle Server then determines at compile time (of the package body) if the function violates the purity level, and raise a compilation error if this is the case. Once the package is compiled, the functions for which assertions have been made can be called in SQL.

Let's explore the specific syntax required to achieve this effect.

A *pragma* is a special directive to the PL/SQL compiler. If you have ever created a programmer−defined, named exception, you have already encountered your first pragma. In the case of the RESTRICT_REFERENCES pragma, you are telling the compiler the purity level you believe your function meets or exceeds.

You need a separate PRAGMA statement for each packaged function you wish to use in a SQL statement, and it must come after the function declaration in the package specification (you do not specify the pragma in the package body).

To assert a purity level with the pragma, use the following syntax:

```
PRAGMA RESTRICT_REFERENCES
   (function_name, WNDS [, WNPS] [, RNDS] [, RNPS])
```

where **`function_name`** is the name of the function whose purity level you wish to assert, and the four different codes have the following meanings:

Purity Code	Description
WNDS	Stands for "Writes No Database State." Asserts that the function does not modify any database tables.
WNPS	Stands for "Writes No Package State." Asserts that the function does not modify any package variables.
RNDS	Stands for "Reads No Database State." Asserts that the function does not read any database tables.
RNPS	Stands for "Reads No Package State." Asserts that the function does not read any package variables.

Notice that only the WNDS level is mandatory in the pragma. That is consistent with the restriction that stored functions in SQL may not execute an UPDATE, INSERT, or DELETE statement. All other states are optional. You can list them in any order, but you must include the WNDS argument. No one argument implies another argument. For example, I can write to the database without reading from it. I can read a package variable without writing to a package variable.

Here is an example of two different purity level assertions for functions in the **`company_financials`** package:

```
PACKAGE company_financials
IS
   FUNCTION company_type (type_code_in IN VARCHAR2)
      RETURN VARCHAR2;

   FUNCTION company_name (company_id_in IN company.company_id%TYPE)
      RETURN VARCHAR2;

   PRAGMA RESTRICT_REFERENCES (company_type, WNDS, RNDS, WNPS, RNPS);
   PRAGMA RESTRICT_REFERENCES (company_name, WNDS, WNPS, RNPS);
END company;
```

In this package, the **`company_name`** function does read from the database to obtain the name for the specified company. Notice that I placed both pragmas together at the bottom of the package specification. The pragma does not need to immediately follow the function specification. I also went to the trouble of specifying the WNPS and RNPS arguments for both of the functions. Oracle Corporation recommends that you assert the highest possible purity levels so that the compiler will never reject the function unnecessarily.

I have found, on the other hand, that the PL/SQL compiler does at times reject my purity level assertions when there does not seem to be any apparent violation. You may at times have to retreat to the minimal WNDS assertion simply to get your package to compile.

1.6.5.2 Asserting the purity level of the initialization section

If your package contains an initialization section (executable statements after a BEGIN statement in the package body), you must also assert the purity level of that section. The initialization section is executed automatically the first time any package object is referenced. So if a packaged function is used in a SQL statement, it will trigger execution of that code. If the initialization section modifies package variables or database information, the compiler needs to know about that through the pragma.

You can assert the purity level of the initialization section either directly or indirectly. To use a direct assertion, you use this variation of the pragma RESTRICT_REFERENCES:

```
PRAGMA RESTRICT_REFERENCES
   (package_name, WNDS, [, WNPS] [, RNDS] [, RNPS])
```

Instead of specifying the name of the function, you include the name of the package itself, followed by all the applicable state arguments. In the following argument I assert only WNDS and WNPS because the initialization section reads data from the configuration table and also reads the value of a global variable from another package (**`session_pkg.user_id`**).

```
PACKAGE configure
IS
   PRAGMA RESTRICT_REFERENCES (configure, WNDS, WNPS);
   user_name VARCHAR2(100);
END configure;

PACKAGE BODY configure
IS
BEGIN
   SELECT lname || ', ' || fname INTO user_name
     FROM user_table
    WHERE user_id = session_pkg.user_id;
END configure;
```

Why can I assert the WNPS even though I do write to the **`user_name`** package variable? Answer: It's a variable from this same package, so the action is not considered a side effect.

You can also assert the purity level of the package's initialization section by allowing the compiler to infer that level from the purity level(s) of all the pragmas for individual functions in the package. In the following version of the company package, the two pragmas for the functions allow the Oracle Server to infer a combined purity level of RNDS, WNPS for the initialization section. This means that the initialization section cannot read from the database and cannot write to a package variable.

```
PACKAGE company
IS
   FUNCTION get_company (company_id_in IN VARCHAR2)
      RETURN company%ROWTYPE;

   FUNCTION deactivate_company (company_id_in IN company.company_id%TYPE)
      RETURN VARCHAR2;

   PRAGMA RESTRICT_REFERENCES (get_company, RNDS, WNPS);
   PRAGMA RESTRICT_REFERENCES (deactivate_name, WNPS);
END company;
```

Generally, you are probably better off providing an explicit purity level assertion for the initialization section. This makes it easier for those responsible for maintaining the package to understand both your intentions and your understanding of the package.

PREVIOUS | HOME | NEXT
1.5 Types of Packages | BOOK INDEX | 2. Best Practices for Packages

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

Advanced Oracle PL/SQL

Programming with Packages

SEARCH

PREVIOUS

Chapter 2

NEXT

2. Best Practices for Packages

Contents:

Packages are the most important construct in PL/SQL for building reusable code and plug−and−play components, and for employing object−oriented design techniques. As you become more comfortable with the language, you will find more of your time spent inside packages −− and using programs from packages built by other developers. This may be a very pleasant and rewarding experience −− if the packages are designed and implemented properly. If, on the other hand, you decide to build your packages in the same helter−skelter method (or lack thereof) I run into way too often, life out there in front of the monitor may get pretty miserable.

This chapter discusses some of the most important "best practices" for package construction and goes into detail on an effective coding style for packages. If you follow the ideas presented below, you have a very good chance of writing packages that are readable, maintainable, and enhanceable. It is also much more likely that other developers will find your packages usable and useful. You will find additional explanation regarding these practices, along with examples of the application of these practices, in the sections covering specific PL/Vision packages.

The following list offers a summary of the best practices for packages covered in this chapter:

- *Start with packages.* Get out of the habit of building standalone procedures and functions. Instead, start with a package.

- *Use a consistent and effective coding style.* Use comment lines as banners to delineate the different kinds of elements in the package. Employ a standard ordering for the elements of the package.

- *Choose appropriate and accurate names for both the package and the elements in the package.* As with any other kind of identifier, your package name should clearly communicate the point of the package. Avoid redundancies in the package name and its element names (such as **`emp.emp_name`**).

- *Organize package source code.* Come up with consistent file−naming conventions for the source code you stored in operating system files. Separate the package specification and body CREATE OR REPLACE statement into separate files.

- *Construct the optimal interface to your package.* Design your package so that it is easy −− and a pleasure −− to use. When you build packages for reuse, other PL/SQL developers become your users. Treat them with respect. Make the parameters in your programs case−insensitive. Don't require users to know about and pass literal values.

- *Build flexibility directly into your packages.* Anticipate areas where options and flexibility will be required and then build them right in −− either with additional parameters or separate programs. Build toggles into your package so the behavior of the package can be changed without having to change the user's application.

- *Build windows into your packages.* Packages allow you to control tightly what a developer can see and affect inside the package. The flip side of this control is blindness and bewilderment. Your users can't figure out what is going on. Package windows allow for controlled read−only access to package data and activity.

- *Overload aggressively for smart packages.* Overloading modules means that you create more than one program with the same name. Overloading transfers the burden of knowledge from the user to the software. You do not have to try to remember the different names of the modules and their specific arguments. Properly constructed, overloaded modules will anticipate the different variations, hiding them behind a single name, and liberate your brain for other, more important matters.

- *Modularize the package body to create maintainable packages.* To build packages that are both immediately useful and enhanceable over the long run, you must develop a wicked allergy to any kind of code duplication inside the package. You need to be ready, willing, and able to create private programs in your package to contain all the shared code behind the public programs of the package.

- *Hide your package data.* Never define variables in the specification of a package (except when explicitly needed that way). Instead, declare them in the body and then provide a procedure to set the value of that variable, and a function to change the value of that variable (get−and−set). If you follow this practice, you will retain control over the values of, and access to, your package data. You can also then trace any reads from and writes to that data.

- *Construct multiple packages simultaneously.* As you build a program, be aware of both existing packages and the need for new areas or layers of functionality. Take the time to break off from development of program A to enhance packages B, C, and D −− and then apply those new features back into A. It might take a little bit longer to finish your first program, but when you are done you will have strengthened your overall PL/SQL development environment and increased your volume of reusable code.

2.1 Starting With Packages

Late in July 1996, I received this note from one of my technical reviewers, John M. Beresniewicz:

> I've built half a dozen pretty hefty packages, and still I find myself wondering at the start of implementing some new functionality: how should I do this? I think packages are intimidating developers out there (maybe I'm wrong) and part of the reason may be that it is very hard to decide what to put where and why. It seems like most of my packages start with an idea that becomes a *JMB_***procname** stored procedure. (All initial experiments are named with the prefix *JMB_* to let me know they are part of my playground.) As soon as the procedure becomes more than 100 lines long or contains code duplication or a related but different procedure suggests itself or needs to stash some persistent data, a package is

> magically born.
>
> Once spawned, packages often have a life of their own, they grow and mature and sometimes die or are subsumed by larger packages. I don't know if there is an idea here, but something that makes deciding what and how to start a package may help developers... I suppose the whole book is just that in a sense.

There is definitely an idea in there, but my perspective is somewhat simpler than what John probably had in mind: Get out of the habit of building standalone procedures and functions. Instead, start with a package! It is certainly the case that most complex programs eventually mutate into or are absorbed by packages. There is nothing wrong with that evolutionary process. You can, however, save yourself some trouble by creating a package to hold that seemingly simple and lonely procedure or function.

If you start with a package, several benefits accrue:

- You immediately work at a higher level of abstraction. You think of your single program as just one component of a whole range of related functionality. In the first implementation of the package you may not be aware of even one other program for this package, but you are allowing for the possibility of such programs.

- Related to the level of abstraction, you find yourself creating separate layers and partitions of functionality. Then, as you work on additional programs, you identify the appropriate package for that program. Lo and behold, you find things falling into place. You realize that everything has a place. Your code takes on an elegant sense of internal organization that makes it easier to use and understand.

- From the beginning, all calls to that program employ the dot notation necessary (and, I would argue, inevitable) to reference a package−based element. You don't have to go back later and change those calls or create another layer of code to support backward compatibility.

You will never regret the minuscule amount of extra time required to encapsulate your standalone programs inside a package.

PREVIOUS | HOME | NEXT
1.6 Building Packages | BOOK INDEX | 2.2 Using Effective Coding Style for Packages

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

2.2 Using Effective Coding Style for Packages

A package is a collection of PL/SQL elements, including data structures (from cursors to constants) and program units (procedures and functions). Packages are generally the most complicated and extended pieces of code PL/SQL developers will write. To make matters worse, the current array of PL/SQL development environments do not offer any tools for viewing and managing a package as a collection. A package is treated and presented no differently from a single function −− just a whole bunch of lines of source code.

As a result, it is up to you to design and write your package to make it as readable and maintainable as possible. There are two fundamental strategies you can employ to help meet this objective:

Strategy 1: Use all available techniques to make your code as clean, modular, and structured as possible.

Strategy 2: Come up with a consistent coding style and format for your packages −− and get people to follow that style.

Many of the other best practices covered in this chapter address the first strategy −− which is clearly the more important and difficult of the two. In this section, I suggest elements of a coding style for packages. It is absolutely critical that you adopt an effective coding style and employ it consistently. This style should be compatible, of course, with the style you use throughout your PL/SQL code. It should also, however, include components that reflect and support the structure and significance of the package.

The most basic elements of a package style are, first of all, no different from the style I encourage for all other kinds of PL/SQL code. These elements include:

- Use consistent indentation to reveal the logical flow of the program and to delineate the different sections of the PL/SQL program structure. Generally, this means that all executable statements are indented in from the BEGIN keyword, the body of a loop is indented within the LOOP and END LOOP keywords, and so on. Within a package, all specification declarations are indented between the IS and END keywords.

- Code all reserved words in the PL/SQL language in upper−case. Use lower−case for all application−specific identifiers. Generally, this is accomplished with hard−coded literals and the use of UPPER and LOWER. This guideline presents more of a challenge when applied to complex expressions passed to PLVgen as default values, as we'll see later.

- Use comments to add value to the code. Don't bother with comments that simply repeat what the code clearly states.

The style elements I find valuable particularly for packages include the following:

-

Use banners (specially formatted comment lines) to mark clearly the different groupings of package elements.

- Use end labels for the package and for all program units defined in the package body.

The best way to demonstrate these coding styles is to show you the template I use for package construction. I have been writing a lot of PL/SQL code in the past year and I found myself typing the same words and phrases over and over again. To improve my productivity and also the consistency of my code, I built a package called PLVgen to generate PL/SQL programs (see Chapter 15, *PLVvu: Viewing Source Code and Compile Errors*). Example 2.1 shows the basic template of a package generated with PLVgen.

> ***NOTE:*** The *PLVgen.pkg* procedure also generated the line numbers to go with the source code.

Example 2.1: A Generated Package Template

```
SQL> exec PLVgen.pkg('emp_maint');
  1 CREATE OR REPLACE PACKAGE emp_maint
  2 /*
  3 || Program: emp_maint
  4 ||  Author: Steven Feuerstein
  5 ||    File: emp_maint.SQL
  6 || Created: APR 13, 1996 18:56:59
  7 */
  8 /*HELP
  9 Add help text here...
 10 HELP*/
 11
 12 /*EXAMPLES
 13 Add help text here...
 14 EXAMPLES*/
 15
 16 IS
 17 /* Public Data Structures */
 18
 19 /* Public Programs */
 20
 21    PROCEDURE help (context_in IN VARCHAR2 := NULL);
 22
 23 END emp_maint;
 24 /
 25
 26 CREATE OR REPLACE PACKAGE BODY emp_maint
 27 IS
 28 /* Private Data Structures */
 29
 30 /* Private Programs */
 31
 32 /* Public Programs */
 33
 34    PROCEDURE help (context_in IN VARCHAR2 := NULL)
 35    IS
 36    BEGIN
 37       PLVhlp.show ('s:emp_maint', context_in);
 38    END help;
 39 END emp_maint;
 40 /
```

There are several features I would like to highlight in my package template:

Lines	Significance

2−7	A standard header for the package, showing the author, filename, and date created.
8−14	Stubs for help text. I have developed an architecture (and the PLVhlp package, described in Chapter 16, *PLVgen: Generating PL/SQL Programs*) to provide online help for PL/SQL programs. These comment blocks provide both inline code documentation and help text to users.
17−19	Banners to identify the two main kinds of elements that can appear in a package specification: data structures and program units.
21	Header for a procedure that delivers online help for this package. Of course, this should only be included if the online help package is being used.
23	The END statement with the package name appended.
28−32	Banners to identify the three kinds of elements that can appear in a package body: private data structures, program units, and the implementation of the public program units.
34−38	The implementation of the help procedure. Notice that the procedure uses an end label with the program name and is also indented in multiple steps from the overall package.

The **emp_maint** package shown in Example 2.1 contains the most important elements of a package's "look and feel." All elements declared in the specification are indented in from the package definition statement. They exist within the context of the package, and that relationship is made clear through the indentation. The same rule holds true in the package body (you can see this with the definition of the **help** procedure). The banner comment lines, on the other hand, are left−justified to match the margin of the package itself. I do this to make sure that these boundary markers stand out as you scan the code.

The banners identifying the different sections become very critical when the package is full of many different elements and runs to hundreds or thousands of lines. They also provide an internal guide during development. As you write a new package program, you may find that you need to create another private variable or private function. If you have the banners in place, you can easily perform a search and then drop this new element into its rightful spot. The alternative (throwing the code in at whatever point of the package you happen to be coding) results in a very chaotic package that is difficult to follow and maintain.

As I make clear in the way I created Example 2.1, you can use *PLVgen.pkg* to generate a package with this (or a modified) format.

2.2.1 Choosing the Order of Elements

As with the declaration sections of procedures and functions, you must (both in the package specification and body) declare all variables and data structures before you declare any program units. But what about the order of these program units themselves? As you can see from my banners, I always try to define all my private modules before any of my public modules. These are the building blocks used by the public programs. I group them together so they are easier to locate.

Is this ordering strictly necessary? Yes and no. Yes, you must define a private program before it is referenced by another program in the package (public or private). No, you do not have to group them together. You could instead define all private modules just before they are used by their public counterparts. This can make sense if the private program is only used by a single public program. If it is shared by many public programs (or other private ones, for that matter), then this placement does not accurately reflect its role in the package.

You can, by the way, place the definitions of the public program units anywhere in the package body (after the variable declarations) −− even *after* they are referenced by another program. How is this possible? Since their headers have already been established in the package specification, the PL/SQL compiler has all the information it needs to resolve the reference.

PREVIOUS HOME NEXT
2.1 Starting With Packages BOOK INDEX

2.3 Selecting Package Names

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

PREVIOUS
Chapter 2
Best Practices for Packages
NEXT

2.3 Selecting Package Names

Have you ever noticed that a package is never executed or referenced? The package is only a container for all the elements *inside* the package. In your code you will execute package−based procedures and functions. You will reference package−based constants, variables, cursors, and so on. Consequently, all references to package−based elements are accomplished with qualified notation: *package.element*. You should take this format into account when you name both the package and the elements within a package.

In this section I discuss the following aspects of naming package−based elements:

- Choosing appropriate and accurate names.
- Avoiding redundancy.
- Avoiding superfluous naming elements.

If you follow the advice in this section, you will design packages that are more easily used and understood by other developers.

2.3.1 Choosing Appropriate and Accurate Names

As a rule, developers are much too careless about the names they give to their packages and the elements inside those packages, (most importantly, procedures and functions.) There are two aspects to coming up with the right names for your code elements:

- The structure of the name should match the role that element plays in your code.
- The name should reflect what the element does in your code.

Have you ever thought about the *structure* of the names you choose? PL/SQL is a computer language. It is much simpler than human languages like Japanese or English, but it still has many of the same grammatical components, such as nouns and verbs. RAISE_APPLICATION_ERROR, for example, contains a verb (RAISE)−noun (APPLICATION_ERROR) combination, as in: "Raise this error." The built−in function, SUBSTR, is an example of a noun (SUBSTR), as in: "if the substring is NULL, then ask for a dollar amount."

PL/SQL on the other hand, is more complicated than human languages because you, the developer, get to make up words in the language as you go. You define new nouns and verbs every time you declare a variable or define a new program. This means that each and every one of us is, at least in part, responsible for the integrity of the PL/SQL language. Keep this in mind as you name your program elements. Let's apply this consideration to packages.

First of all, the name of the package should always be structured as a noun. The package itself does not *do* anything, so it cannot and should not be an action verb. The package name declares, as simply as possible, the contents of the package. If you are writing a package to analyze sales, the name of the package should be something like:

```
sales_analysis
```

and not either of these:

```
perform_sales_analysis
calculate_sales
```

It should also probably not be something as vague as "sales". There are many different aspects to sales; there would be no way to tell from the name that this package performs analyses on sales figures.

Beyond the package name itself, you must be very careful in your naming of elements within the package. A procedure is an executable statement, a command to the PL/SQL compiler. Consequently, the structure of the procedure name should be similar to a command:

```
Verb_Subject
```

as in:

```
Calculate_P_and_L
Display_Errors
Confirm_New_Entries
```

A function, on the other hand, is used like an expression in an executable statement. Because it returns, or represents, a value, the structure of a function name (as well as all constants and variables) should also be a noun:

```
Description_of_Returned_Value
```

as in:

```
Net_Profit
Company_Name
Number_of_Jobs
Earliest_Hire_Date
```

If I use the wrong grammatical structure for my names, they do not read properly.

2.3.2 Avoiding Redundancy

Keep in mind that when you reference a package element outside of the package you must use dot notation (*package.element*). As a result, you will want to avoid redundancy in your package and element names. For example, suppose I have a package named **`emp_maint`** for employee maintenance. One of the procedures in the package sets the employee salary.

Here is a redundant naming scheme:

```
PACKAGE emp_maint
IS
   PROCEDURE set_emp_sal;
END;
```

With this approach, I would then execute the procedure as follows:

```
emp_maint.set_emp_sal;
```

I do not need to mention **emp** again in the procedure name. The entire package is all about maintaining employees. That should be assumed in the names of all elements defined within the package. A more sensible approach would be:

```
PACKAGE emp_maint
IS
   PROCEDURE set_sal;
END;
```

With this new approach, I can then execute the procedure as follows:

```
emp_maint.set_sal;
```

In this way, I type less and the resulting code is more readable.

2.3.3 Avoiding Superfluous Naming Elements

I often recommend that you include as a suffix or prefix to an element name an abbreviation that indicates clearly the type of element. So whenever I declare a cursor, for example, I always append a suffix of "_cur" as shown in the example below:

```
DECLARE
    CURSOR emp_cur IS
       SELECT ...;
BEGIN
    FOR emp_rec IN emp_cur
    LOOP
        ...
    END LOOP;
END;
```

You can go overboard with these abbreviations and end up with names that are unwieldy and trip over themselves. The package name is one of those instances. I recommend that you do not append suffixes like "pkg" or "pak" to the names of packages. It will be clear enough from the way the packaged elements are referenced and used that they are defined within a package. Let's look at an example to illustrate the point.

Suppose I define my **emp_maint** package as follows:

```
PACKAGE emp_maint_pak
IS
   PROCEDURE set_sal;
END;
```

With this verbose approach, I then execute the procedure as follows:

```
emp_maint_pak.set_sal;
```

What do I gain by including the "pak" in the call, except to add to my typing? There can be no doubt at all that the **set_sal** procedure is defined within a package.

Similarly, I have worked at companies whose naming conventions dictate that whenever you create a procedure, you must preface the name with "pr" as in:

```
pr_calc_totals;
```

Function names must, of course, be prefaced with "fu" as in:

```
        v_totsal := fu_total_salary;
```

This is serious overkill; if you have conventions like these, you need to find a better balance between self−documentation, readability, and productivity.

Some readers may notice an inconsistency in my approach to using suffixes. I suggest that you do not include "pkg" in your package names. I do continue to recommend, on the other hand, that you use a suffix for cursors and records, such as **`emp_cur`** and **`obj_rec`**. Why not drop the suffix for all of these elements? After all, it is usually pretty clear when I refer to a cursor or record. The clearest justification has to do with avoiding name duplication within the same PL/SQL block scope. Package names must be unique within an Oracle account. Within a single package, however, you may well want to define records, cursors, PL/SQL tables, programmer−defined record TYPEs, and so on for, say, the **`emp`** entity. If you do not use standard suffixes (or prefixes), you will end up with a bewildering variety of names. Conventions based on the entity name −− such as **`emp`** or **`dept`** or **`orders`** −− offer the simplest and clearest way to distinguish between these different elements of the PL/SQL language.[1]

[1] Thanks to John Beresniewicz for this insight.

PREVIOUS | HOME | NEXT
2.2 Using Effective Coding Style for Packages | BOOK INDEX | 2.4 Organizing Package Source Code

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

2.4 Organizing Package Source Code

Most Oracle shops still rely on SQL*Plus to create and compile PL/SQL programs. This means that the source code resides in one or more operating system files. To avoid losing control of that source, you should adopt some simple conventions for the extensions of your files. The approach I have taken is shown in the table below.

File Extension	Description
procname.*sp*	The definition of a stored procedure.
funcname.*sf*	The definition of a stored function.
pkgname.*spp*	A stored package definition that contains the code for both the package specification and the package body.
pkgname.*sps*	The definition of a stored package specification only.
pkgname.*spb*	The definition of a stored package body only.
scriptname.*sql*	An anonymous PL/SQL block or a SQL*Plus script (SQL statement plus SQL*Plus commands).
procname.*wp*	The wrapped[2] definition of a stored procedure.
funcname.*wf*	The wrapped definition of a stored function.
pkgname.*wpp*	A stored package definition that contains the wrapped code for both the package specification and the package body.
pkgname.*wps*	The wrapped definition of a stored package specification only.
pkgname.*wpb*	The wrapped definition of a stored package body only.
scriptname.*sql*	An anonymous PL/SQL block or a SQL*Plus script (SQL statement plus SQL*Plus commands).

[2] As of PL/SQL Release 2.2, you can "wrap" your PL/SQL source code into an encrypted format. This format can be compiled into the database, but is not readable. Wrapped code is primarily of value to third−party vendors who provide PL/SQL−based applications, but are not interested in letting the competition see how they built their applications.

With your code separated and easily identified in this manner, you will be able to locate and maintain it more easily. You can fine−tune these extensions even more. For example, I often use the ".tab" extension for SQL*Plus Data Definition Language (DDL) scripts that create tables. The most important aspect of these naming conventions is the implied separation of package specification and body (**sps** and **spb**).

There are two advantages to creating and compiling specifications and bodies separately:

- *Minimize the need to recompile programs.* If you recompile a package specification, then any program that references an element in that package will need to be recompiled (its status in the USER_OBJECTS view is set to INVALID). If you recompile only the body, on the other hand, none of the programs calling that package's element are set to invalid. So if you change a package's body and not its specification, you should not recompile the specification −− which means that you should keep those elements of the package in different files.

- *Allow different packages to depend upon each other.* This codependency of packages is explored below.

2.4.1 Creating Codependent Packages

Codependency is not just an issue for psychologists and the self−help publishing industry. It can also rear its ugly head with PL/SQL packages. Suppose that package A calls a program in package B, and that package B calls a program in package A. These two packages depend on each other. How can one be defined before the other? How can either or both of these packages be made to compile? Simple: define all specifications and then define all bodies.

I ran into this codependency problem just before I was to give a class on packages. I planned to give out a copy of PL/Vision Lite and started work on an installation disk. Most of my packages were stored in **`spp`** files. The package specification and body were, in other words, stored in the same file. So I placed calls to execute all of these scripts in my installation file and tested the process in a fresh (of PL/Vision) Oracle account. The installation failed miserably and I couldn't understand the problem. I was able to compile any of these individual packages in my existing PL/Vision account (PLV) without any difficulty.

Suddenly, I realized the problem: when I compiled a package in my PLV account, it could reference the other packages that already existed. The package would, as a result, compile successfully. In an account with no preexisting PL/Vision code, however, when I tried to compile the *p* package body (a very basic package used by almost every other package in PL/Vision), it could not find the PLVprs package, which was not yet defined because it referenced the *p* package (among others). PLVprs was compiled later in the installation script.

For about five minutes I despaired. Had I constructed a product that wasn't even capable of installing? Then I came to my senses. The package specification and body do not have to be compiled together. And if the *p* package relies on the PLVprs package, it only requires the package *specification* for PLVprs to be in place. The PL/SQL compiler only needs to know, in other words, that the *p.l* procedure is calling **`PLVprs.display_wrap`** properly −− and that information is contained in the specification. I didn't have a faulty product. I had a faulty installation script!

Take a look at the **`PLVinst.sql`** file on your disk. This SQL*Plus script now installs the PL/Vision packages in a more sensible fashion. You will see that there are two phases to the installation of PL/Vision: first, all the package specifications are created, and then package body creation scripts are executed. In this way, I can leverage all the different, handy elements of PL/Vision in other parts of the product.

I learned from this experience that I should *always* separate the scripts for the creation of the package specification and body, even if the packages are very short and simple.

> ***NOTE:*** This separation of specification and body will not work in all codependent situations. If the specification of package A references an element in package B, and the specification of package B references an element in package A, you will *not* be able to compile these two packages. Each specification requires the other specification to be previously defined, which simply isn't possible.

PREVIOUS | HOME | NEXT
2.3 Selecting Package Names | BOOK INDEX | 2.5 Constructing the Optimal Interface to Your Package

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8*i* Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

2.5 Constructing the Optimal Interface to Your Package

The interface to your package consists of the names of the public elements and, in the cases of procedures, functions, and cursors, the parameter lists and RETURN datatypes.

The interface of your package is, in the broadest sense, affected by almost every best practice in this chapter. There are several recommendations that apply more narrowly to the interface (particularly the parameter lists); those are covered in the following sections. Before delving into those particulars, however, I need to step way back and address a philosophical issue of package design that is fundamental to making your packages as useful and usable as possible: the need to see other developers as users, and the impact that has on your package design.

2.5.1 Seeing Developers as Users

The vast majority of the packages I build are utilities and components for other developers. PL/SQL developers are, in other words, my users. Now you are probably aware that developers generally don't have many good things to say about *their* users. "Those users" are always modifying their requirements and are incredibly lazy; they change their minds on a daily or hourly basis; and they could care less about your resource issues. Why (I hear again and again), if it's not absolutely obvious and easy to navigate through an application, "those users" complain and complain −− and sometimes refuse outright to use your application. Ungrateful wretches.

Well, I have news for you: developers as users are no different from end users as users. They (and I include myself fully in this characterization) have a very low tolerance for rigmarole and wasted motion. They will use a utility I offer them only if they can understand it intuitively and put it to use instantly. I believe that attitude is appropriate. Our software should be smart and easy to use. Ease of use is, however, just the first requirement. Developers also want total flexibility. They don't want to have to do things *my* way just because I "wrote the book" on PL/SQL and wrote the package, too.

How do I know that my users will be so fussy? Because for the last year I have been struggling to use my own software and have constantly needed more flexibility in order to make that software useful to *me*. I have watched PL/Vision grow both in number of packages and internal complexity of those packages. In the process I have taught myself several techniques that I explore in this section. You'll see these over and over again in PL/Vision (especially the building into my packages of toggles and windows).

As far as I am concerned, it is always worth it for me to spend more time in the design and development of my code if it results in programs that are smarter and therefore easier to use. I don't necessarily believe that we should follow the tenet that the user is always right, but we should generally take a more respectful view towards our users −− especially the developer ones. In almost every case, they have a legitimate gripe. Computers and the software installed on them are not nearly as intuitive and accessible as they should be. Do your part in your design of PL/SQL packages to improve that situation.

2.5.2 Making Your Programs Case−Insensitive

Make sure users don't trip over senseless obstacles on the path to using your programs. A common source of frustration is the requirement that arguments to a program be in one case or another (usually upper or lower).

Consider the following program:

```
CREATE OR REPLACE FUNCTION twice
   (string_in IN VARCHAR2, action_in IN VARCHAR2)
RETURN VARCHAR2
IS
BEGIN
   IF action_in = 'UL'
   THEN
      RETURN (UPPER (string_in) || LOWER (string_in));

   ELSIF action_in = 'LU'
   THEN
      RETURN (LOWER (string_in) || UPPER (string_in));

   ELSIF action_in = 'N'
   THEN
      RETURN string_in || string_in;
   END IF;
END twice;
/
```

This function (which is not even defined inside a package; this best practice, like many others, applies equally to standalone modules as well) returns a string concatenated to itself, with some optional case−conversion action. You pass it UL or LU or N, and the appropriate transformation of the string is made. But what if someone calls **`twice`** as follows?

```
bigger_string := twice (smaller_string, 'ul');
```

The PL/SQL runtime engine will actually raise an exception:

```
ORA−06503: PL/SQL: Function returned without value
```

This is very poor behavior by the function. If developers are going to reuse your code, they need to get dependable results from it. It should never raise the −6503 exception or, in general, any exception at all. Instead it should return a value that indicates a problem whenever possible. Beyond that, users of **`twice`** should not have to care about the case of the string they pass for the action code. Your program should automatically force all entries of this kind (action codes and types) to either lower− or upper−case and then proceed from there. The best way to do this, I have found, is to declare a local variable that accepts as a default value the case−converted argument. This technique is shown in the following example:

```
CREATE OR REPLACE FUNCTION twice
   (string_in IN VARCHAR2, action_in IN VARCHAR2)
RETURN VARCHAR2
IS
   v_action VARCHAR2(10) := UPPER (action_in);
BEGIN
```

With this approach, you never reference the parameter **`action_in`** in the function. Instead, you work with **`v_action`** in the body of the function, and case is never an issue. This may seem like a small issue, but it can loom large when a developer is under lots of pressure, wants to use your code, and fails the first three times because the case is wrong or the literal used for the action code is in some way erroneous.

2.5.3 Avoiding Need for User to Know and Pass Literals

If you follow the advice of the previous section, a user of the **`twice`** function will be able to enter **`UL`**, **`ul`**, **`uL`**, **`LU`**, **`lu`**, **`N`**, or **`n,`** and the program will react properly. But in an ideal world, users wouldn't even have to know about these literal values −− and they certainly wouldn't have to place such literals in their program.

What if someone decides to change the particular constants used by **`twice`** to recognize different kinds of actions?

Removing literals from your programs for these kinds of arguments is made particularly easy using packages. There are two ways to achieve this objective:

1. Provide separate programs for each of the different actions.
2. Provide package−based constants that hide the action values and offer a named element in their places.

Creating different program specifications for each action is practical only if there is a fixed number of actions. I use this approach in PLVexc; there, I convert a handler action argument in the handle procedure to four different procedures:

```
PROCEDURE stop;
PROCEDURE recNstop;
PROCEDURE go;
PROCEDURE recNgo;
```

This proliferation of procedures is not desirable if you think that the set of possible actions might change or expand. Also, in some cases, you really want to stick with one overloaded name and not bewilder the user with a whole suite of programs. For example, if I took the PLVexc approach with the **`twice`** function I would end up with:

```
FUNCTION twiceUL ...;
FUNCTION twiceLU ...;
FUNCTION twiceN ...;
```

As an alternative, I could define a set of constants, one for each action, as shown in the package specification below:

```
CREATE OR REPLACE PACKAGE dup
IS
   lu CONSTANT VARCHAR2(1) := 'A';
   ul CONSTANT VARCHAR2(1) := 'B';
   n  CONSTANT VARCHAR2(1) := 'X';
   FUNCTION stg
      (stg_in IN VARCHAR2,
       action_in IN VARCHAR2 := n,
       num_in IN INTEGER := 1)
   RETURN VARCHAR2;
END dup;
/
```

Notice that the **`twice`** function has now been replaced with **`dup.stg`**, a more generalized string−duplication function. The default action for a call to **`dup.stg`** is now the constant **`n`**, rather than the literal **`N`**. So if I want to duplicate a string 10 times and convert it to UPPER−lower format, I would call **`dup.stg`** as follows:

```
v_bigone := dup.stg (v_ittybitty, dup.ul, 10);
```

Sure, I have to know the names of the constants, but I will be informed *at compile time* if I got it wrong. This is a very important distinction from the mysterious, hard−to−trace error I will receive if I simply pass the wrong literal value. The compiler could care less about if I pass the right literal. There are no right or wrong literal values as far as the compiler is concerned; my code must therefore be qualitatively more robust to handle this error gracefully.

The other advantage to the package constant approach is that you can change the underlying values without affecting anyone's use of *dup.stg*. As you can see in the package specification, I deliberately gave these constants values that did not match the previous values. This will flush out old usages and force compliance with the use of the constants, rather than the literals. You don't have to do this, and may not be able to for reasons of backward compliance, but it is a useful technique to keep in mind.

PREVIOUS | HOME | NEXT
2.4 Organizing Package Source Code | BOOK INDEX | 2.6 Building Flexibility Into Your Packages

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8*i* Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

2.6 Building Flexibility Into Your Packages

Who is going to argue with this one? Sure, we want our code to be flexible, in a practical sort of way. It is quite another thing to internalize this issue in the context of packages and figure out how to take full advantage of the package structure to offer maximum flexibility.

If a program is going to be widely reusable, it should be able to adapt to different circumstances to meet different needs. It is easy to talk about flexibility. I have found that when it comes to packages there are two basic ways to be flexible when writing programs for others to use:

1. Offer lots of parameters in the parameter lists of the package's functions and procedures. This is the traditional, well−worn path.
2. Provide toggles or on−off switches, distinct from the main programs of the package, which modify the behavior of those programs. This approach takes advantage of the package structure to offer a new way of doing things.

It certainly makes sense to offer arguments in a packaged program unit to improve the flexibility of that *individual* program. Consider the **display** procedure of the PLVtab package, whose header is shown below:

```
PROCEDURE display
 (table_in IN date_table,
  end_row_in IN INTEGER,
  header_in IN VARCHAR2 := NULL,
  start_row_in IN INTEGER := 1,
  failure_threshold_in IN INTEGER := 0,
  increment_in IN INTEGER := +1);
```

This procedure has a whole bunch of parameters, and every one of them makes sense for the display of a particular table. Do you want to provide a header different from the default "Contents of Table"? Provide an argument to the **header_in** parameter. Do you want to display every fifth row? Pass in 5 for **increment_in**. Sensible defaults are, on the other hand, provided for almost every parameter, so you only need to provide values if you want to override these defaults.

What do you do, however, when you want to provide flexibility that affects the behavior of the package *as a whole*, not just for a particular program? What if you want to alter the configuration of a package for an entire session? Furthermore, what if you want to change the behavior of your package without changing the application code that uses your package?

Again, let's take a look at the PLVtab package for an illustration of this situation. PLVtab is a low−level package used throughout PL/Vision under many different circumstances. In some situations, I wanted to be able to display the row number in which the data is found. In other scenarios, I did not want any header to display before the table data was shown. Finally, I thought it would be useful to be able to see a translation of a blank line (i.e., does the line contain actual blanks or is it NULL or is it some other non−printing character?).

I could simply have kept adding new parameters to the **display** procedure (actually, adding new parameters to the *nine* different overloaded versions of **display**) to handle all of these variations. I would then end up with a header for **display** that looked like this:

```
PROCEDURE display
 (table_in IN date_table,
  end_row_in IN INTEGER,
  header_in IN VARCHAR2 := NULL,
  start_row_in IN INTEGER := 1,
  failure_threshold_in IN INTEGER := 0,
  increment_in IN INTEGER := +1,
  use_header_in IN BOOLEAN := TRUE,
  show_rownums_in IN BOOLEAN := FALSE,
  show_blanks_in IN BOOLEAN := FALSE);
```

I don't know about you, but when I look at programs with more than six or seven parameters, my head starts to spin. Human brains are not, according to numerous studies, well equipped to deal with more than seven items at once. You could contend that these additional parameters increase the flexibility of the **display** procedure. I would argue, instead, that these additional parameters doom the *PLVtab.display* procedure to the dustbin of history. Few people will be brave enough to try to use it, particularly if they have to modify the default values of those trailing arguments.

Fortunately, certain aspects of the PL/SQL package provide an alternative to turning your procedure into a sinking ship (weighed down by too many parameters): you can build *toggles* into your packages that allow a user of the package to change the behavior of the utility with the "flip of a switch."

2.6.1 Toggling Package Behavior

You will find toggles appearing throughout the PL/Vision packages. A toggle is a set of three programs: two procedures that allow you to turn a feature on or off, and a function to tell you the current status (on or off). The liberal application of toggles can transform the usability of your packages. The easiest way to teach you this technique is to show you how I use it in PL/Vision.

In PLVtab, I did not add a use header argument to the nine display procedures. Instead, I offer a toggle or on−off switch using these three programs:

```
PROCEDURE showhdr;
PROCEDURE noshowhdr;
FUNCTION showing_header RETURN BOOLEAN;
```

The *showhdr* program turns on the showing of the header. The *noshowhdr* turns off the display of the header. The **showing_header** function returns TRUE if the header is currently set to be shown. These three programs contain very little. They simply maintain and access a private global variable, as shown below:

```
v_display_header BOOLEAN := TRUE;

PROCEDURE showhdr IS
BEGIN
  v_display_header := TRUE;
END;

PROCEDURE noshowhdr IS
BEGIN
  v_display_header := FALSE;
END;

FUNCTION showing_header RETURN BOOLEAN IS
BEGIN
  RETURN v_display_header;
END;
```

How do I put these toggles to use? Suppose that in most cases in my application I wish to hide the header. Since the default value for **`v_display_header`** is TRUE, I must turn off the display of the header at the start of my session. I could do that in my SQL*Plus *login.sql* script as follows:

```
exec PLVtab.noshowhdr;
```

Alternatively, if I am using PLVtab within an Oracle Developer/2000 Oracle Forms screen, I might place this call inside the When−New−Form−Instance trigger:

```
PLVtab.noshowhdr;
```

If, at some point in my application, I want to display a table with its header, I can temporarily override the default setting as follows:

```
PLVtab.showhdr;
PLVtab.display (selected_comp_tab, v_tot_selected, 'Selected Companies');
PLVtab.noshowhdr;
```

2.6.2 Toggles for Code Generation

Now consider the PLVgen package. PLVgen generates many different kinds of PL/SQL code elements. I used PLVgen earlier in this chapter, in fact, to generate a template for a package to show you a recommended format for packages. Since there are many variations in the way you might want to generate your code, PLVgen contains *nine* toggles that affect the appearance and contents of the generated code. It is totally impractical to add nine arguments to every one of my two dozen code generator procedures. It is very practical, on the other hand, to offer you the toggles to set, in effect, your own standard approach to generating PL/SQL code.

To offer just two examples, the default settings for this PL/SQL code generator package are to include auto−generated comments and to *not* include a standard header for program units. I can, however, change those defaults with calls to the appropriate toggles as shown below:

```
SQL> exec PLVgen.usehdr
SQL> exec PLVgen.nousecmnt
```

These toggle programs set the values of private global variables in the package. These variables are then referenced to determine the behavior of the package. If you look inside the *PLVgen.spb* file (the package body), you will also see instances where I call PLVgen toggles from *inside* some code generators so that I can achieve just the behavior I desire. Consider the *helptext* package below.

```
PROCEDURE helptext (context_in IN VARCHAR2 := PLVhlp.c_main)
IS
   v_save BOOLEAN := using_hlp;
BEGIN
   /* Turn off help, but then restore if necessary. */
   usehlp;
   put_help (context_in);
   IF NOT v_save THEN nousehlp; END IF;
END;
```

This procedure generates a comment stub for help text. It calls the private **`put_help`** procedure to construct that stub. If, however, the user has previously turned off help text generation, this program will do nothing. So the *helptext* procedure saves the current setting for using help text, turns that toggle on, generates the help text, and then turns the help text setting off, if that was the previous setting.

2.6.3 Changing Package Behavior Without Changing the Application

One of the most exciting benefits of package toggles is that they allow a user of the package to modify the behavior of the package without changing any application code that calls the package element. Let's start with an example to explain that complicated statement, and then I will generalize.

Suppose you want to use the PLVlog package to keep track of any changes made to the **`emp`** table. To do this, you will make calls to **`PLVlog.put_line`** in the appropriate database triggers. Here is an example of one such call:

```
PLVlog.put_line ('insert', :new.empno, :new.empname);
```

This request logs the fact that I am inserting a new employee with the specified ID number and name. The log mechanism also records the current user, as well as date and time. This code works just fine and goes into production. Then my company, in the true enterprising spirit of the 1980s and 1990s, purchases a company ten times its own size (which means no more raises for me, since they must now use all their money to pay off interest on the assumed debt). Suddenly, I must add 25,000 employees to my **`emp`** table. My log table cannot handle this volume of data in its current structure. Furthermore, I don't even really want an audit of this activity. The data should just be "slammed" in and used as a new baseline for corporate employment.

If I did not have a toggle in PLVlog, what would I have to do to turn off logging? I can think of two options:

1. Go into each trigger and comment out the call to PLVlog.

2. Disable all triggers on the **`emp`** table.

The first approach should make you shudder. You never, ever want to have to go into production code and make such temporary changes −− even (especially?) if those changes are not in a program per se, but are instead a part of the data structures. The second solution is not much better. You have to write a script to disable all the triggers and then this code is disabled for *all* users of the application, not just the single process, which is going to batch load all of the new employees. So if you disable triggers, you have to deny access to the application by other users. Two very ugly prospects.

If, on the other hand, you have a PL/Vision toggle in place, this situation does not cause you any grief at all. Before you start the process to load the employees (let's call it session A), you simply execute this command:

```
SQL> exec PLVlog.turn_off
```

Now, whenever the database trigger calls **`PLVlog.put_line`** for DML initiated by session A, nothing happens. Why? Because the first thing **`put_line`** does is check the value of the private toggle variable (by calling the toggle function) as shown below:

```
IF logging OR override_in
THEN
   ... log the information ...
END IF;
```

You didn't have to change your program and you didn't have to modify the state of your database. From *outside* the package, you call the toggle program to reach *inside* the package and change the way the package will behave. This ability to leave your own code intact comes in particularly handy not only for special exceptions but also for testing, as I explore below.

2.6.3.1 The test/debug cycle in PL/SQL

A common debug and test cycle in PL/SQL shops goes like this:

1. You identify incorrect behavior in your program.

2. Unable to understand the cause of the behavior, you place numerous calls to *DBMS_OUTPUT.PUT_LINE* (or, with your purchase of this book, PL/Vision's much more friendly *p.l* procedure) and other kinds of tracing lines of code so that you can see what is going on.

3. You analyze the output, track down the problem, and fix it.

4. You finally decide that all the bugs are gone.

5. You notify your manager that the application is ready to go. Excitement mounts. Other organizations are told to start moving the code from test to production. Suddenly, you break out in a cold sweat and tell your bewildered manager to "hold off a minute."

6. You forgot about all that debugging code you littered into your application. It can't go into production like that. You have to go back into the program to comment out or outright remove all that trace code. No problem, you tell yourself. Easy to do...but there *could* be a problem. After all, any time you touch the code, you can break it. After any changes of any kind to your code, you really should retest.

7. So you have to go back to your manager and ask for more time to make sure everything really is all right. Not a pleasant situation in which to find yourself.

If, on the other hand, you used packages with toggles to trace your debugging activity (such as PLVtrc and even the lower−level *p* package), you would not have to worry about any of that. You could keep your code intact and simply issue a call to the appropriate package toggle to turn off any superfluous activity, as in:

```
SQL> exec PLVtrc.turn_off
SQL> exec p.turn_off
```

Of course, you can do more with toggles than simply turn functionality on and off. Remember that logging capability I built into my **`emp`** table triggers? Suppose that I want to write my log information to an operating system file instead of to a database table. That is a pretty major change in how the log will work, and a daunting task if the log mechanism is designed poorly. Yet with PL/Vision it requires no change at all to the database triggers. The call to PLV**`log.put_line`** remains exactly the same. Instead of modifying that application's code, I can simply redirect the output of the logging package with a call to the appropriate procedure as follows:

```
SQL> exec PLVlog.to_file ('log.dat');
```

and then all subsequent calls to PLV**`log.put_line`** for that particular Oracle session will write the information to the *log.dat* file on the server.

In my experience, package toggles make an enormous difference in the flexibility and usability of my packages. You can never add too many toggles. Just make sure that the default setting is the value that's normally desired. Then only those people who need flexibility in that particular fashion ever need to bother with the toggle. You can always add toggles later; it is generally not the kind of thing you have to plan in advance. This is particularly true if you have been aggressive in modularizing your package body code. If you have religiously avoided code redundancy and repetition (get it?), there will usually be just one place you have to apply the toggle to achieve a new level of flexibility.

I cannot overemphasize the importance of toggles in your packages. They are an essential element in transforming your package from a handy utility into a robust, flexible component or what is, in essence, a product.

To paraphrase an over−paraphrased saying: "If you toggle your package, they will use it."

PREVIOUS 2.5 Constructing the Optimal Interface to Your Package

HOME BOOK INDEX

NEXT 2.7 Building Windows Into Your Packages

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

2.7 Building Windows Into Your Packages

A special kind of toggle can be used to provide what I call a window into a package. This window allows a restricted view into the inner workings of a package, which can be critical to making the package usable in a complex, multilayered application.

As I've explained in Chapter 1, *PL/SQL Packages*, packages are broken up into the specification and the body. The specification defines those elements that can be called from outside of the package (the *public* elements). The body contains the implementation of public elements and also of *private* elements (those elements that can only be referenced inside the body of the package). This dichotomy allows us to hide quite securely implementation details that users need not be aware of in order to make use of the package.

This "information hiding" aspect of packages is a great feature −− until a developer needs to know what is going on inside the package. The black box in this case can become a hindrance. For example, I built a package called PLVdyn (which stands for "PL/Vision DYNamic SQL") to make it easier for developers to use the built−in DBMS_SQL package. PLVdyn lets the user parse and execute dynamically constructed SQL and PL/SQL statements without fussing with all the details inherent in the built−in package.

With PLVdyn, you construct a SQL statement and pass that string to a PLVdyn program for parsing or execution. It's a big time−saver, but it also implies a loss of some control. You trust PLVdyn to do the right thing −− and it does. The question that is more likely in need of an answer: what is your code passing to PLVdyn?

The code we write to construct the dynamic SQL statement is often complicated. The PL/Vision packages themselves make extensive use of PLVdyn. As I tested PLVdyn, I often found that I wanted to see the SQL statement that PLVdyn was executing, so I could verify that my *calling* program (in PLVio or PLVlog or...) had put the SQL together properly. This was not, conceptually, a difficult problem. I could simply place calls to DBMS_OUTPUT *before* each of my calls to PLVdyn modules. In this way, I would not have to change PLVdyn (it is not, after all, the fault of PLVdyn −− or its author! −− that I wasn't sure what my code was doing). With this approach, if I used *PLVdyn.ddl* to execute a DDL statement, I could simply preface it with a call to *p.l* (the PL/Vision version of DBMS_OUTPUT) as follows:

```
p.l (ddl_statement);
PLVdyn.ddl (ddl_statement);
```

For all its simplicity, there is a key drawback to this solution: I would have to add calls to *p.l* in all the places I call a PLVdyn program. This meant going back to existing programs to make changes. I would have to remember to add this call whenever I used PLVdyn or felt the need to trace my activity. In either case, it involved changes to my code. Such changes invite misspellings and logical bugs.

This weakness, combined with the need to see what PLVdyn is doing almost caused me to abandon PLVdyn. Rather than use the package, developers would cannibalize it for the parts that seem useful. Or they would simply ignore this package−based solution and write all of their dynamic SQL directly into their programs. The result? Applications that do not reuse prebuilt code, but instead create maintenance and enhancement nightmares.

2.7.1 Centralizing the View Mechanism

A far superior approach would allow users to view the string they passed to PLVdyn *without* changing any of their own code. This view mechanism would be sophisticated enough to handle any number of different scenarios for SQL statement output, such as very long strings. The way to implement this approach successfully is to build the viewing feature directly into the PLVdyn package itself.

With the trace implemented inside PLVdyn, I can avoid modifying my own code when the output from that trace is needed. Instead, I can simply call a program in the PLVdyn package to tell it turn on the trace. I can then view the output until it is no longer needed and call a program to direct the package to turn off the trace. This sequence of commands is illustrated below, along with the toggle, a call to the PLVcmt which turns off commit processing. I want to run a test of my program to shift employees; I want to check my dynamic SQL without actually committing any possible mistakes.

```
SQL> exec PLVdyn.showsql
SQL> exec PLVcmt.turn_off
SQL> @test_move_emps
PLVdyn: INSERT INTO emp VALUES (1506, 1105, 'SMITH')
PLVdyn: UPDATE emp SET sal = 150000
PLVdyn: UPDATE emp SET hiredate = SYSDATE
```

I execute these steps, look over the trace, and decide that this all looks good. I then turn on commit processing, turn off the SQL trace, and run the program. All without making a single change to the **`move_emps`** program.

```
SQL> exec PLVdyn.noshowsql
SQL> exec PLVcmt.turn_on
SQL> @move_emps
```

By incorporating the trace into PLVdyn, I can't deny that I make my own job that much more difficult. I have to write the code for the trace and then figure out how best to implement it comprehensively for all PLVdyn modules. Yet once I have provided this feature, it is available for all users of PLVdyn. This kind of tradeoff (author effort vs. user ease of use) is always worthwhile in my view.

There are two aspects to keep in mind when building a trace or window into a package:

1. You need to provide the programmatic interface so that a developer can turn on/off the trace. This interface is a typical PL/Vision toggle and will usually take the same form in any package. As a result, it is a prime candidate for generation with the PLVgen package (see the **`toggle`** and **`gas`** procedures in Chapter 15).

2. You need to implement the trace carefully inside your package. What information will you provide? What mechanism will you use to display the trace? DBMS_OUTPUT or the *p* package or maybe even the PLVlog package? And, most importantly, *where* will you put the trace in the package so that you can minimize the number of different places it will appear? Remember: you want to avoid code redundancy. If you were aggressive about modularizing your package body, you should be able to identify a few programs or maybe even just one program (when you wish upon a star...) in which the trace can be implemented, but will then be used by all programs in the package. This process is explored in the next section.

2.7.2 Designing the Window Interface

The first step in installing a window in a package is to design the interface for the window, also referenced in this section as a trace. To do this, I must ask and answer these questions:

How should the user ask to turn the trace on and off?

- What other information can I provide to or ask from the user?

The easiest way for developers to specify their desires is to call a procedure. The first inclination might be to build a single procedure that accepts actions such as ON or TRACE vs. OFF or NO_TRACE as a single parameter. The header for such a procedure would look like this:

```
PROCEDURE set_trace (onoff_in IN VARCHAR2);
```

The developer would then call **`set_trace`** in SQL*Plus or another execution environment as follows:

```
SQL> exec PLVdyn.set_trace ('ON');
SQL> exec PLVdyn.set_trace ('OFF');
```

The problem with this approach is that the developer must then know what value to pass to the procedure to achieve the proper effect. Is it ON or YES? Is the value case−insensitive? Why, I ask myself in this situation, should a developer have to worry about such things? Even the seemingly clear TRUE/FALSE Boolean values are open to interpretation. If you generally do not want the trace in action, then TRUE should mean "keep it off." If you are often interested in the output of the trace, you would most naturally conclude that TRUE means "show the trace."

A completely different technique is to provide two different programs to turn the trace on and off. The names of the programs themselves would make it very clear what they did. You don't have to worry about getting a literal value wrong. If you type in the wrong program name, the runtime engine will inform you immediately of the error.

I can employ a very generic naming convention for this pair of on/off procedures as follows:

```
PROCEDURE turn_on;
PROCEDURE turn_off;
```

As an alternative, I could use names that describe the type of trace being provided. This is especially important when more than one trace is provided in the same package.

In PLVdyn, I opted for the less generic style and so provide these two procedures in the package specification:

```
PROCEDURE showsql;
PROCEDURE noshowsql;
```

With these programs in place, I could turn on my trace in SQL*Plus as follows (*ssoo.sql* is a PL/Vision script that sets SERVEROUTPUT to ON, enabling the DBMS_OUTPUT package in SQL*Plus):

```
SQL> @ssoo
SQL> execute PLVdyn.showsql;
```

The third program of the package trace is a function that lets me know the current status of the PLVdyn trace facility. In the PLVdyn package, I offer the **`showing_sql`** function. This program returns TRUE if the trace is turned on, FALSE otherwise:

```
FUNCTION showing_sql RETURN BOOLEAN;
```

The PLVdyn uses this function (shown later in this section) when trying to determine whether or not the SQL should be shown. Even the package body respects the interface of the toggle and uses the function instead of a direct reference to the private variable.

The **showing_sql** function also provides a sense of completeness to the interface for the trace facility. A developer using PLVdyn can remain within the API of the package to obtain all the information she needs to use the trace and get the most out of the package.

The full implementation of the trace facility in PLVdyn even goes a bit further than you have seen so far. The header for the *showsql* procedure is:

```
PROCEDURE showsql (start_with_in IN VARCHAR2 := NULL);
```

You can, in other words, provide a string to *showsql* to indicate the point in the SQL from which you want to view the text. You could ask to see, for example, everything after the WHERE keyword by calling *showsql* as follows:

```
SQL> exec PLVdyn.showsql ('where');
```

This additional flexibility can come in handy when you don't want to have to read your way through a long, complicated SQL statement. It's quite easy to provide additional functionality to a package window once it has been put in place. My first implementation of **showsql** did not support this "start with" argument nor did it display long strings very gracefully. I was able to add all of this functionality incrementally as I identified the need.

2.7.3 Implementing the Window

Now that the interface to the window has been defined, I need to implement the code that will fill that window with data. One of the biggest challenges in crafting a trace facility in a package like PLVdyn is to figure out where to put the trace. PLVdyn is a big package, offering many different high−level operators to perform dynamic SQL.

I did not want to have to add calls to DBMS_OUTPUT all over the package. That would make it more difficult to maintain and enhance. So I analyzed the way the package (and dynamic SQL) works and found my attention drawn back continually to the **open_and_parse** function. Before you can execute a SQL statement, you have to open a cursor and then parse the SQL.

The **open_and_parse** function was one of the first programs I created in PLVdyn, and it is used by all other programs before they move on to their specific dynamic tasks. As a result, **open_and_parse** acts as a kind of gateway into the rest of the package. I reasoned, therefore, that if I added the trace capability to **open_and_parse**, I could then make the trace available to the entire package. Now that's a payoff from earlier modularization! Here is the body of **open_and_parse**:

```
FUNCTION open_and_parse (string_in IN VARCHAR2,
    mode_in IN INTEGER := DBMS_SQL.NATIVE) RETURN INTEGER
IS
   cur INTEGER := DBMS_SQL.OPEN_CURSOR;
BEGIN
   display_dynamic_sql (string_in);
   DBMS_SQL.PARSE (cur, string_in, mode_in);
   RETURN cur;
END;
```

As you can see, the **display_dynamic_sql** procedure intercepts the string that is going to be parsed by the built−in PARSE procedure. A simplified version of the **display** program is shown below:

```
PROCEDURE display_dynamic_sql (string_in IN VARCHAR2)
IS
BEGIN
   IF showing_sql
   THEN
      PLVprs.display_wrap ('PLVdyn: ' || v_string, 60);
   END IF;
END;
```

Notice that **`display_dynamic_sql`** only displays information when **`showing_sql`** returns TRUE. It also takes advantage of the **`PLVprs.display_wrap`** procedure to show long SQL statements in paragraph form wrapped at a line size of 60 columns.

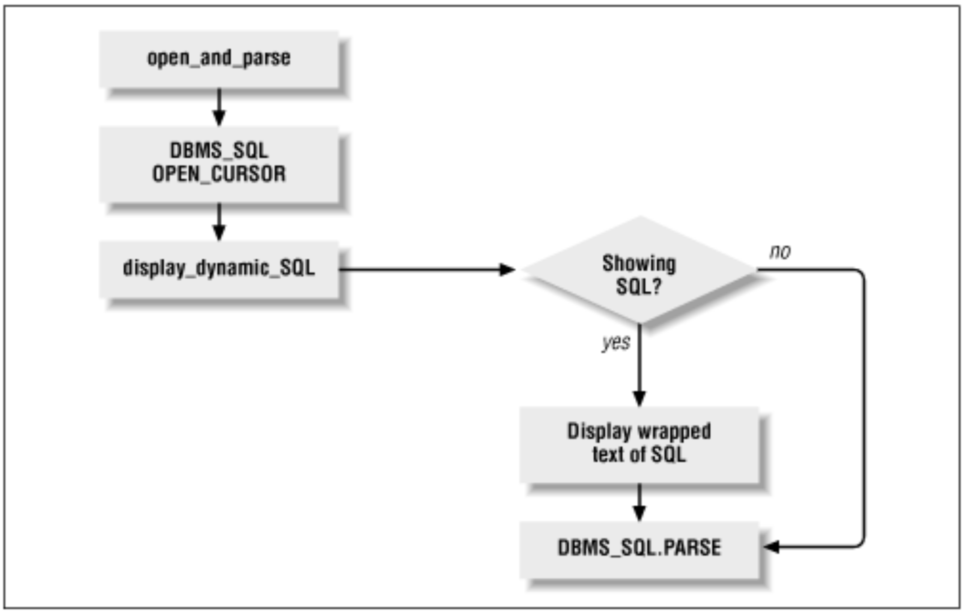
The **`open_and_parse`** program is called six times in PLVdyn. Actually, as I wrote this section, I had been thinking that the count would be even higher. It turns out that any of the programs that call **`open_and_parse`** are, in turn, called by other PLVdyn modules, keeping the direct references to **`open_and_parse`** from exploding. The **`display_dynamic_sql`** program, on the other hand, is called just once. When I want to upgrade or change the functionality of my trace, I can go to this one program and make all the changes.

The way I was able to implement the trace in PLVdyn is a best−case scenario. The requirement in dynamic SQL to parse your SQL statement, combined with my initial modularization and reuse of **`open_and_parse`**, offered an easy way to put the trace in place. In other PL/Vision packages and your own as well, you may need to include calls to your trace display mechanism more than once. That's fine, as long as you do create a separate procedure to display the information (do not just call DBMS_OUTPUT.PUT_LINE directly in your package) and as long as you minimize the number of repetitions of the program.

2.7.4 Summarizing the Window Technique

The trace facility of PLVdyn illustrates some important principles of both generic package structure and high−quality reusable code (see Figure 2.1). First, the public−private nature of the package allows me to construct a window into PLVdyn. This window offers a very controlled glimpse into the interior of the package. I let developers view the dynamic SQL string, but they can't look at or do anything else. This level of control allows Oracle to give us all of those wonderful built−in packages like DBMS_SQL and DBMS_PIPE. And it lets developers provide reusable PL/SQL components to other developers without fearing corruption of internal data structures.

Figure 2.1: Window/trace in PLVdyn



The three elements of the interface to the window are:

1.

A procedure to open the window and turn on the trace

2. A procedure to close the window and turn off the trace

3. A function that returns TRUE if the window is open, FALSE otherwise

Your package can have more than one window, in which case you will want to have a distinct triumvirate of programs for each trace (multiple toggles, in other words). If you have a number of different kinds of windows, you may also want to build a master switch that turns on and off all of the windows at once. The PLVgen **`usemin`** and **`usemax`** procedures are good examples of this meta−toggle for multiple flags in the package.

As you build more and more sophisticated packages, you will find yourself building code in multiple layers that interact in ways mysterious to normal human beings. The windowing technique illustrated by *PLVdyn.showsql* will be absolutely critical to making your packages widely accessible and usable. If you do not provide clearly defined windows into your inner workings, there is a good chance that developers will first be baffled and then become frustrated. The end result is that they will not use your packages.

PREVIOUS | HOME | NEXT
2.6 Building Flexibility Into Your Packages | BOOK INDEX | 2.8 Overloading for Smart Packages

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8*i* Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

2.8 Overloading for Smart Packages

One of the most powerful aspects of the package is the ability to overload program units. When you overload, you define more than one program with the same name. These programs will differ in other ways (usually the number and types of parameters) so that at runtime the PL/SQL engine can figure out which of the programs to execute. You can take advantage of the overloading feature of packages to make your package−based features as accessible as possible.

Does overloading sound unfamiliar or strange? Well, have you ever used the TO_CHAR function? If so, then you have already been enjoying the creature comforts of overloading. TO_CHAR converts both numbers and dates to strings. Have you ever wondered why you don't have to call functions with names like TO_CHAR_FROM_DATE or TO_CHAR_FROM_NUMBER? Probably not. You probably just took TO_CHAR for granted, and that is how it should be.

In reality, there are two different TO_CHAR functions (both defined in the STANDARD package): one to convert dates and another to convert numbers. The reason that you don't have to care about such details and can simply execute TO_CHAR is that the PL/SQL runtime engine examines the kind of data you pass to TO_CHAR and then automatically figures out which of the two functions (with the same name) to execute. It's like magic, only it's better than magic: it's intelligent software!

When you build overloaded modules, you spend more time in design and implementation than you might with separate, standalone modules. This additional up−front time will be repaid handsomely down the line in program productivity and ease of use.

You will not have to try to remember the different names of the modules and their specific arguments. Properly constructed, overloaded modules will have anticipated the different variations, hidden them behind a single name, and liberated your brain for other, more important matters.

See *Chapter 16*, of *Oracle PL/SQL Programming* for a more comprehensive coverage of overloading restrictions and examples.

2.8.1 When to Overload

When you overload, you take the first step towards providing a declarative interface to PL/SQL−based functionality. With a declarative approach, a developer does not write a program to obtain the necessary functionality. Instead, she describes what she wants and lets the underlying code handle the details (this follows the approach used by the SQL language). The process of overloading involves abstracting out from separate programs into a single action.

You want to display a date? You want to display a number? You want to display a string *and* a number? Hold on a minute. The common element is that you want to display something −− lots of somethings, in fact. So don't create **`display_date`**, **`display_string`**, etc. procedures. Instead, offer a single **`display`** procedure, which is in fact many overloaded **`display`** procedures.

With the overloading in place, your user must only remember this: when I want to display something, I simply ask the **`display`** program to take care of it for me. What do I pass to it? Whatever I want it to display. I will not (and do not have to) worry about the *how* of the display mechanism. Those details are hidden from me.

Here are some of the circumstances that cause the PL/SQL fairy to whisper in my ear "Overload, overload...":

- Apply the same action to different kinds or combinations of data.
- Allow developers to use a program in the most natural and intuitive fashion; you use overloading to fit your program to the needs of the user.
- Make it easy for developers to specify, unambiguously and simply, the kind of action desired.

I explore these circumstances in the following sections.

2.8.1.1 Supporting many data combinations

This is probably the most common reason to employ overloading. The **p** package of PL/Vision (see the following sidebar) offers an excellent example of this kind of overloading opportunity. This package contains eight overloadings of the **l** procedure so that you can pass many different combinations of data and have the package interpret and display the information properly. The following headers show, for example, a simplified portion of the specification for the *p* package, which illustrates the overloading:

```
PROCEDURE l (date_in IN DATE, mask_in IN VARCHAR2 := PLV.datemask);
PROCEDURE l (char_in IN VARCHAR2, number_in IN NUMBER);
PROCEDURE l (boolean_in IN BOOLEAN);
```

Because of my extensive overloading, I can pass a complex date expression (taking me back 18 years) and see the date and time in a readable format with a minimum of effort:

```
SQL> exec p.l(ADD_MONTHS(SYSDATE,−316));
February 18, 1970 17:50:12
```

I can combine strings and numbers together easily, as shown in this exception section:

```
BEGIN
    p.l (1/0);
EXCEPTION
   WHEN ZERO_DIVIDE
   THEN
      p.l (SQLERRM, SQLCODE);
END;
/

SQL> @above_script
ORA−01476: divisor is equal to zero: −1476
```

And, finally, I can pass a Boolean expression directly to the *p.l* procedure and have it display meaningful information:

```
SQL> exec p.l ('a' IN ('d', 'e', 'f'));
FALSE
```

Just to give you a sense of the benefit of overloading in this case, if I did not have access to the *p* package and instead relied on DBMS_OUTPUT.PUT_LINE to generate my output, I would have to write the following code to handle the last call to *p.l*:

```
IF bool_value IN ('d', 'e', 'f')
THEN
   DBMS_OUTPUT.PUT_LINE ('TRUE');
ELSE
   DBMS_OUTPUT.PUT_LINE ('FALSE');
END IF;
```

Why do I need to do this? The DBMS_OUTPUT package *does* overload its PUT_LINE procedure, but only for single string, date, and number values. It does not handle Booleans at all. It also does not allow me to pass combinations of data. And it does not show the time component of a date variable. What a hassle! For all these reasons, my extra layer of overloaded code in the *p* package liberates me from having to write extra code. I just tell *p.l* what I want to see and it figures out how to display that information.

Why Name a Package "p"?

I talk about coming up with names for your package that are clear, accurate, and easy to remember. Then I showcase the *p.l* procedure in my best practice on overloading. Surely I am not going to argue that *p* is a good name for a package −− and what about *l* as the name of a procedure? What justification could I possibly have for the names I chose for these elements?

My *p.l* procedure is a substitute for and *rebellion against* the DBMS_OUTPUT.PUT_LINE procedure. I hated the 20 characters I had to type to generate output from my PL/SQL programs (in uppercase no less, since that is my convention). I was frustrated by the limited overloading of the package itself. So when I set out to build my own layer of code around DBMS_OUTPUT, I was determined to use the fewest characters possible. The result is *p.l*.

I found it difficult to justify this obscure name, but John Beresniewicz, my able and deep−thinking reviewer, contributed this observation: "It's possible that the need for clearly descriptive (i.e., lengthy) names is directly proportional to the amount of work performed by the procedure and inversely proportional to the frequency of use. That is, procedures that implement a high level of functionality need clearly descriptive names and they will presumably be called less frequently (and these long names won't clog up the source code). Conversely, low−level routines called frequently need shorter names (to avoid clog) but nobody forgets their names (even if cryptic) since they are used all the time." I couldn't have stated it better myself!

The same technique is also readily visible in the PLVtab package. This PL/SQL table−oriented package offers nine overloadings of the **`display`** procedure, one for each kind of PL/SQL table predefined in the package. As far as a user of *PLVtab.display* is concerned, there is just one program to display a PL/SQL table. The only difference between each of the versions of *PLVtab.display* is the first argument, the table type, as shown in the following header for the **`display`** procedure:

```
PROCEDURE display
 (table_in IN number_table|boolean_table|date_table,
  end_row_in IN INTEGER,
  header_in IN VARCHAR2 := NULL,
  start_row_in IN INTEGER := 1,
  failure_threshold_in IN INTEGER := 0,
  increment_in IN INTEGER := +1);
```

When you see that vertical bar in documentation for program headers, by the way, that means you are dealing with an overloaded program. The more variations of data you provide in your overloadings, the more useful you make your package. There is, of course, a price to pay for your overloadings. While the user thinks there is just one program to call, you know that in reality there is a *different* program for each overloading.

A key challenge, therefore, that comes with successful overloading is to figure out how to implement all those programs without creating a total mess in your package body. This challenge is addressed in the section called "Modularizing for Maintainable Packages" later in this chapter.

2.8.1.2 Fitting the program to the user

Does the idea of fitting a program to your user sound odd or unnecessary? If so, change your attitude. We write our software to be used, to help others get their jobs done more easily or more efficiently. You should always be on the lookout for ways to improve your code so that it responds as closely as possible to the needs of your users. Overloading offers one way to achieve a very close fit.

You may sometimes end up with several overloadings of the same program because developers will be using the program in different ways. In this case, the overloading does not provide a single name for different activities, so much as providing different ways of requesting the same activity. Consider the overloading for the *PLVlog.p*`ut_line` (shown in simplified form below):

```
PROCEDURE put_line
   (context_in IN VARCHAR2,
    code_in IN INTEGER,
    string_in IN VARCHAR2 := NULL,
    create_by_in IN VARCHAR2 := USER);

PROCEDURE put_line (string_in IN VARCHAR2);
```

The first header is the low−level version of **`put_line`**. It allows you to specify a full set of arguments to the program, including the context, the code, a string and the Oracle account providing the information. The second header asks only for the string, the text to be logged. What happened to all the other arguments? I suppressed them, because I found that in many situations a user of PLVlog simply doesn't care about all of those arguments. He simply wants to pass it a string to be saved. So rather than make him enter dummy values for all the unnecessary data, I provide a simpler interface, which in turn calls the low−level **`put_line`** with its own dummy values:

```
PROCEDURE put_line (string_in IN VARCHAR2) IS
BEGIN
   put_line (NULL, 0, string_in, USER);
END;
```

It wasn't necessary for me to take this step and provide this overloading. I could simply require that anyone who uses **`PLVlog.put_line`** provide values for all those non−defaulted parameters. If developers really had to use PLVlog, they would follow my bidding. And if I were on some kind of power trip, I would feel properly stroked. But if a developer could choose between PLVlog and another package or utility that didn't make him feel dumb, PLVlog would simply not be used. We almost always have choices. I would rather that my software be used because it was too useful and easy to use to reject.

2.8.1.3 Unambiguous, simple arguments

A less common application of overloading offers a way for developers to specify very easily which of the overloaded programs should be executed. The best way to explain this technique is with an example. The PLVgen package allows you to generate PL/SQL source code, including procedures, functions, and packages. Let's consider how to request the generation of a function.

A function has a datatype: the type of data returned by the function. So when you generate a function, you want to be able to specify whether it is a number function, string function, date function, etc. If I ignored overloading, I might offer a package specification like this:

```
PACKAGE PLVgen
IS
    PROCEDURE stg_func (name_in IN VARCHAR2);
    PROCEDURE num_func (name_in IN VARCHAR2);
    PROCEDURE date_func (name_in IN VARCHAR2);
END;
```

to name just a few. Of course, this means that a user of PLVgen must remember all of these different program names. Is it *num* or *nbr*? *Stg* or *strg* or *string*? Why use the four−letter *date* when the others are just three letters? Wow! That is very confusing. Let's try overloading of the kind previously encountered in this chapter. I will declare a named constant for each kind of data and then, well, it would seem that I really only need one version of the *func* procedure:

```
PACKAGE PLVgen
IS
    stg CONSTANT VARCHAR2(1) := 'S';
    num CONSTANT VARCHAR2(1) := 'N';
    dat CONSTANT VARCHAR2(1) := 'D';
    PROCEDURE func (name_in IN VARCHAR2, type_in IN VARCHAR2);
END;
```

I could then generate a numeric function as follows:

```
SQL> exec PLVgen.func ('booksales', PLVgen.num);
```

Now, I still need to know the names of the constants, so it is pretty much the same situation as we encountered in my first *func* attempt. Furthermore, I would like to be able to pass a default value to be returned by the generated function, so I really would need to overload as shown in the next iteration:

```
PACKAGE PLVgen
IS
    stg CONSTANT VARCHAR2(1) := 'S';
    num CONSTANT VARCHAR2(1) := 'N';
    dat CONSTANT VARCHAR2(1) := 'D';
    PROCEDURE func
      (name_in IN VARCHAR2, type_in IN VARCHAR2, defval_in IN VARCHAR2);
    PROCEDURE func
      (name_in IN VARCHAR2, type_in IN VARCHAR2, defval_in IN NUMBER);
    PROCEDURE func
      (name_in IN VARCHAR2, type_in IN VARCHAR2, defval_in IN DATE);
END;
```

Might there not be a simpler way to handle this? Notice that the second parameter is a way for the user to specify the datatype of the function. You pass in a string constant, and PLVgen uses an IF statement to determine which constant you have provided. Why not skip the constant and simply pass in *data* itself of the right type? Then the PL/SQL runtime engine itself would automatically perform the conditional logic to determine which program to run, which code to execute. Consider this next version of the PLVgen package specification:

```
PACKAGE PLVgen
IS
    PROCEDURE func
      (name_in IN VARCHAR2, type_in IN VARCHAR2, defval_in IN VARCHAR2);
    PROCEDURE func
      (name_in IN VARCHAR2, type_in IN NUMBER, defval_in IN NUMBER);
    PROCEDURE func
      (name_in IN VARCHAR2, type_in IN DATE, defval_in IN DATE);
END;
```

The named constants are gone, no longer needed. I can now generate a numeric function with a default value of 15,000 as follows:

```
SQL> exec PLVgen.func ('booksales', 1, 15000);
```

It doesn't really matter what value I pass as the second argument; it doesn't matter if the argument is a literal or a variable or an expression. It just has to evaluate to a number, so that the PL/SQL runtime engine will know to execute the code associated with the second header in the specification. What could be simpler? You want a numeric function? Pass a number −− any number −− as the type argument. You want a date function? Pass a date −− be it SYSDATE or some locally declared variable.

I am sure that many readers are looking at that last specification and wondering why I just didn't use the **`defval_in`** argument to determine the datatype of the function and skip the **`type_in`** argument entirely. Take a look at the final PLVgen package specification. I provide a default value of NULL for all the **`defval_in`** arguments. I reasoned that you shouldn't have to provide a default value for the function. So I do need that separate, second argument (always required since it has no default value) to guarantee that you will unambiguously specify one of the function generators.

PLVgen uses this technique both for the *func* procedures and the *gas* (get−and−set) procedures. Oracle Corporation also uses this overloading approach in the DBMS_SQL built−in package (check out the DEFINE_COLUMN procedure). In fact, it was the DEFINE_COLUMN overloading that gave me the idea for the overloading you find in the PLVgen package. It took me a while to think through what PL/SQL was doing with DEFINE_COLUMN; I found the simplicity simultaneously clever, devilishly simple, and extremely elegant. It is a technique we should all put into use whenever appropriate.

2.8.2 Developing an Appreciation of Overloading

You should now have a solid feeling for the technique of overloading. To build excellent packages, however, you will need to move beyond simply overloading occasionally to overloading at every possible opportunity and in every possible way. You need to develop a sensitivity to when you should overload and how you can overload most effectively.

The benefits and the beauty of overloading can be appreciated fully only by using overloaded programs −− and then in most cases, you won't even notice, because overloading hides the underlying complexity so you can concentrate on more important issues. You will, I hope, get a sense of the value of overloading from using −− and perhaps even extending −− PL/Vision. Do take some time to pursue the various *spb* files (the package bodies) and examine the many different examples of overloading you will find there.

When I've successfully overloaded an interesting set of programs and succeeded in hiding much of the underlying complexity of my package, I get an all−the−pieces−falling−into−place feeling and a this−is−as−it−should−be feeling and a sense of how−elegant! If you think I sound a bit strange, please withhold judgment until you do some really fancy and extensive overloading and then tell me how you feel.

The more you overload your packaged procedures and functions, the more functionality you offer to your users. Where overloading is appropriate, it is also impossible to overdo your overloading. If you see another interesting and useful combination, if you see a way to simplify the way a user passes information to your package, then overload for it! It will always be the right thing to do; your biggest challenge will be in figuring out how to implement all these overloadings in a modular and maintainable fashion. This issue is addressed in the next section.

PREVIOUS	HOME	NEXT
2.7 Building Windows Into Your Packages	BOOK INDEX	2.9 Modularizing for Maintainable Packages

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

2.9 Modularizing for Maintainable Packages

To build packages that are both immediately useful and enhanceable over the long−run, you must avoid any kind of code duplication inside the package. You need to be ready, willing, and able to create private programs in your package to contain all the shared code behind the public programs of the package. The alternative is a debilitating reliance on the Windows cut−and−paste feature. Cut−and−paste will let you build rapidly −− but what you will be building is a wide, deep hole from which you will never see the light of day.

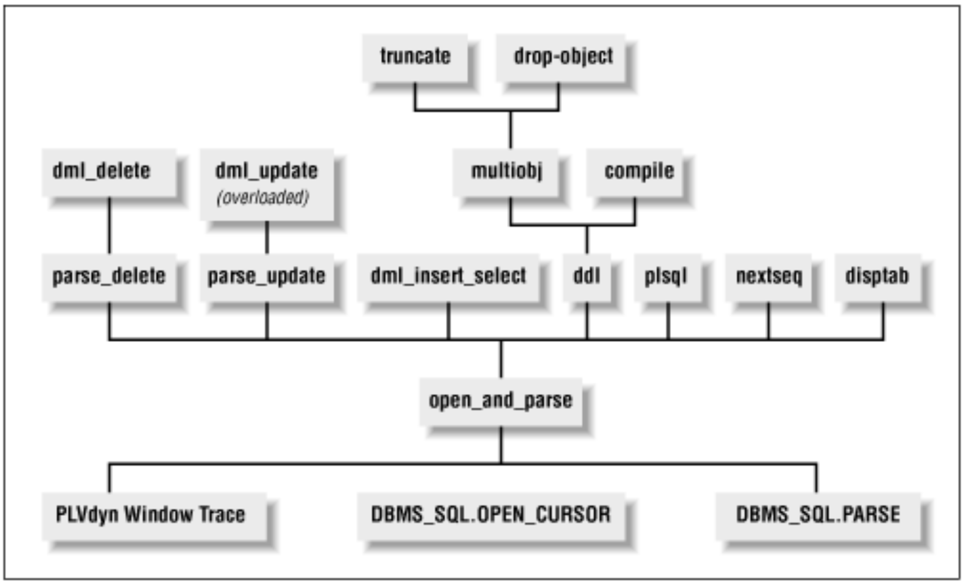
I set a simple rule for myself when building packages: never repeat a line of code. Instead, construct a private module and call that module twice (or more, depending on the circumstances). By consolidating any reused logic rigorously, you have less code to debug and maintain. You will often end up with multiple layers of code right *inside* a single package. These layers will make it easier to enhance the package and also to build in additional functionality, such as the windows and toggles discussed earlier in this chapter.

The PLVdyn package offers an example of in−package layering. As explained in the section on "Building Windows into Packages," the **`open_and_parse`** function consolidates the open and parse phases of dynamic SQL. This function is then called by many other higher−level operators in PLVdyn. These operators are in turn called by still other programs. The result is at least five layers of code as shown in Figure 2.2.

> ***NOTE:*** While it is uncommon, it is certainly possible for two programs to have the same name (and therefore be overloaded) but have little or nothing in common in their implementation. In this situation, you will probably not be able to consolidate the code in the package body for these two programs into a single, private program. There is nothing wrong with this situation (except that it might raise question of why you are using the same name for both programs).

The need for modularization inside a package is most clear when it comes to implementing overloaded programs. The next section will explore implementation strategies for overloading.

Figure 2.2: Layers of reusable code in PLVdyn



2.9.1 Implementing Overloading with Private Programs

Overloading and modularization must be considered two sides of the same coin if you are going to implement your package properly. The previous section encouraged you to overload frequently and thoroughly. When you overload, you offer multiple versions of the same program. By doing so, you simplify the interface for the user, which is critical. At some point, however, you have to deal with the package body. If you've overloaded a particular procedure ten times, are you going to end up with ten completely separate procedure bodies and a large volume of redundant code that is very difficult to maintain?

Let's first understand the problem you can encounter inside packages when you overload. Consider that simplest (at first glance) of packages: the *p* package. You might be tempted to think that all it really does is provide a layer of code over the DBMS_OUTPUT.PUT_LINE built−in so that you can pass it more and different types of data. If that were the case, I could implement the *p.l* procedure as shown by the two of seven implementations below:

```
PROCEDURE l (char_in IN VARCHAR2, number_in IN NUMBER) IS
BEGIN
   DBMS_OUTPUT.PUT_LINE (char_in || ': ' || TO_CHAR (number_in));
END;

PROCEDURE l
   (char_in IN VARCHAR2, date_in IN DATE,
    mask_in IN VARCHAR2 := PLV.datemask)
IS
BEGIN
   DBMS_OUTPUT.PUT_LINE
      (char_in || ': ' || TO_CHAR (date_in, mask_in));
END;
```

I have achieved the objective of overloading by taking on the job of combining the different pieces of data before passing it to the built−in package. No need for a private program shared by all the **l** procedures, is there? Well, that depends on just how useful you want that package to be.

Let's pretend that it is July 1994. I am writing *Oracle PL/SQL Programming* and just beginning to get a handle on packages. The *p* package (at that time called the **do** package) is one of my first and I throw it together, just as you see it above: a "raw" call to DBMS_OUTPUT. Then I start to use it to debug the PLVlst package (as it first appeared in that book) and at some point pass it a string with 463 characters. Suddenly, my program is generating a VALUE_ERROR exception. After a hour of debugging, I realize that the problem is not occurring in PLVlst, but in my *p* package. The DBMS_OUTPUT.PUT_LINE program cannot handle values with more than 255 bytes. I mutter venomously about the brain−dead implementations proffered at times by Oracle Corporation and quickly move to fix the problem, as you can see below:

```
PROCEDURE l (char_in IN VARCHAR2, number_in IN NUMBER) IS
BEGIN
   DBMS_OUTPUT.PUT_LINE
      (SUBSTR (char_in || ': ' || TO_CHAR (number_in), 1, 255));
END;

PROCEDURE l
   (char_in IN VARCHAR2, date_in IN DATE,
    mask_in IN VARCHAR2 := PLV.datemask)
IS
BEGIN
   DBMS_OUTPUT.PUT_LINE
     (SUBSTR (char_in || ': ' || TO_CHAR (date_in, mask_in), 1, 255));
END;
```

Remember, I do this for all eight versions of the *l* procedure, not just the two you see. Well, that certainly takes care of that problem! So I continue my debugging and soon discover that when I ask DBMS_OUTPUT.PUT_LINE to display a NULL value or any string that LTRIMs to NULL, it just ignores me. I do not see a blank line; it just pretends that I never made the call. This is very confusing and irritating, but again the fix is clear: use the NVL operator. So now each of the *l* procedures looks like this:

```
PROCEDURE l (char_in IN VARCHAR2, number_in IN NUMBER) IS
BEGIN
   DBMS_OUTPUT.PUT_LINE
      (NVL (SUBSTR (char_in || ': ' || TO_CHAR (number_in), 1, 255),
       'NULL'));
END;

PROCEDURE l
   (char_in IN VARCHAR2, date_in IN DATE,
    mask_in IN VARCHAR2 := PLV.datemask)
IS
BEGIN
   DBMS_OUTPUT.PUT_LINE
     (NVL (SUBSTR (char_in || ': ' || TO_CHAR (date_in, mask_in),
       1, 255), 'NULL'));
END;
```

On and on I go, discovering new wrinkles in the implementation of DBMS_OUTPUT.PUT_LINE and scrambling to compensate in each of my eight procedures (see Chapter 6, *PLV: Top−Level Constants and Functions* , for more details on these wrinkles). Eventually each of my *l* procedures grows very convoluted, very similar to all the others, and very tedious to maintain. This is clearly not the way to go.

Now compare that process with the final state of the *p* package. Each of the *l* procedures consists of exactly one line of code, as you can see below:

```
PROCEDURE l
   (char_in IN VARCHAR2, number_in IN NUMBER,
    show_in IN BOOLEAN := FALSE)
IS
BEGIN
   display_line (show_in, char_in || ': ' || TO_CHAR (number_in));
END;

PROCEDURE l
   (char_in IN VARCHAR2, date_in IN DATE,
    mask_in IN VARCHAR2 := PLV.datemask,
    show_in IN BOOLEAN := FALSE)
IS
BEGIN
```

```
      display_line
         (show_in, char_in || ': ' || TO_CHAR (date_in, mask_in));
   END;
```

Only two actions are performed inside the *l* procedure:

1. Create the string to be displayed (usually occurring right inside the call to **display_line**), and

2. Call the private module, **display_line**, to handle all the other issues.

The **display_line** procedure, in turn, looks like this:

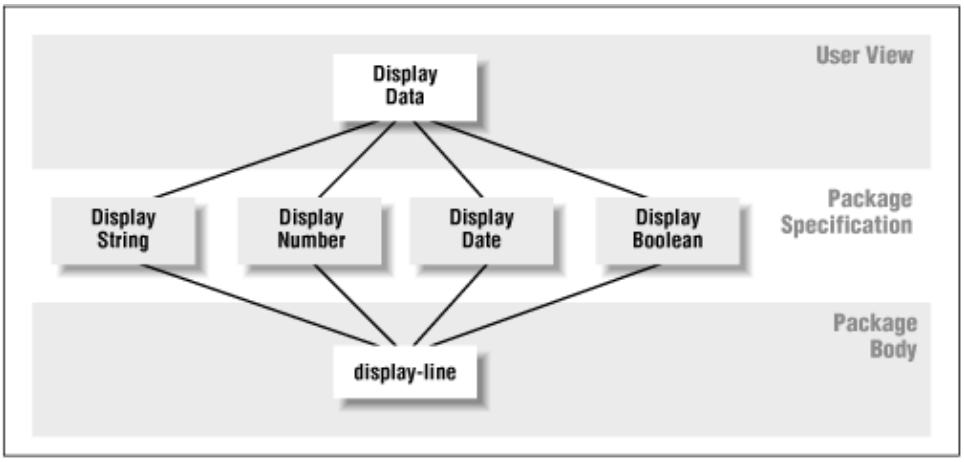
```
PROCEDURE display_line (show_in IN VARCHAR2, line_in IN VARCHAR2)
IS
   v_maxline INTEGER := 80;
BEGIN
   IF v_show OR show_in
   THEN
      IF RTRIM (line_in) IS NULL
      THEN
         put_line (v_prefix || PLV.nullval);

      ELSIF LTRIM (RTRIM (line_in)) = v_linesep
      THEN
         put_line (v_prefix);

      ELSIF LENGTH (line_in) > v_maxline
      THEN
         PLVprs.display_wrap (line_in, v_maxline-5, NULL);

      ELSE
         put_line
            (v_prefix ||
             SUBSTR (line_in, 1, c_max_dopl_line-v_prefix_len));
      END IF;
   END IF;
END;
```

Wow! It got really complicated, didn't it? In the final version of *p.l*, in fact, you can turn off the display of information using the built−in toggle. You can display long lines in paragraph−wrapped format. You can identify a character as a line separator so that white space can be preserved inside stored code and displayed as true blank lines in SQL*Plus.

I didn't come up with all of these features in a single flash of inspiration. I built them in over a period of months. Once I had transferred all common logic into the **display_line** procedure, however, it was a cinch to provide significant new functionality: I only had to make the changes in *one location in my package*. No user of the *p* package ever calls the **display_line** procedure; it is hidden. It exists only to consolidate all the common logic for displaying information.

I use this same approach throughout PL/Vision. Again and again, you will see the many overloadings of the package specification reduced to a single program inside the package body. I like to think of this of the overload−modularize diamond for packages, which is shown in Figure 2.3. The upper point of the diamond is the user view: a single action (i.e., overloaded name) known and called by the user. The facets of the diamond broaden out to the different, overloaded programs in the specification. The lower point of the diamond represents the narrowing of the different programs to a single private program in the package body.

Figure 2.3: The overload−modularize diamond for packages



Sometimes creating this diamond shape in your packaged code is straightforward. The *p* package illustrates this simple case. The only difference between each of the overloaded programs is the way the string is constructed for display. In other packages, it takes lots more thought and creative programming to come up with a way to conform to my "only in one place" rule. The PLVtab package is such a package; in fact, the complexity of modularizing the internals of PLVtab resulted in what I call the *lava lamp effect*.

2.9.2 Lava Lamp Code Consolidation

The objective of PLVtab is to make it easier for developers to use PL/SQL tables, particularly when it comes to displaying the contents of these tables. PLVtab predefines a set of table TYPE structures, such as tables of numbers, strings of various lengths, Booleans, and dates. It then offers a separate **`display`** procedure for each of the table TYPEs. Since each table TYPE is a different datatype, a separate, overloaded program is needed for each TYPE. The headers for two of these follow:

```
PROCEDURE display
 (table_in IN number_table,
  end_row_in IN INTEGER,
  header_in IN VARCHAR2 := NULL,
  start_row_in IN INTEGER := 1,
  failure_threshold_in IN INTEGER := 0,
  increment_in IN INTEGER := +1);

PROCEDURE display
 (table_in IN vc30_table,
  end_row_in IN INTEGER,
  header_in IN VARCHAR2 := NULL,
  start_row_in IN INTEGER := 1,
  failure_threshold_in IN INTEGER := 0,
  increment_in IN INTEGER := +1);
```

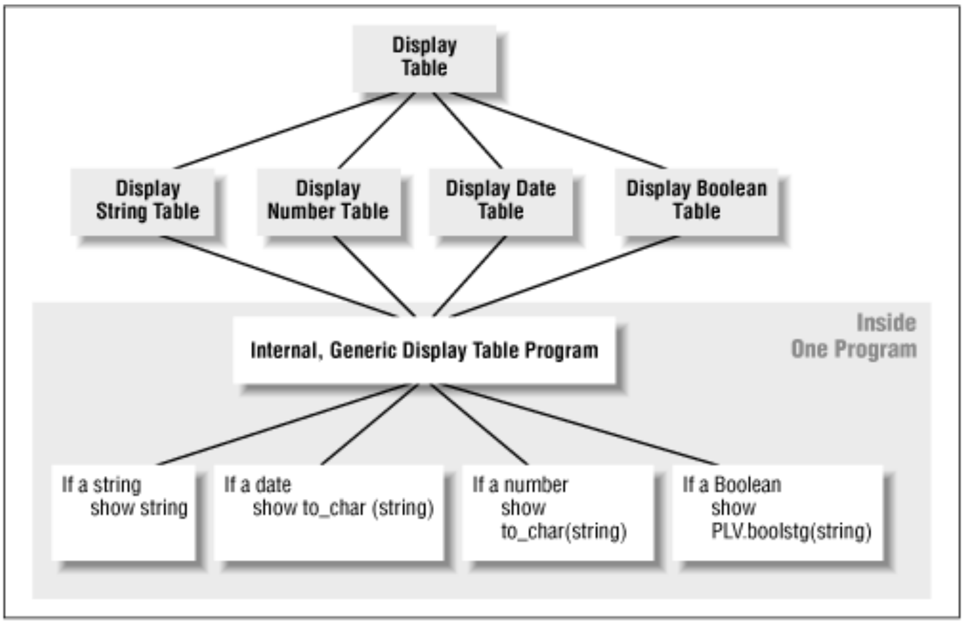
As you can see from all of the parameters in the **`display`** procedures, PLVtab offers lots of flexibility in what you display and how you display it. This kind of flexibility *always* means a more complex implementation behind the scene in the package body. In fact, I use 184 lines of code spread across two private procedures to handle all the logic. Yet the body of each **`display`** procedure consists of just three lines as illustrated by the **`number_table`** **`display`** procedure below:

```
PROCEDURE display
   (table_in IN number_table,
   end_row_in IN INTEGER,
   header_in IN VARCHAR2 := NULL,
   start_row_in IN INTEGER := 1,
   failure_threshold_in IN INTEGER := 0,
   increment_in IN INTEGER := +1)
IS
BEGIN
   internal_number := table_in;
   internal_display
      (c_number, end_row_in, header_in, start_row_in,
       failure_threshold_in, increment_in);
   internal_number := empty_number;
END;
```

The first line copies the incoming table to a private PL/SQL table of the same type; the second line calls the consolidated, internal version of the **display** program; and the third line empties the private PL/SQL table to minimize memory utilization. These same three lines appear in each of the nine **display** procedures, the only difference being the type of the private PL/SQL table and the first argument in the call to **internal_display**. This value tells **internal_display** which table is to be displayed. This approach allows me to create the lower point of my diamond: a single program called by each **display** program.

The difference in this case -- what I call the lava lamp effect -- is that deep within the **internal_display** procedure, I broaden out my code again (creating a base for my lava lamp; see Figure 2.4) with a large IF statement.

Figure 2.4: The lava lamp effect for consolidating overloaded code


Display Table
Display String Table
Display Number Table
Display Date Table
Display Boolean Table
Internal, Generic Display Table Program
Inside One Program
If a string show string
If a date show to_char (string)
If a number show to_char(string)
If a Boolean show PLV.boolstg(string)

The main algorithm of **internal_display** is this WHILE loop:

```
WHILE in_range (current_row) AND within_threshold
LOOP
   display_row
      (type_in, failure_threshold_in, increment_in,
      count_misses, current_row, within_threshold);
END LOOP;
```

which translates roughly as "display each row within the specified range." The **display_row** is another private procedure that converts the **type_in** argument (the type of table being displayed) and the **current_row** into the row value to be displayed. To do this, it uses a big IF statement, a portion of which is shown here:

```
...
```

```
   ELSIF type_in = c_date
   THEN
      rowval := TO_CHAR (internal_date (current_row), PLV.datemask);

   ELSIF type_in = c_integer
   THEN
      rowval := TO_CHAR (internal_integer (current_row));

   ELSIF type_in = c_number
   THEN
      rowval := TO_CHAR (internal_number (current_row));

   ELSIF type_in = c_vc30
   THEN
      rowval := internal_vc30 (current_row);
   ...
```

The overloading and modularization in PLVtab (and PLVgen as well) reminds me of playing an accordion: First, I squeeze in to present a single program for the user. Then I pull out to define the many overloadings in the specification. Squeeze back in to implement all those overloadings with a single private module. Pull back out inside that private module to handle all the different types of data. It may seem like a convoluted road to travel, but the end result is code that is very easily maintained, enhanced, and expanded.

As an exercise for the reader, I suggest that you perform this exercise: make a copy of *PLVtab.sps* and *PLVtab.spb* and see if you can figure out the steps required to add support for another type of PL/SQL table.

PREVIOUS | HOME | NEXT
2.8 Overloading for Smart Packages | BOOK INDEX | 2.10 Hiding Package Data

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

2.10 Hiding Package Data

You implement PL/SQL−based global data with *package data*. Package data is any data structure declared in a package body or specification. There are two kinds of package data: public data (declared in the specification) and private data (declared in the body).

What's the difference between public and private? Public global data is the proverbial "loose cannon" of programming. Public package data is certainly very convenient. Simply declare a few variables in a package specification and they are available from/to any module. If you need to get a piece of information, just grab it from the global. If you want to change the value of that variable, go at it. Reliance on global data structures, however, leads to two significant problems:

- *Loss of control*. When you declare a data structure in the package specification, you lose control over that data structure's value. Since any program can write to it, you can never trust its value. Instead, you must trust developers to do the right thing when working with that variable. Now, I am as trusting as the next programmer, but anarchy really has little place in the world of software development.
- *Loss of flexibility*. When you allow programmers to make direct references to global data, you lose the flexibility you need to enhance your application to take advantage of new features. Very specifically, you limit your ability to change the data structures used to implement your global data.

You don't *have* to create these troublesome globals to gain many of the advantages of PL/SQL global data structures. You can regain control of your package data and also ease your maintenance and enhancement frustrations by building a programmatic interface around your data. This interface is also referred to as get−and−set programs or "access routines," since they usually get and set the values of data and control access to those data structures.

2.10.1 Gaining Control of Your Data

I recommend, in fact, that you *never* define variables in the specification of a package (except when explicitly needed that way, as discussed at the end of this section). Instead, you always declare the variable in the package body. You then provide a procedure to set the value of that variable and a function to retrieve the value of that variable.

Let's look at a very simple example to drive home the point and then move on to more interesting applications of this practice. Suppose I have a profit−and−loss package that maintains a "last statement date" in a package variable. With the variable defined in the specification, my package looks like this:

```
PACKAGE P_and_L
IS
   last_stmt_dt DATE;
END P_and_L;
```

Suppose further that I have a business rule that applies to the last statement date: it can never be in the future.

Since the variable is defined in the package specification, any user with execute authority on this package can directly reference and modify the variable as shown in these code fragments:

```
P_and_L.last_stmt_dt := SYSDATE + 12;
v_newdate := P_and_L.last_stmt_dt;
```

In the first line, my code violates the business rule −− and there is nothing I can do to stop this violation.

Let's now move the **`last_stmt_dt`** inside the package body. When I do this, I must write some code to provide a programmatic interface to that date variable. The resulting package specification and body shown in Example 2.2 provide get−and−set routines to get the current value of the last statement date and also set the value of that variable.

Example 2.2: The P_and_L Package with Private Data

```
PACKAGE P_and_L
IS
   FUNCTION last_date RETURN DATE;

   PROCEDURE set_last_date (date_in IN DATE);
END P_and_L;

PACKAGE BODY P_and_L
IS
   last_stmt_dt DATE;

   FUNCTION last_date RETURN DATE IS
   BEGIN
      RETURN last_stmt_dt;
   END;

   PROCEDURE set_last_date (date_in IN DATE) IS
   BEGIN
      last_stmt_dt := LEAST (date_in, SYSDATE);
   END;
END P_and_L;
```

Sure, this is a lot more code than was necessary to simply "publish" the last statement date variable in the package specification. The benefits of this code are, however, significant and will now be explored. First of all, notice that the **`set_last_date`** procedure applies or enforces the business rule whenever anyone tries to change the value of the **`last_stmt_dt`** variable. Let's examine the impact of this enforcement. With my packaged interface, the two lines of code I showed you earlier would be changed to:

```
P_and_L.set_last_date (SYSDATE + 12);
v_newdate := P_and_L.last_date;
```

Now instead of setting the last statement date to twelve days in the future, **`set_last_date`** intervenes and sets the date to the system date. (Of course, in the real world, you would probably not enforce a business rule by simply overriding a user action. For purposes of demonstration, however, it gets the point across.)

By moving **`last_stmt_dt`** to the inside of the package, I have exerted control over my package data. I can now guarantee the integrity of this data to any user of the package; you know what you are getting when you call the **`last_date`** function. In the first version of the *P_and_L* package, there was no way to know how the value was set.

This control and integrity is the most important benefit accrued from hiding your data in the body of the package. Many other wonderful advantages are possible, however, once you have taken this step. These are covered in the following sections.

2.10.2 Tracing Variable Reads and Writes

Have you ever lost control of your application? I once worked on an Oracle Forms application in which there was no doubt that the complexity of the code (and workarounds in the code) had caused it take on a life of its own. This application relied heavily on Oracle Forms GLOBAL variables −− to the tune of 400 or so of these useful, but dangerous constructs. And, sad to say, we could not, in a number of circumstances, figure out why and how a particular global was being set to NULL or to some other value that made no sense for the action at hand.

There had been no forethought in the use of the global variables. Everyone was scrambling to meet deadlines with a very early version of Oracle Forms (4.0.6 for those of you who know to shudder at such things) and just threw direct references to the globals willy−nilly throughout the code. There was no way, consequently, to trace where and when a global value was changed. If, on the other hand, the original developers of the application had built a package around the use of Oracle Forms globals, such a trace would have been very possible, and much agony would have been averted.

I demonstrate below the tracing technique for the *P_and_L* package. You can then apply this technique to Oracle Forms global variables and any other variable data structure.

Let's go back to the *P_and_L* package shown in Example 2.2 and the last statement date. The variable is declared in the package body. A function is provided to return the current value of **`last_stmt_dt`**. A procedure, **`set_last_date`**, allows me to change the variable's value. I build an application making many references to these programs and then I start testing that application. I soon run into trouble. The last statement date is being set improperly, but it is very difficult for me to figure out how and why its value is being changed.

What I would really like to do is obtain a trace of every contact with that variable. If I had not hidden the last statement date variable inside a package, my situation would be hopeless. I would have no way to know when my programs were touching the last statement date.

With my **`last_date`** function and **`set_last_date`** procedure in place, on the other hand, I can with just a few lines of code get all the information I need. In the upgraded version of the **`P_and_L`** package below, I use the PLVtrc package (see code in bold) to add an execution trace to the last statement date's get−and−set:

```
PACKAGE BODY P_and_L
IS
   last_stmt_dt DATE;

   FUNCTION last_date RETURN DATE IS
   BEGIN
      PLVtrc.show ('Retrieve last_date', last_stmt_dt);
      RETURN last_stmt_dt;
   END;

   PROCEDURE set_last_date (date_in IN DATE) IS
   BEGIN
      PLVtrc.show ('Set last_date', date_in);
      last_stmt_dt := LEAST (date_in/, SYSDATE);
   END;
END P_and_L;
```

The **`PLVtrc.show`** procedure intercepts attempts to read or write the **`last_stmt_dt`** variable. This trace is, however, not active, until the following command is used to turn on the trace for the current session:

```
PLVtrc.turn_on;
```

When she turns trace on, a developer can view (or write to the PL/Vision log) a record of every effort to read or write the variable. And if the PL/SQL programs that call the *P_and_L* package make use of the PLVtrc startup and terminate programs, this record will automatically include the names of the programs or context when the **last_stmt_dt** variable was referenced (see Chapter 20, *PLVcmt and PLVrb: Commit and Rollback Processing*). Just a little bit of added code produced a significant enhancement in functionality!

Furthermore, all of my tracing changes occurred to the package body; the specification was left intact. As a result, none of the programs that call the **P_and_L** elements need to be changed or even recompiled. No one even has to know that the package has been upgraded with the new feature; it will be invisible until turned on −− and then only for the current Oracle session, not for all users.

Once I built the get−and−set around my date variable, adding an execution trace facility was very simple. Just get that layer of code in place and many seemingly and formerly impossible tasks become easy!

2.10.3 Simplifying Package Interfaces

Another reason for moving data into the package body is to simplify the interfaces to the package elements. When data are declared in the package body, they are global *within the package*. All programs defined in the package (specification and body) can reference these variables directly. You can use this fact to your advantage by *not* passing in these values in the parameter lists of the package elements.

Consider the PLVobj package, which provides a programmatic interface to the ALL_OBJECTS data dictionary view. PLVobj works with a current object, which is made up of three elements:

- The owner or the schema of the object
- The name of the object
- The type of the object

The PLVobj package and other packages such as PLVio, perform many different operations on this current object, including the following: bind the object for dynamic SQL execution, open a cursor into the ALL_OBJECTS view for this object, read the source code for that object, and so on.

Suppose that I did not store this current object in the package. Then every time I wanted to perform one of the above actions, I would have to provide the values for each of these elements of the current object in the parameter list. Let's look at some examples.

Instead of calling **PLVobj.open_objects** without any arguments like this:

```
PROCEDURE open_objects;
```

I would need to modify the header as follows:

```
PROCEDURE open_objects
   (name_in IN VARCHAR2, type_in IN VARCHAR2, schema_in IN VARCHAR2);
```

And deep within the PLVio package, I could no longer simply call the *bindobj* program relying on the context or current object previously set, as I do here:

```
PLVobj.bindobj (cur);
```

Instead, I would have to maintain variables inside PLVio with the current object values and then pass them into **bindobj** as follows:

```
PLVobj.bindobj (cur, currobj_name, currobj_type, currobj_schema);
```

Would you use a package designed that way? I don't think I would. All those arguments, passed in over and over again. Each time thinking: why can't the package just keep track of that for me?

Well, it can and PLVobj does just that. The current object of PLVobj is defined by three private package variables:

`v_currschema`
The owner of the object

`v_currname`
The name of the object

`v_currtype`
The type of the object(s)

Since the above elements are private variables, a user of PLVobj will never see or reference these variables directly. Instead, I provide a program to set the current object. Its header is:

```
PROCEDURE setcurr (name_in IN VARCHAR2);
```

where the argument is the module name, which can actually be a composite of the schema, name, and type.

With the **`setcurr`** procedure assigning values to my current object, the parameter lists of my object−management programs in PLVobj become short and sweet. They are much easier to use.

There is, of course, a tradeoff when you rely on package global data instead of passing parameters. Sure, the data is private and access to it is controlled. But it also means that the package program is completely dependent on that data. You cannot use the program to analyze or manipulate data until it is set into the package globals. The only way you can use the PLVobj package is to first call the **`setcurr`** procedure.

I believe that in many cases, this tradeoff is a good investment. It reinforces my perspective on the package as an *environment* more than simply a collection of related code elements.

2.10.4 When to Make Data Public

You shouldn't always hide your data in the package body. Sometimes you really do want to let someone directly access the information. I have found, for example, that if you are going to execute dynamically constructed PL/SQL code with the DBMS_SQL package and you want to reference any kind of external data directly, it must be defined in the specification of some package. Dynamically executed PL/SQL blocks are never nested within another block. As a result, they can only reference variables declared in the dynamic block or in a package specification (see Chapter 18, *PLVcase and PLVcat: Converting and Analyzing PL/SQL Code*, for more details).

Another place in PL/Vision where I violate this practice and declare data structures in the specification is the PLVio package. You can choose to use a PL/SQL table as a target with the following call:

```
PLVio.settrg (PLV.pstab);
```

Then all subsequent calls to **`PLVio.put_line`** will deposit information in another row of data in the PLVio−based PL/SQL table, defined in the specification as follows:

```
   target_table PLVtab.vc2000_table;
   target_row BINARY_INTEGER;
```

Why did I put this table in the specification? I suppose I could have hidden it away in the body and then built some programs that would maintain the contents of the table, along these lines:

```
PROCEDURE init_table;
PROCEDURE set_row (val_in IN VARCHAR2);
FUNCTION rowval (row_in IN INTEGER) RETURN VARCHAR2;
PROCEDURE display;
```

Maybe I just got lazy that night. But maybe, just maybe, it actually makes more sense in this case to allow the developer to do whatever she wants with the table and its contents. It is just a repository, after all, for the output from calls to the **`PLVio.put_line`** procedure. You might, in fact, want to write some information from PLVio and then add a few rows of data from your own, independent source. Rather than put up the barrier of get−and−set routines, I just leave the table in the specification and make the user responsible for its contents.

2.10.5 Anchoring to Public Variables

There is one other case in which specification−based variables are useful: anchored declarations. You can anchor or base the declaration of a variable on another, predefined structure. To do this, you use the %TYPE and %ROWTYPE attributes. The most common way %TYPE is used is to anchor a local PL/SQL variable to a database column, as shown below:

```
v_ename emp.ename%TYPE;
```

You can also, however, anchor variables to other PL/SQL data structures. You can define variables in one package (a repository of subtypes) that are used to define variables in another package. In this case, the variables must be declared in the specification. An example from PL/Vision will demonstrate this technique.

A number of PL/Vision packages manipulate PL/SQL source code (PLVgen, PLVcase, PLVcat, etc.). One important element of PL/SQL code is the *identifier*. An identifier is a named element of the language. Today, identifiers can be up to 30 characters in length and must start with a letter.

As I built packages to read and parse identifiers (see PLVprsps), I would declare local variables to hold those values. At first, I declared the variable as follows:

```
v_ident VARCHAR2(30);
```

This always made me uncomfortable, though. I could just see Oracle Corporation in its next release announce that it would now allow identifiers to be up to, say, 60 characters in length. My code would instantly become very vulnerable. So I would often compensate by declaring the variable as:

```
v_ident VARCHAR2(100);
```

I felt safe, but dissatisfied. The justification for that declaration was weak; it would be hard (embarrassing?) to explain to another developer why I chose this number. After too many months, I found the ideal solution: use an anchored declaration.

So I added the following declaration to the PLV package specification:

```
plsql_identifier VARCHAR2(100) := 'IRRELEVANT';
```

I decided to use 100 because my identifier variable needed to hold identifiers of the form "package.element" and so that I had some extra space with which to work. I then changed my hard−coded declaration of **`v_ident`** and many other variables to this format:

```
v_ident PLV.plsql_identifier%TYPE;
```

Now if I ever do need to change the length or other characteristic of variables that represented PL/SQL identifiers, I could make that change in just one place. Notice that I assigned the default value of IRRELEVANT to the variable. I did that to emphasize that the value contained in *plsql_identifier* is irrelevant. It is never referenced (or intended to be referenced) for its value, only for its datatype.

> ***NOTE:*** You might be thinking that I should just have declared *plsql_identifier* as a constant and then the value of this "reference only" structure could not be mucked with. That certainly makes sense. I found, however, that you cannot reference a constant in an anchored declaration. If I wanted to use **`plsql_identifier`** to anchor other variable declarations, it had to be declared a variable.

So there are certainly circumstances in which you will want to declare data structures in the package specification. This should occur, however, on an exception basis −− and you should be able to justify your action with some application−specific requirements. Otherwise, hide that package data in the body and you will reap many benefits.

PREVIOUS 2.9 Modularizing for Maintainable Packages | HOME BOOK INDEX | NEXT 2.11 Simultaneous Construction of Multiple Packages

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8*i* Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

Not Found

The requested URL /oracle/advprog/ch02_11.htm was not found on this server.

Apache/1.3.20 Server at books.i−set.ru Port 80

3. The PL/SQL Development Spiral

Contents:

How many times have you written a program, gotten it to compile first time around, and found that it had no bugs? Let me rephrase that question: have you ever written a program of more than, say, five lines that compiled and executed without bugs the first time around? Please send me your resume if the answer is yes. We need developers like you. I will make a confession: I have never once been able to get it right the first time. Perhaps I am just too impatient to walk through my code properly. I certainly haven't found the patience to discover the joys of computer−assisted software engineering. I am just a hacker at heart.

Even with lots of patience and prior analysis, however, I believe that it is wrong to set as a realizable objective to "get it right the first time." Software development should be seen largely as an iterative process. You get closer and closer to "perfection" as you take multiple passes at a solution. I like to think of this process as a spiral towards excellence in code. A spiral is different from a cycle, which is the term often used to portray the, well, "lifecycle" of development. A cycle or circle has you coming back around to where you were before. A spiral implies that when you come back around, you are at a higher place than you were on the previous spin. You are closer to the ideal.

There are many ways to apply this philosophical thinking to PL/SQL development. The single most important non−technical (i.e., not specific to computers) skill is that of problem−solving. The single most important technical skill to nurture as a programmer is that of code modularization. In this chapter, I explore the so−called spiral towards excellence in the context of an exercise in building a very basic and −− on the surface −− trivial PL/SQL utility.[1]

[1] All of the code for this chapter is included on the companion disk in the file *spiral.all*.

Most of this chapter traces the evolution of a standalone function. Why, in a book about packages, am I showing you how to build this function? First, the lessons you will absorb from this chapter apply very directly to the more complex work on package construction. Second, the best solution to the problem addressed by my function turns out to be a package!

3.1 The Basic Problem

Let's suppose that I am writing an application that does lots of string manipulation. One action that I find myself coding again and again is the duplication of a string. In other words, I need to convert "abc" to "abcabc". I write this:

```
v_new_string := v_old_string || v_old_string;
```

and then I write this:

```
v_comp_name := v_comp_name || v_comp_name;
```

and I sense a pattern.[2]

[2] You are not allowed to wonder why I would need to do something like this. I am not obligated to construct a real−life application that does this. You just have to take my word for it.

Whenever I notice a repetition in my coding, I instinctively put on the brakes and examine more closely. A pattern implies that I can generalize to a formula. I can then encapsulate my formula into a function or procedure. I can then write the formula once and apply it often. This cuts down on my typing and improves my ability to maintain and even enhance my code.

In this case, the pattern is clear: I want to double the supplied string and return it to another PL/SQL variable. I want, in other words, to create a function. So in very quick order I write the *twice* function as shown in Example 3.1:

Example 3.1: The First Version of the twice Function

```
CREATE OR REPLACE FUNCTION twice (string_in IN VARCHAR2)
RETURN VARCHAR2
IS
BEGIN
   RETURN string_in || string_in;
END twice;
/
```

With *twice* created in the database, I can replace those two explicit concatenations with these calls to *twice*:

```
v_new_string := twice (v_old_string);
v_comp_name := twice (v_comp_name);
```

I have added another fine implement to my toolbox and I continue on my merry, programming way. Lo and behold, just as I would have predicted, I soon run into the need for *twice* again. This time, I want to double−up the description of a product type, but I also need to make sure that it is in upper case, so I type:

```
v_prodtype:= twice (UPPER (v_prodtype));
```

So far so very good. I code my little heart out until I run into a new variation on my *twice* theme: I want to double the string, but this time I need to uppercase the first instance and lowercase the second. It doesn't take too lengthy an analysis to conclude that the *twice* function cannot handle this requirement.

I now face a crucial and common modularization dilemma: should I try to enhance **`twice`** to handle this new twist or should I leave it as is and build yet another function for UPPER−lower? A part of me would like to tell you that this isn't really much of a dilemma, that you should always widen the scope of your existing program. That would be the most elegant solution, but it would also be irresponsible advice.

We write programs so people can use them, not so we can marvel at their elegance. There are definitely situations in which it makes more sense to leave well enough alone and start with a brand−new build. I am not yet ready to give up on *twice*, for several reasons:

- The point of this chapter is to step you upwards through the spiral. Abandoning the function now would be to start a *new* spiral, not climb this one.
-

> The *twice* function is still very basic and unexplored from the standpoint of scope expansion. It is at least worth trying to incorporate this new requirement. If it proves unnatural, then the effort ought to be abandoned.

Once I have made this analysis, I am ready to revamp the *twice* function in order to add value for the end user (currently myself, but in time others as well).

PREVIOUS HOME NEXT

2.11 Simultaneous Construction of Multiple Packages BOOK INDEX 3.2 Adding Value

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8*i* Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

3.2 Adding Value

Sometimes it is worth stepping back and searching for the bigger picture before embarking on one's enhancements. In this case, I find myself wondering what other twists and turns I might encounter in my application development. Since I need UPPER−lower string duplication, I might also run into a requirement to perform lower−UPPER string duplication. As long as I am changing the *twice* function for one of these variations, I should try to stay ahead of the game and handle both variations.

So I will restate the new requirements of **`twice`**: double the specified string. Return the new string with the same case as the original, and return it in UPPER−lower or return it in lower−UPPER, depending on the user request.

When stated in this way, an obvious question pops up: how is the user going to specify the case handling in the call to **`twice`**? For a standalone function, this means adding a parameter. Instead of just accepting the string value for doubling, *twice* must also receive the type of action to perform. The new header for *twice*, therefore, must be:

```
FUNCTION twice (string_in IN VARCHAR2, action_in IN VARCHAR2)
```

where the action can be one of these values:

`N`

No change to case

`UL`

UPPER−lower case conversion

`LU`

lower−UPPER case conversion

Once the parameter and valid options are in place, the implementation is straightforward (and is shown in Example 3.2). I simply use an IF statement to direct the runtime engine to the right RETURN statement.

Example 3.2: The twice Function with Alternative Actions

```
CREATE OR REPLACE FUNCTION twice
   (string_in IN VARCHAR2, action_in IN VARCHAR2)
RETURN VARCHAR2
IS
BEGIN
   IF action_in = 'UL'
   THEN
      RETURN (UPPER (string_in) || LOWER (string_in));

   ELSIF action_in = 'LU'
   THEN
      RETURN (LOWER (string_in) || UPPER (string_in));
   ELSIF action_in = 'N'
   THEN
      RETURN string_in || string_in;
   END IF;
END twice;
```

With this new version of *twice*, I can display the following string doublings:

```
SQL> exec DBMS_OUTPUT.PUT_LINE (twice ('abc', 'UL'));
ABCabc
SQL> exec DBMS_OUTPUT.PUT_LINE (twice ('abc', 'LU'));
abcABC
SQL> exec DBMS_OUTPUT.PUT_LINE (twice ('abc', 'N'));
abcabc
```

My *twice* function is starting to look interesting. It handles a number of different flavors of conversion and seems easy to use. I'm glad I decided to enhance *twice*.

PREVIOUS | HOME | NEXT
3.1 The Basic Problem | BOOK INDEX | 3.3 Supplying Backward Compatibility

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8*i* Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

3.3 Supplying Backward Compatibility

Having created a very flexible *twice* function and tested it successfully with my new UPPER−lower and lower−UPPER requirements, I can now step back into the stream of application development. (I consider work on *twice* as part of the process of building my generic PL/SQL toolset.) So I continue to code and, after a while, come back to one of my earlier uses of **`twice`**:

```
v_prodtype:= twice (UPPER (v_prodtype));
```

While I don't change this line of code, I do have to modify others in the same procedure. I then recompile that procedure and am shocked to get this error:

```
PLS−00306: wrong number or types of arguments in call to 'TWICE'
```

Suddenly code that was working earlier in the day is no longer even able to compile. What went wrong?

When I enhanced the *twice* function, I added a second parameter −− and I certainly needed to do that. I did not, unfortunately, take into account existing uses of *twice*. The way that I changed the parameter list actually invalidated those prior instances. Since I did not provide a default value for the **`action_in`** parameter, it became necessary for all executions of *twice* to include two values in the argument list. This is an unacceptable way to enhance existing code.

If I am going to make changes to programs currently in use across my production applications (or in any version of previously existing programs), I want to do so in a way that allows that code to continue to work as it did before without any changes. Otherwise I am facing a maintenance nightmare that would, in effect, stop me from enhancing code. It simply isn't possible (especially given the state of PL/SQL development and analysis tools) to search (and replace!) efficiently for all uses of a given program.

I must instead come up with a technique that will support backward compatibility with earlier uses of *twice*, while simultaneously allowing me to use that same program in new ways. Default values for my **`action_in`** parameter offer this possibility.

When an IN parameter has a default value, it is not necessary to include a value for that argument when the program is called. If a value is not specified, the program will use the default value in its execution. If these IN parameters are all trailing parameters (they come at the end of the parameter list), you can simply ignore them when calling the program. If the IN parameters are positioned before one or more IN or IN OUT parameters, you will have to use named notation to skip over that parameter. (See *Oracle PL/SQL Programming* for more details on named notation).

I can make a very simple change to the header of the *twice* function:

```
FUNCTION twice
   (string_in IN VARCHAR2, action_in IN VARCHAR2 DEFAULT 'N')
```

Then, if I call *twice* with just a single argument, it will assume that I do not want to perform any kind of case conversion. With this change in place, all previous occurrences of *twice* will work as they did before I even thought of a case−conversion action parameter. IN parameters with default values are a critical technique in ensuring backward compatibility for enhanced PL/SQL programs.

PREVIOUS HOME NEXT
3.2 Adding Value BOOK INDEX 3.4 Improving the User Interface

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

3.4 Improving the User Interface

A couple of weeks go by before I encounter another need for *twice*. Then I need to call it for lower−UPPER conversion on a company name. So I put this line in my program:

```
v_full_name := twice (comp_rec.short_name, 'lu');
```

but when I execute the program, the full name is not in lower−UPPER format. It is all uppercased and, as I trace my way back to the data, that is just how the company short name is stored in the database. It doesn't seem to be doing any conversion at all.

Frustrated, I decide to head back to the source code. Of course, I can't remember where I stored the source code on disk. It was just a dinky little program. And it's generally not too easy to view the source code as it exists in the USER_SOURCE data dictionary view. Fortunately, I have built a PL/Vision package named PLVvu (more about this in Chapter 14, *PLVtmr: Analyzing Program Performance*) to view the code and so I execute that program to refresh my memory:

```
SQL> exec PLVvu.code('twice');
-----------------------------------------------------------
Code for FUNCTION TWICE
-----------------------------------------------------------
Line#  Source
-----------------------------------------------------------
    1 FUNCTION twice
    2    (string_in IN VARCHAR2, action_in IN VARCHAR2)
    3 RETURN VARCHAR2
    4 IS
    5 BEGIN
    6    IF action_in = 'UL'
    7    THEN
    8       RETURN (UPPER (string_in) || LOWER (string_in));
    9    ELSIF action_in = 'LU'
   10    THEN
   11       RETURN (LOWER (string_in) || UPPER (string_in));
   12    ELSIF action_in = 'N'
   13    THEN
   14       RETURN string_in || string_in;
   15    END IF;
   16 END twice;
```

The problem becomes clear: the action must be passed in as uppercase: *LU* and not *lu*. The solution seems to me to be equally clear: fix my line of code to pass upper case.

```
v_full_name := twice (comp_rec.short_name, 'LU');
```

Well, that certainly is one way to solve the problem. Unfortunately, it is really just a variation on "blame the victim." Why couldn't I pass *lu* in lowercase to get the action I wanted? It's not as if the lowercase version is used by *twice* to perform some other kind of conversion. The case of the action should not be a factor in the way *twice* works. Unfortunately, the way I wrote the program, any user must be aware of this inflexibility of *twice* −− be aware of minute implementation details of *twice* −− or risk introducing bugs in her code.

These are danger signs pointing to a poorly designed program. A user should not have to know anything about the internals of *twice* to use it. Furthermore, the program should be smart enough to accept the action in any number of different formats and do the right thing for the user.

The solution is straightforward: convert the action value provided by the user to upper or lower case and then test based on that case. In this way, the user can enter lower, upper, or mixed case and the program will function as expected. Example 3.3 shows the "smart" version of *twice*, which utilizes this parameter−conversion technique.

Example 3.3: The twice Function with Parameter Conversion

```
CREATE OR REPLACE FUNCTION twice
   (string_in IN VARCHAR2,
    action_in IN VARCHAR2 DEFAULT 'N')
RETURN VARCHAR2
IS
   v_action VARCHAR2(10) := UPPER (action_in);
BEGIN
   IF v_action = 'UL'
   THEN
      RETURN (UPPER (string_in) || LOWER (string_in));

   ELSIF v_action = 'LU'
   THEN
      RETURN (LOWER (string_in) || UPPER (string_in));

   ELSIF v_action = 'N'
   THEN
      RETURN string_in || string_in;
   END IF;
END twice;
/
```

Whenever you require your user to enter literals to direct activity in your program, you should make sure that they do not have to know about the "proper" case in which to enter the literal. Make your program smart enough to interpret a range of entries.

PREVIOUS | HOME | NEXT
3.3 Supplying Backward Compatibility | BOOK INDEX | 3.5 Rough Waters Ahead

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

3.5 Rough Waters Ahead

I have now fixed the *twice* function so that a user can enter UL, LU, or N in any case he wants. The function seems functional; I figure that someone else might even want to use it. So I send out an email to my development team describing how to use *twice*.

A day later I get a call from a co−developer complaining about **`twice`**. The function is once again not following orders. He asked *twice* to perform upper−lower conversion and all he got was this message:

```
ORA−06503: PL/SQL: Function returned without value
```

Suddenly I am in the role of telephone support and it's not much fun. I am quickly baffled and ask him to read to me exactly what he typed in. He says:

```
new_name := twice (old_name, 'BS');
```

"What's `BS'?", I ask him, feeling as though I am walking right into something I will regret.

"Big−Small," he responds. I sigh with relief. He continues: "I thought that's what I was supposed to pass to **`twice`**: B for big letters and S for small letters."

It doesn't take long for me to straighten him out (that is, to tell him the secret codes). Turns out that my email message just assumed that my co−developers would understand the U and L stuff. Intuitive, really. But of course our minds all work differently and what is obvious to one person is obscure, at best, to another.

Unfortunately, the way I built *twice* assumed that the user would know the correct codes. And my assumption was so strongly held that I don't even include any code to let the user know that a mistake was made. Worse, if the user passes an unacceptable action, *twice* does not exactly handle it gracefully. Instead, none of the IF statement clauses evaluate to TRUE and the function never executes a RETURN statement, bringing about the −6503 error.

This experience points out two glaring problems with *twice*, a function that just days ago I thought was, well, pretty solid. These problems are:

1. The function does not execute a RETURN statement in some cases. This is a big no−no, indicating that the structure of the function is very poorly designed.

2. I have made assumptions that I do not bother to validate in my program.

These faults can lead to unexpected program failure and must be corrected.

Interface Function

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8*i* Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

3.6 Building a Structured Function

Consider the problem of a function that does not execute a RETURN. The whole point of a function is to return a value. Not only should it return a value when everything goes right, it should even return a value when the function fails and raises an exception (NULL being the usual candidate under these circumstances).

In the *twice* function all my RETURN statements are nested inside IF clauses. So an invalid entry by the user means that all those RETURNs are ignored. There are lots of ways to fix this specific problem. You could include an ELSE statement. You could make sure that the action was valid at the start of the function (we'll look at that in a moment). The best all−around solution, however, is to always construct your functions with the following templated structure:

```
FUNCTION twice RETURN VARCHAR2
IS
   v_retval VARCHAR2(100) := 'null';
BEGIN

   RETURN v_retval;

EXCEPTION
   WHEN OTHERS
   THEN
      RETURN NULL;
END twice;
```

In this template I declare a local variable (the return value or **`v_retval`**) with the same datatype as the function itself. I then always make the last line of the function a RETURN of the **`v_retval`** variable's value. In addition, my exception returns NULL if any kind of exception is raised. You will never get a −6503 error with this template −− and it is easier to debug than functions with RETURN statements scattered throughout the body of the program.

A version of *twice* that follows the template is shown in Example 3.4. Now I have a return value variable as the last line of the function body. To do this, I simply replaced each of the individual RETURN statements inside the IF statement with an assignment to **`v_retval`**. I have not, therefore, added any kind of special handling for invalid actions. Yet I no longer have to worry about −6503, because I have chosen a structure for my function that automatically rules out that possibility. Furthermore, it even returns a sensible value in the case of a bad action code. The **`v_retval`** is initialized by PL/SQL to NULL. If the user passes a code like BS, the value of **`v_retval`** will not be changed and, as a result, NULL will be returned, indicating an incorrect value (or, come to think of it, NULL input).

Example 3.4: A Template−based twice Function

```
CREATE OR REPLACE FUNCTION twice
   (string_in IN VARCHAR2,
    action_in IN VARCHAR2 DEFAULT 'N')
RETURN VARCHAR2
IS
   v_action VARCHAR2(10) := UPPER (action_in);
   v_retval VARCHAR2(100);
BEGIN
   IF v_action = 'UL'
   THEN
      v_retval := UPPER (string_in) || LOWER (string_in);

   ELSIF v_action = 'LU'
   THEN
      v_retval := LOWER (string_in) || UPPER (string_in);

   ELSIF v_action = 'N'
   THEN
      v_retval := string_in || string_in;
   END IF;
   RETURN v_retval;
EXCEPTION
   WHEN OTHERS
   THEN
      RETURN NULL;
END twice;
/
```

PREVIOUS | HOME | NEXT
3.5 Rough Waters Ahead | BOOK INDEX | 3.7 Handling Program Assumptions

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

3.7 Handling Program Assumptions

Now let's address the problem of invalid action codes. You've already seen the downside: the user is not notified of an invalid entry; the program simply failed with −6503. With my latest version of *twice*, you no longer get the error. On the other hand, the function now returns the same value if you pass in a NULL string or if you pass in a bad action code. This is not a good way for a function to notify the user of errors. And that is because I do not explicitly handle an underlying assumption of my program.

Just about every piece of software you write makes assumptions about the data it manipulates. For example, parameters may have only certain values or must be within a certain range; a string value should have a certain format, or perhaps an underlying data structure is assumed to have been created. It's fine to have such rules and assumptions, but it is also very important to verify, or assert, that none of the rules are being violated. Because if you assume it and you don't check, your program can end up acting very strangely.

In the *twice* function, I assume that you, the user, know that you use *UL* for UPPER−lower, *LU* for lower−UPPER, and *N* for no case conversion. But how are you supposed to know this? You either have to see the source code, which is not always going to be possible or desirable, or you have to be given external documentation about the function. And even if you read the documentation on Monday, who says you are going to remember it on Friday?

If a low−level utility like *twice* is going to be successfully reused, it has to have the smarts built into it to check for bad actions and inform the user of the problem. The best way to do this is to assert that the incoming argument is correct. The following line of code asserts, for example, that the action code is correct. If not, it raises an exception.

```
IF v_action NOT IN ('UL', 'LU', 'N')
THEN
   RAISE VALUE_ERROR;
END IF;
```

If the action is valid, then *twice* would function as it normally does. Now if the action code is invalid, an exception is raised and no value is returned from the function. Is this a violation of my recommendation that a function always return a value? I would suggest that in this case an exception is more appropriate. The use of *twice* is invalid if it is not passed a valid code. In this context, it doesn't even make sense to continue processing. This is not the kind of error that occurs in production. My IF statement uncovers a design−level error in the code that must be corrected before you can even worry about data entry errors or other application−level concerns.

One problem with the IF statement is that it doesn't really inform the user about the problem. It just raises a generic, system exception. I think that if you are going to assert assumptions, you should display some feedback when the assumption is not met. Furthermore, I suggest that instead of building IF statements like this throughout your code, you create a single assert procedure like the one shown in Example 3.5. This program accepts the Boolean expression that needs to be true and a string to be displayed in case of failure.

Example 3.5: A Very Generic Assertion Routine

```
PROCEDURE assert
    (bool_in IN BOOLEAN, stg_in IN VARCHAR2 := NULL)
IS
BEGIN
   IF NOT bool_in OR bool_in IS NULL
   THEN
      IF stg_in IS NOT NULL
      THEN
         DBMS_OUTPUT.PUT_LINE (stg_in);
      END IF;
      RAISE VALUE_ERROR;
   END IF;
END;
```

With the **`assert`** routine added to my arsenal, I now have a very robust *twice* function (see Example 3.6). If another codeveloper tries the same BS from an anonymous block in SQL*Plus, here is the feedback she will receive:

```
Please enter UL LU or N
declare
 *
ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error
```

With my assert program in place, I spend less time on telephone support for *twice*. And if someone does call, I will tell them to "RTFM!", as in: "Read The Fancy Message!"

Example 3.6: Using an Assertion Routine Inside twice

```
CREATE OR REPLACE FUNCTION twice
   (string_in IN VARCHAR2,
    action_in IN VARCHAR2 DEFAULT 'N')
RETURN VARCHAR2
IS
   v_action VARCHAR2(10) := UPPER (action_in);
   v_retval VARCHAR2(100);
BEGIN
   assert
      (v_action IN ('UL', 'LU', 'N'),
       'Please enter UL LU or N');
   IF v_action = 'UL'
   THEN
      v_retval := UPPER (string_in) || LOWER (string_in);

   ELSIF v_action = 'LU'
   THEN
      v_retval := LOWER (string_in) || UPPER (string_in);

   ELSIF v_action = 'N'
   THEN
      v_retval := string_in || string_in;
   END IF;
   RETURN v_retval;
END twice;
/
```

PREVIOUS | HOME | NEXT

3.6 Building a Structured Function | BOOK INDEX | 3.8 Broadening the Scope

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8*i* Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

3.8 Broadening the Scope

Surely I am done with *twice* now. It is well structured, handles errors gracefully, and offers a reasonable amount of flexibility. It has come a long way from its original one−line version. So I would have to say that, yes indeed, I am done with *twice*. But a few days of programming go by and I encounter a very interesting requirement:

Take a string and return it repeated it three times, not just twice!

Of course, I instantly think of *twice* and how it would be very easy to create another function called *thrice* that performs an additional concatenation −− but that otherwise is unchanged. But then I take a coffee break and realize in my moment away from the screen (excellent thinking time −− I recommend it to all my readers!) that tomorrow I could run into a need for four repetitions and then five. The *twice* function is finished −− but only within its limited scope. What would be really great is a function that allows me to perform any number of duplications, as specified by the user. Now that would be a neat little function. So let's build it.

First of all, since I am going to let the user specify the number of repetitions, I will need to: (a) change the name of the function and (b) add a third parameter. Here is the new header for my new function:

```
CREATE OR REPLACE FUNCTION repeated
   (string_in IN VARCHAR2,
    action_in IN VARCHAR2 DEFAULT 'N',
    num_in IN INTEGER := 1)
RETURN VARCHAR2
```

The name of the function reflects its general utility. It returns a string repeated any number of times. The third parameter, **`num_in`**, indicates the number of times to repeat the string. Notice that the default is 1, which means a single repetition −− thereby matching the functionality of *twice*. Otherwise the parameter list is the same.

It probably won't take much thought on your part to realize two things about the implementation of *repeated*:

1. I can use a numeric FOR loop in a very straightforward way to create a string which repeats a substring N times.

2. The case conversion logic that applied itself so clearly in *twice* is less obvious now. If users specify UPPER−lower, do they want UPPER−lower−UPPER−lower or do they want UPPER−lower−lower−lower?

There is only one answer to this question: I don't know. A different user may want or expect a different outcome. As the creator of **`repeated`**, I can either build the function to handle both these two scenarios and other case conversion options, or simply decide that *repeated* will offer only one option.

In this chapter, I implement *repeated* in such a way that its case conversion is limited to applying the first half of the conversion to the input string and second half of the conversion to all the repetitions of that string. The following example shows what *repeated* will do:

```
SQL> exec DBMS_OUTPUT.PUT_LINE (repeated ('abc','UL',2));
ABCabcabc
SQL> exec DBMS_OUTPUT.PUT_LINE (repeated ('abc','LU',2));
abcABCABC
```

I will leave it to my readers to come up with an implementation of *repeated* that offers other patterns (or all patterns).[3] The full implementation of *repeated* is shown in Example 3.7. Here I step through that implementation.

[3] Please send me your solutions at Compuserve 72053,441.

The first thing I want to do in *repeated* is assert the validity of all of my assumptions. I have the same assumption for action that *twice* did, but I have another assumption as well: that the **`num_in`** argument will not be negative. So *repeated* will add this call to assert:

```
assert
   (num_in >= 0, 'Duplication count must be at least 0.');
```

Once I know that my arguments are all right, I can proceed to my algorithm. With my new approach to case conversion, I have two different kinds of strings for repetition: the initial string and the repetition string. The cases of these two strings need to be set separately (as you read this section, see if you can tell how *twice* is only a special case of this logic), based on the action code. I do this in the following IF statement:

```
   IF v_action = 'UL'
   THEN
      initval := UPPER (string_in);
      nextval := LOWER (string_in);
   ELSIF v_action = 'LU'
   THEN
      initval := LOWER (string_in);
      nextval := UPPER (string_in);
   ELSE
      initval := string_in;
      nextval := string_in;
   END IF;
```

Once I have set the initial and repetition (or next) strings, I can set the initial value for the return value and then use a FOR loop to generate the repeated string:

```
   v_retval := initval;
   FOR dup_ind IN 1 .. num_in
   LOOP
      v_retval := v_retval || nextval;
   END LOOP;
```

And the return value variable is then ready to be RETURNed by the function.

Example 3.7: The repeated Function

```
CREATE OR REPLACE FUNCTION repeated
   (string_in IN VARCHAR2,
    action_in IN VARCHAR2 DEFAULT 'N',
    num_in IN INTEGER := 1)
RETURN VARCHAR2
IS
   v_action VARCHAR2(10) := UPPER (action_in);
   initval VARCHAR2(32767);
   nextval VARCHAR2(32767);
   v_retval VARCHAR2(32767) := string_in;
```

```
BEGIN
   assert
      (v_action IN ('UL', 'LU', 'N'),
       'Please enter UL LU or N');
   assert
      (num_in >= 0,
       'Duplication count must be at least 0.');

   IF v_action = 'UL'
   THEN
      initval := UPPER (string_in);
      nextval := LOWER (string_in);
   ELSIF v_action = 'LU'
   THEN
      initval := LOWER (string_in);
      nextval := UPPER (string_in);
   ELSE
      initval := string_in;
      nextval := string_in;
   END IF;

   v_retval := initval;
   FOR dup_ind IN 1 .. num_in-1
   LOOP
      v_retval := v_retval || nextval;
   END LOOP;
   RETURN v_retval;
END duploop;
/
```

Now that *repeated* is coded, let's walk through that code for some specific argument values to see if my logic holds up.

3.8.1 When the num_in Argument Is 0

This is a *boundary check*. Zero is the lowest allowable value for **`num_in`**. Any test case that lies on the boundary of a range of values is a prime candidate for failure. How often have you written an algorithm that works fine in general, but which breaks down exactly on the low or high end or another kind of special case that is perfectly valid?

When **`num_in`** is 0, the FOR loop does not execute even once. Therefore, the return value is set to the initial value and that is what is returned: the string passed in by the user, converted as specified, repeated zero times.

The *repeated* function checks out for **`num_in`** equal to 0. Of course, I should and do execute the function for this case as well, but the code walkthrough comes first. You should be able to deduce logically that your code runs fine before you run it. Here goes:

```
SQL> exec DBMS_OUTPUT.PUT_LINE (repeated ('abc','UL',0));
ABC
```

3.8.2 When string_in Is NULL

Another special case. The *twice* function handles it smoothly since NULL concatenated to NULL is still NULL. Will *repeated* act any differently? The answer is no. It might execute more concatenations, but it still will return NULL when a NULL string is passed to it for the first argument.

In addition to my code walkthroughs for these cases, I executed *repeated* for a variety of different inputs and it seems to work just fine. I now have a very generic function to generate string repetitions with case conversion.

PREVIOUS
3.7 Handling Program Assumptions

HOME
BOOK INDEX

NEXT
3.9 Considering Implementation Options

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

3.9 Considering Implementation Options

That's right. We're still not finished. As you build more and more complex programs, you should always remember the following piece of wisdom:

In PL/SQL, just as in SQL, there is always more than way to implement a requirement.

SQL is notorious (at least to me, since I am far from a SQL guru) for having many different potential solutions for a single data request. Usually 99.95% of these solutions perform horribly. The situation is similar, though perhaps less extreme, for the PL/SQL language. You can usually find at least two or three ways to solve a given problem. And it is usually worth the trouble to at least consider more than one implementation. Why? First, even though your instinctive (first) approach may work fine, another technique might be even better from the standpoints of performance, maintainability, or readability. Second, by coming at the problem from another angle, you might well uncover logical flaws in your prior implementation(s), such as unconsidered cases.

Let's take another look at the *repeated* function with these issues in mind. Actually, as wrote the last sentence of the previous section, I was immediately reminded of another implementation. That sentence read, in part:

"I now have a very generic function to generate string repetitions..."

I thought to myself: "Big deal. I mean, PL/SQL itself has a very generic function or two to generate repetitions of a string." These builtin functions don't perform case conversion, so I don't feel downright stupid writing my own string−repeater. It is worth, however, considering those builtins for use within the repeated function.

Which functions do I refer to as "string−repeaters"? RPAD and LPAD. These pad functions are commonly used to pad on the left or right with spaces. Yet that is simply the default mode of operation for these functions. You can pad to the specified length with any pattern of characters you want. The following use of LPAD, for example, pads the string "Eli" with the words "My son" to a length of 20 characters:

```
SQL> exec DBMS_OUTPUT.PUT_LINE (LPAD ('Eli', 20, 'My son '));
My son My son My Eli
```

Notice that it stuck "My" in three times. That's because it pads as far as possible to fill the 20 characters and then stop. I can put this builtin repeater to work quite easily in the **repeated** function. The only trick is to calculate the total length of the string I want to generate. Example 3.8 contains the full implementation of the RPAD version of *repeated*.

Example 3.8: An RPAD−Based Implementation of repeated

```
CREATE OR REPLACE FUNCTION rep_rpad
   (string_in IN VARCHAR2,
    action_in IN VARCHAR2 DEFAULT 'N',
    num_in IN INTEGER := 1)
RETURN VARCHAR2
IS
```

```
   v_action VARCHAR2(10) := UPPER (action_in);
   initval VARCHAR2(32767);
   nextval VARCHAR2(32767);
   v_retval VARCHAR2(32767);

BEGIN
   assert
      (v_action IN ('UL', 'LU', 'N'),
       'Please enter UL LU or N');
   assert
      (num_in >= 0,
       'Duplication count must be at least 0.');

   IF v_action = 'UL'
   THEN
      initval := UPPER (string_in);
      nextval := LOWER (string_in);
   ELSIF v_action = 'LU'
   THEN
      initval := LOWER (string_in);
      nextval := UPPER (string_in);
   ELSE
      initval := string_in;
      nextval := string_in;
   END IF;

   v_retval := RPAD (initval, LENGTH (string_in) * (num_in+1), nextval);

   RETURN v_retval;
END rep_rpad;
/
```

It is exactly the same as the FOR loop version, except that in place of the loop, I use this line:

```
v_retval := RPAD (initval, LENGTH (string_in) * (num_in+1), nextval);
```

The total length of the return value is the length of the specified string multiplied by the number of repetitions plus one. So if the user specifies zero repetitions, the total length is the same as the original string, and RPAD does nothing. If the user wants one repetition, the total length is double the original, leaving enough room for RPAD to pad **`initval`** on the right with *nextval* just once −− resulting in twice the original string. This pattern works for additional multiples as well.

The RPAD approach requires fewer lines of code than the loop version. For example, with RPAD I don't even have to initialize the return value variable to *initval*. The single assignment covers the **`num_in`** = 0 case as well as the non−trivial repetitions. Which technique should I use? More to the point, which should I make available to others to use?

The deciding factor in this case should be: which is more efficient? This is a low−level utility. It might be called many times deep down in the bowels of an application. So a minor difference in performance between the two implementations could have a multiplying effect on overall performance of the application.

PREVIOUS | HOME | NEXT
3.8 Broadening the Scope | BOOK INDEX | 3.10 Choosing the Best Performer

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

3.10 Choosing the Best Performer

If there is a difference in execution time between the performance of my two string−repeaters, it will not be a big one. I will need to execute the functions many times to compare the cumulative difference. The best way to calculate the elapsed time of PL/SQL code execution is with the GET_TIME function of the DBMS_UTILITY builtin package. I have encapsulated GET_TIME inside the PLVtmr package (PL/Vision TiMeR) to make it easier to use. Example 3.9 shows the kind of script I used.[4]

> [4] By the way, this code was for the most part generated for me with the PLVgen package to compare the performance of **`repeated`**, **`rep_rpad`**, and also the recursion−based implementation of **`repeated`** (see sidebar).

This SQL*Plus script (stored in the file *timerep.sql* on the disk) takes three arguments. The first, *&1*, accepts the number of times to execute each function. The second, *&2*, accepts a string that is to be duplicated. The third, *&3*, accepts the number of repetitions of the string. I ran the script several times as shown below:

```
SQL> @timerep 100 abc 1
duprpad Elapsed: .77 seconds. Factored: .0077 seconds.
duploop Elapsed: .66 seconds. Factored: .0066 seconds.
recrep Elapsed: .71 seconds. Factored: .0071 seconds.

SQL> @timerep 100 abc 10
duprpad Elapsed: .71 seconds. Factored: .0071 seconds.
duploop Elapsed: .99 seconds. Factored: .0099 seconds.
recrep Elapsed: 1.54 seconds. Factored: .0154 seconds.
```

I ran each of these tests several times to allow the numbers to stabilize. The results are very interesting and certainly reinforce the need for a careful test plan. When repeating the string just once, the recursion−based implementation is superior. Upon reflection, this should not be a surprise. It handles a single repetition as a special case: an unmediated concatenation of two strings. The loop−based implementation comes in second, but all of the timings are very close. When we move to multiple repetitions of the string, however, the *recrep* function becomes extremely slow; again, I would expect that behavior because of the extra work performed by the PL/SQL runtime engine to manage a recursive program. The big news from this round, however, is that the RPAD implementation of **`repeated`** establishes itself clearly as the fastest technique.

Example 3.9: A Performance Comparison Script

```
DECLARE
   a VARCHAR2(100) := '&2';
   aa VARCHAR2(10000);
BEGIN
   PLVtmr.set_factor (&1);
   PLVtmr.capture;
   FOR rep IN 1 .. &1
   LOOP
      aa := rep_rpad (a, 'UL', &3);
   END LOOP;
   PLVtmr.show_elapsed ('duprpad');
```

```
      PLVtmr.set_factor (&1);
      PLVtmr.capture;
      FOR rep IN 1 .. &1
      LOOP
         aa := repeated (a, 'UL', &3);
      END LOOP;
      PLVtmr.show_elapsed ('duploop');

      PLVtmr.set_factor (&1);
      PLVtmr.capture;
      FOR rep IN 1 .. &1
      LOOP
         aa := recrep (a, 'UL', &3);
      END LOOP;
      PLVtmr.show_elapsed ('recrep');
   END;
   /
```

Using Recursion to Repeat the String

Just when you think you've covered the bases, someone comes along and shows you a new way. I often use the *twice* function in my classes to demonstrate the development spiral. While training a group of 35 students at the Oracle Netherlands training center in De Meern, I reached the point where it was time to expand the scope of *twice* to allow any number of repetitions of the string. So I asked my class for some ideas. Immediately, a quiet voice piped up from the first row: "Use recursion." Recursion? It had never crossed my mind. I must admit that recursion is not an approach to which my brain readily turns. But it certainly seemed like a logical approach to take with the *repeated* function.

Never one to scorn a student's idea, we quickly cobbled together the version of *repeated* you see in Example 3.10. It is called *recrep* for RECursive REPeater.

Of course, I also need to compare the performance for different kinds of strings. I ran the same **`timer`** script as follows to see how each function handled NULL values:

```
SQL> @timerep 200 null 10
duprpad Elapsed: 1.59 seconds. Factored: .00795 seconds.
duploop Elapsed: 2.03 seconds. Factored: .01015 seconds.
recrep Elapsed: 2.91 seconds. Factored: .01455 seconds.
```

In this scenario, the RPAD implementation was considerably faster than the loop and recursion techniques (though, once again, I found that if the number of repetitions was set to 1, the **`recrep`** function was faster). Finally, I greatly increased the number of string repetitions and then all became clear:

```
SQL> @timerep 100 abc 100
duprpad Elapsed: .77 seconds. Factored: .0077 seconds.
duploop Elapsed: 4.28 seconds. Factored: .0428 seconds.
recrep Elapsed: 5.22 seconds. Factored: .0522 seconds.
```

The conclusion I draw from my tests is that the RPAD technique offers a much more stable solution than that based on the FOR loop. Regardless of the number of repetitions, RPAD takes about the same amount of time. With the FOR loop and recursion approaches, as the repetitions increase, the performance degrades. That is not the sign of a healthy algorithm.

Given the results, it would make sense to implement the **`repeated`** function using the RPAD technique. You could possibly optimize further by using the FOR loop approach for small numbers of repetitions, and then switch to RPAD for larger repetitions. The gain with the FOR loop for minimal repetitions is, however, minimal −− it's probably not worth the trouble.

I was glad to see that the RPAD approach is faster. You should always use a builtin if it exists, rather than build your own. The FOR loop technique arose quite naturally from the way I expanded the scope of the *twice* function. It turned out, however, that it was not the path to the optimal solution. As for recursion, well, it is always an interesting phenomenon to watch and puzzle out, but it rarely offers the best implementation (except when it is the *only* implementation feasible).

Example 3.10: The Code for the Recursive Implementation of repeated

```
CREATE OR REPLACE FUNCTION recrep
   (string_in IN VARCHAR2,
    action_in IN VARCHAR2 := NULL,
    num_in IN INTEGER := 1)
   RETURN VARCHAR2
IS
   v_action VARCHAR2(10) := UPPER (action_in);
   initval VARCHAR2(32767);
   nextval VARCHAR2(32767);
   v_retval VARCHAR2(32767);

BEGIN
   assert
      (v_action IN ('UL', 'LU', 'N'),
       'Please enter UL LU or N');
   assert
      (num_in >= 0, 'Duplication count must be at least 0.');

   IF v_action = 'UL'
   THEN
      initval := UPPER (string_in);
      nextval := LOWER (string_in);
   ELSIF v_action = 'LU'
   THEN
      initval := LOWER (string_in);
      nextval := UPPER (string_in);
   ELSE
      initval := string_in;
      nextval := string_in;
   END IF;

   IF num_in = 1
   THEN
      RETURN initval || nextval;
   ELSE
      /* No more case conversions performed... */
      RETURN (initval || repeated (nextval, 'N' , num_in-1));
   END IF;
END recrep;
/
```

PREVIOUS
3.9 Considering Implementation Options

HOME
BOOK INDEX

NEXT
3.11 Don't Forget Backward Compatibility

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8*i* Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

3.11 Don't Forget Backward Compatibility

Now that I have stabilized a version of *repeated* that performs best, I have one more issue to consider: what about all those calls to the *twice* function? The *repeated* function (whichever implementation I go with) handles the same requirement as that covered by *twice*. I would rather not have several different functions floating around in my environment, especially since they duplicate lots of the same logic. For example, if I decide to add yet another kind of case conversion, such as InitCap, I would have to enhance both the *twice* and the *repeated* functions. That is a real bummer, from a maintenance standpoint.

I do not, on the other hand, necessarily want to get rid of the *twice* function. It is already used in a number of programs, some of which are in production. I would much rather leave the calls to *twice* in place and thereby minimize the disruption to existing code. I need a path that offers backward compatibility while at the same time avoids a maintenance nightmare.

The solution is a direct translation to code of that stated need: keep the header to *twice* the same, but completely gut and replace its internals with...a call to *repeated*! This approach is shown here:

```
CREATE OR REPLACE FUNCTION twice
   (string_in IN VARCHAR2, action_in IN VARCHAR2 DEFAULT 'N')
RETURN VARCHAR2
IS
BEGIN
   RETURN (repeated (string_in, action_in, 1));
END;
```

I could leave off the third argument of 1, since that is the default and I explicitly designed the function so that the default would match the current functionality of *twice*. That is, however, a dangerous approach. What if the default changes? You are much better off being explicit -- especially since I do not really want the default value. I want a single repetition. That just happens to be the default -- today.

Now all of the programs that call *twice* will work as is -- no changes required. Yet any changes I make to *repeated* will automatically carry into the *twice* function as well.

PREVIOUS | HOME | NEXT
3.10 Choosing the Best Performer | BOOK INDEX | 3.12 Obliterating the Literals

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

3.12 Obliterating the Literals

There are still two things that bother me about the *repeated* function: first, the function is not defined in a package and, second, a user of repeated has to know the correct literals to pass to it to get the right kind of conversion action. On general principles, I believe that everything built in PL/SQL should be placed inside a package. This construct is the cornerstone of programming in the PL/SQL language and offers many advantages, explored in detail throughout this book. My second concern about literals can be answered by creating a package −− so I will show you how to convert the standalone **`repeated`** function into a package.

I do not believe that a user of my code should have to remember the specific characters to pass in a string literal. Is it *UL* or *BS*? Is it *n* for "no action" or *l* for "leave alone"? With the function as implemented throughout this chapter, there is no way for a developer to know *at compile time* if she called *repeated* properly.

Beyond this difficulty, applications the world over would be much better off if their creators avoided the use of hard−coded literals in their code. Every time the *repeated* function is called, some string literal is being hard−coded into a program. If the *repeated* function is ever modified to expand the scope of action and different literals are used, all those other programs could go haywire. A much better approach would provide *named constants* in place of the hard−coded strings so that (a) at compile time a developer would know if the call to the function is correct and (b) the actual string values for the action codes can be hidden from view −− and changed as often as is necessary.

The best way (really, the only way) to create named constants for use throughout a PL/SQL application is to put these constants −− and the code with which they are used −− into a package. The *stg* package shown in Example 3.11 offers the same functionality as the repeated function, with the additional benefit of named constants. Now instead of having a standalone repeated function, I have a *dup* function in the *stg* package, and the following constants:

stg.ul

Indicates that you want UPPER−lower case conversion

stg.lu

Indicates that you want lower−UPPER case conversion

stg.n

Indicates that you do not want any case conversion

So when I want to duplicate or repeat the string "abc" 10 times with UPPER−lower conversion, I would execute this statement:

```
stg.dup ('abc', stg.ul, 10);
```

By referencing the *stg.ul* constant, I can verify at compile time that I am using a valid action code for case conversion.

Notice that I have placed the *dup* function within a very generic string package. I do this to anticipate future requirements for string processing. By creating this package, I have established a repository in which I can place other, related functions and procedures as I think of them. All will be called with the "stg" prefix, indicating that they are oriented to string processing.

Example 3.11: A Duplicate String Package

```
CREATE OR REPLACE PACKAGE stg
IS
   lu CONSTANT VARCHAR2(1) := 'A';
   ul CONSTANT VARCHAR2(1) := 'B';
   n  CONSTANT VARCHAR2(1) := 'X';

   FUNCTION dup
      (stg_in IN VARCHAR2,
       action_in IN VARCHAR2 := n,
       num_in IN INTEGER := 1)
   RETURN VARCHAR2;
END stg;
/
CREATE OR REPLACE PACKAGE BODY stg
IS
   FUNCTION dup
      (string_in IN VARCHAR2,
       action_in IN VARCHAR2 DEFAULT n,
       num_in IN INTEGER := 1)
   RETURN VARCHAR2
   IS
      v_action VARCHAR2(10) := UPPER (action_in);
      initval VARCHAR2(32767);
      nextval VARCHAR2(32767);
      v_retval VARCHAR2(32767);

   BEGIN
      assert
         (v_action IN (lu, ul, n),
          'Please use the package constants: ul, lu or n');
      assert
         (num_in >= 0, 'Duplication count must be at least 0.');

      IF v_action = ul
      THEN
         initval := UPPER (string_in);
         nextval := LOWER (string_in);

      ELSIF v_action = lu
      THEN
         initval := LOWER (string_in);
         nextval := UPPER (string_in);

      ELSE
         initval := string_in;
         nextval := string_in;
      END IF;

      v_retval :=
         RPAD (initval, LENGTH (string_in) * (num_in+1), nextval);

      RETURN v_retval;
   END dup;
END stg;
/
```

PREVIOUS HOME NEXT
BOOK INDEX

3.11 Don't Forget Backward Compatibility

3.13 Glancing Backward, Looking Upward

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8*i* Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

3.13 Glancing Backward, Looking Upward

Now that was a journey through the valleys and peaks of modularization! I started with a simple solution to what seemed to be a very simple request. I ended up with a very generic, well−structured function that handles the simple request and many others as well.

Along the way, I applied many best practices of recommendations for module construction (some of which are covered in Chapter 2, *Best Practices for Packages*, some of which are discussed in *Oracle PL/SQL Programming*). With each successive change to *twice* (and then *repeated*), I took another turn up along the spiral that represents the rise in quality of my PL/SQL coding techniques. At the end, I had a polished function with proven performance and wide applicability. Take a look at the final version of my repeater function (the *dup* package). Could you have ever predicted that endpoint from the first version of the *twice* function? I certainly could not have. In fact, when I started this chapter, the *repeated* function looked quite different from the way it does now. I found many improvements to make from my first progression of *twice* (a thorough improvisation performed "live" during a class in Tulsa, Oklahoma) as I "rationalized" the code into an chapter.

And so we come face to face with one of the most extraordinary characteristics of the programming spiral. It's not like a Slinky. That toy has the right shape, but it also has a beginning and an end. The spiral for developers has a beginning (though you would probably have to make an arbitrary choice to locate it), but it certainly has no end. You can always find ways to improve your code, your coding philosophy, and your quality of programming life.

So the next time you sit down to write a program, don't settle for "getting the job done." Instead, push yourself up that spiral towards excellence.

4. Getting Started with PL/Vision

Contents:

4.1 What Is PL/Vision?

As I've mentioned in earlier chapters, PL/Vision is a collection of PL/SQL packages and supporting SQL*Plus scripts that can change radically the way you develop applications with the PL/SQL language. This chapter describes the product in greater detail, lists the packages included in it, and provides instructions for installing the PL/Vision Lite Online Reference provided on the companion disk.

4.1.1 The Benefits of PL/Vision

What can PL/Vision do for you? The possibilities are almost endless:

- *Improve your productivity.* PL/Vision goes a long way towards helping you avoid reinventing the wheel. Need to change a long string into a word−wrapped paragraph? Use the *PLVprs.wra* procedure. Want to display the contents of a PL/SQL table? Call the *PLVtab.display* procedure. Need to log activity to the database or maybe even write your information to a PL/SQL table? Call the **`PLVlog.put_line`** program. By using PL/Vision, you write much less code yourself, and instead spend your time deciding which prebuilt components of PL/Vision to plug into your own applications. You are able to focus much more of your effort on implementing the business rules for your systems.

- *Decrease the number of bugs in your code and fix the bugs you do find more rapidly.* Since you will be writing less code, you will minimize the opportunities for bugs to creep into your own programs. When you do have compile errors, you can call the *PLVvu.err* program to show you precisely where the error occurred in your program. Furthermore, PL/Vision packages offer many ways to trace the actions taken within those packages. When you need more information, you simply call the appropriate package toggle to turn on a trace and then run your test. When you are done testing and debugging, you turn off the trace mechanisms and put your application into production.

- *Help you develop and enforce coding standards and best practices.* You will do this in two ways: first, by using packages that explicitly support coding standards, such as PLVgen; second, by examining the code behind PL/Vision. This PL/SQL library has provided numerous opportunities for me to put my own best practices for packages into action. You will, no doubt, find occasional violations of my best practices, but by and large the code in PL/Vision should provide a wealth of ideas and examples for your own development.

- *Increase the percentage of reusable code in your applications.* The more you leverage PL/Vision, the fewer new programs you have to write yourself. And this advantage doesn't just accrue to individual developers. You can use PL/Vision across multiple applications −− it can be part of a truly enterprise−wide object and module library.

-

Demonstrate how to modularize and build layers. I don't want you to simply use PL/Vision. I want you to learn how and why I built PL/Vision so that you can accomplish the same kind of development yourself. We all need to be fanatically devoted to modularizing code for maximum reusability. We all need to become sensitive to identifying program functionality that should be broken out into different layers. To some extent, you can develop such sensitivity only by practicing the craft of software construction. But you can also examine closely the work of others and learn from their example (both the good and the bad).

- *Inspire you to be creative, to take risks in your coding.* I have found that the real joy of programming is to be found in trying out new ways of doing things. When you stretch boundaries −− whether they are the boundaries of your own experience or those of the documented features of a language −− you make discoveries. And when those discoveries turn out to be productive, you create new things.

4.1.2 The Origins of PL/Vision

PL/Vision has both top−down and bottom−up origins. Many of the pieces of PL/Vision were pulled together after the fact; I would come up with an interesting package, then draw it into the embrace of PL/Vision. After incorporating it I would reexamine other packages in PL/Vision to see how I could leverage this latest addition. I consider that the bottom−up approach.

PL/Vision as a coherent library of interlocking packages first sprang out of a recognition of the need to break up a single large program into multiple layers of code. In the fall of 1995 I became obsessed with the idea of writing a program that would "pretty−print" or reformat PL/SQL source code to follow my suggested coding style. It would read the code from the ALL_SOURCE data dictionary view and perform such tasks as upper−case all keywords and apply consistent indentation and line breaks to the code. I worked feverishly after hours on this project for a week and found myself with a working prototype: the *psformat* procedure. It was an exciting week for me. I could feed *psformat* (which ran to almost 1000 lines of text) the code for a procedure stuffed into one long, unformatted string and retrieve from it a nicely formatted PL/SQL program unit.

I was careful not to write a parser for the PL/SQL language. I didn't think such a step was necessary to handle my limited scope and I sure didn't want to spend the time required for such a task. Yet I found (as many of you would probably anticipate) that *psformat* didn't handle all the nuances of PL/SQL code correctly. So I would dive back in and tinker with it just a *little* bit more so that it would understand this or that element of the language.

Two weeks later, I had completely reimplemented *psformat* three times. With each implementation I came closer to handling the conversion and formatting of a wide range of PL/SQL program syntax. At the same time, my one big program grew that much bigger and more convoluted. I was just so taken up in implementing my obsession that I did not feel that I had the time to modularize my code. And each time I came that much closer to cobbling together a parser for PL/SQL.

After round three and too little sleep, I began to realize that I would probably *never* meet my objectives taking this piecemeal approach. Painfully and reluctantly, I gave up on this program (as you will see, PL/Vision does not provide a program to automatically indent and line−break your code). I did not, however, abandon all the code I had developed. Instead, I set my sights on a more limited and achievable goal: a program that would convert the case of source code to follow the UPPER−lower method. I would leave the format of the code as is, but change all keywords to upper−case and all application−specific identifiers to lower−case.

In the process of enhancing *psformat*, I had constructed a table that contained the keywords for the PL/SQL language and information about how those keywords affected the format of the source code (this is now the *PLV_token* table). I also recognized that I wanted to be able to read the original code from any number of different sources and write the converted code to any number of different targets.

At the same time that I shifted my expectations and goal, I forced myself to take the time to break up my monster procedure and employ top−down design to straighten out my logic. Thus was PL/Vision born. Directly out of this process, I created a single PLVtext package to manipulate PL/SQL source text. This single package eventually transformed itself into PLVio, PLVobj, and PLVfile to handle very generically a variety of source and target repositories for PL/SQL source code. I separated out many string−parsing tasks from *psformat* into PLVprs, which over time broadened into PLVprs, PLVlex, PLVtkn, and, finally, PLVprsps. When all of my layering and partitioning was done, I got back to building the PLVcase package to perform the actual case conversion. My monstrous single procedure became a very small, relatively simple package.

As I broke out these different aspects of PL/SQL functionality, I began to see the outlines of a whole substantially larger than the sum of its parts: a very generic and highly reusable toolbox that could be used in any PL/SQL−based development environment. From that point on, I was on the lookout for any utility or package I had built or would build that could add to the scope and functionality of PL/Vision.

Throughout this development process, I found myself learning more and more about what was really involved in building packages and leveraging reusable code. My conception of a set of best practices for packages also crystallized.

My desire to share what I had learned and built grew with the number of packages in PL/Vision. Finally, I realized that the best way to make the most of my knowledge was to write a book centered around the code and lessons of PL/Vision. I hope that you will find the software and the accompanying text useful.

PREVIOUS | HOME | NEXT
II. PL/Vision Overview | BOOK INDEX | 4.2 PL/Vision Package Bundles

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8*i* Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

PREVIOUS
Chapter 4
Getting Started with
PL/Vision
NEXT

4.2 PL/Vision Package Bundles

PL/Vision Lite consists of 32 PL/SQL packages. From a conceptual standpoint, the various packages are grouped into three bundles: building blocks, developer utilities, and plug−and−play components. These packages in each of these bundles are summarized in the following sections. Chapter 5, *PL/Vision Package Specifications*, contains a full summary of all package specifications.

> ***NOTE:*** Some of the packages listed in the following tables are not described in detail in this book. In cases in which neither the package conception nor its implementation offered any additional insights that would help you build your own packages, I elected to omit the discussion from the book. For more details about these packages (indicated by * in the tables), see the PL/Vision Online Reference on the companion disk.

4.2.1 Building Blocks

The building block packages provide low−level functionality upon which other packages in PL/Vision are built. You can, of course, also use these building blocks to construct your own applications. Examples of building block packages include an interface to PL/SQL tables, string parsers, a file I/O manager, and a message handling mechanism.

Table 4.1: PL/Vision Building Block Packages

Package	Description
p	Offers a feature−rich and minimal−typing substitute for the DBMS_OUTPUT package.
PLV	Top−level package for PL/Vision. Contains constants and generic utilities for use throughout the library.
PLVchr	Provides information about single characters in a string. You can display the ASCII codes for characters in a string and perform other operations relating to individual characters.*
PLVfile	Manages operating system file I/O. PLVfile provides a layer of code around the UTL_FILE builtin package and offers some very high−level capabilities, such as a file copy. Use this package only if you are using Oracle Server Release 7.3 and above.
PLVio	Generalized input/output package used to both read from and write to repositories for PL/SQL source code. For example, you can use PLVio, via the PLVcase package, to read a program from its operating system file, convert keywords to upper−case, and then store it in the database.
PLVlex	Lexical analysis and parsing package. Recognizes lexical elements of the PL/SQL language and can be used to read one PL/SQL identifier at a time.*
PLVlst	Generic list manager for PL/SQL that's built on PL/SQL tables. Following the specification of the LIST package of Oracle Developer/2000, this package provides a comprehensive interface to lists in PL/SQL.*

PLVmsg	Stores standard messages for use in an application. Use this package to consolidate all different kinds of message text with which you can associate a number (such as error number). You can also use PLVmsg to override standard Oracle error messages with your, more application−specific information.
PLVobj	Programmatic interface to the ALL_OBJECTS data dictionary view. This package encapsulates logic for specifying and reading source code for a given object. It provides an excellent model for building a package around a view or cursor. PLVobj even implements a kind of dynamic cursor FOR loop through a procedure with the **`loopexec`** program.
PLVprs	Performs string parsing actions. This is the most generic of the string manipulation packages of PL/Vision.
PLVprsps	Parses PL/SQL source code. You can parse a single string or an entire program unit. The parsed tokens are placed in a PL/SQL table, which can then be used as the basis for further analysis or conversion of the code.
PLVstk	Generic stack manager package, built on PLVlst. Provides a full set of operations for both FIFO (first−in−first−out) queues and LIFO (last−in−first−out) stacks.*
PLVtab	Provides an interface to predefined PL/SQL table structures. Also allows you to easily and flexibly display the contents of PL/SQL tables.
PLVtkn	Package interface to the PLV_token table, which contains information about tokens, particularly keywords, in the PL/SQL language. Use PLVtkn to determine if an identifier is a keyword or a reserved word, and even the type of keyword (syntax, builtin, symbol, etc.).

4.2.2 Developer Utilities

The developer utilities of PL/Vision are self−contained utilities that you can use to improve your development environment. Examples of building block packages include a PL/SQL code generator and an online help delivery mechanism for PL/SQL programs.

Table 4.2: PL/Vision Developer Utility Packages

Package	Description
PLVcase	Converts the case of PL/SQL code to the UPPER−lower method. You can convert a single token, a line, a program, or a set of programs.
PLVcat	Catalogues the contents of PL/SQL code, placing the results in one of two database tables. You can either catalogue the list of elements referenced by a particular program (the PLVrfrnc table) or the list of elements defined in the specification of a package (the PLVctlg table).
PLVddd	Dumps Data Definition Language (DDL) syntax from a particular schema. Allows you to recreate database objects easily in other schemas. You can also use output from PLVddd to compare data structures between schemas or analyze changes over time.*
PLVgen	Generates PL/SQL program units and SQL*Plus scripts. This package can greatly improve developer productivity, adherence to coding standards and best practices, and the overall quality of code produced.
PLVhlp	Provides an architecture by which developers can provide online help for their PL/SQL programs to their (developer) users. Using this package, you can make comment text in your source code available in a structured way to users of your code.
PLVtmr	Allows you to measure elapsed time of PL/SQL code down to the hundredth of a second. This package offers a programmatic layer around the GET_TIME function of the builtin DBMS_UTILITY package.

PLVvu	Multifaceted view package. Shows you the errors in a stored object compile, or specified lines of source code from the data dictionary, etc. Offers a convenient substitute for the SHOW ERRORS command of SQL*Plus.

4.2.3 Plug−and−Play Components

The most advanced packages in PL/Vision are the plug−and−play components. These packages allow developers to replace whole sections of code with programs from PL/Vision packages. In essence, you plug in PL/Vision code and immediately gain benefits in your application, employing a declarative style of programming in a procedural language. The best example of a PL/Vision plug−and−play component is PLVexc, which provides very high−level exception handling programs.

Table 4.3: PL/Vision's Plug−and−Play Packages

Package	Description
PLVcmt	Offers a programmatic interface to the execution of commits, rollbacks, and the setting of savepoints. Gives you more flexibility than calling the corresponding builtins. You can, for example, opt to turn off commits in your application without changing any of your code.
PLVdyn	Offers a high−level interface to the DBMS_SQL builtin package. You can perform many complex operations with a call to a single PLVdyn program. This package is strongly recommended over direct use of the DBMS_SQL builtin packages.
PLVdyn1	Built upon PLVdyn, this package encapsulates dynamic SQL operations that require single bind variables (PLVdyn does not work with any bind variables).
PLVexc	Generic exception−handling package. Instead of writing your own exception handlers, just call one of the PLVexc prebuilt, high−level handlers. Your errors will be displayed or written to the PL/Vision log, as you specify.
PLVfk	Provides foreign key management, including a single function to perform lookups of foreign keys for any table and any structure. This package can greatly reduce the volume of code you have to write to manage foreign keys. The dynamic SQL in PLVfk works surprisingly quickly.
PLVlog	This package provides a generic logging facility for PL/Vision−based applications. With PLVlog, you can write information to a database table, PL/SQL table, operating system file (for PL/SQL Release 2.3 and above), or standard output.
PLVrb	Provides a programmatic interface to rollback and savepoint processing. Allows you to specify savepoints by variable, instead of hard−coding identifiers in your code. You can also opt to turn off rollbacks in your application without changing any of your code.
PLVtrc	Provides an execution trace for PL/SQL programs. Mirrors the overloading of the **`p`** package to allow you to show different kinds of data. Offers **`startup`** and **`terminate`** procedures that allows PLVtrc to maintain its own execution call stack.

PREVIOUS | HOME | NEXT
4.1 What Is PL/Vision? | BOOK INDEX | 4.3 Installation Instructions

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built−in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built−ins Pocket Reference

4.3 Installation Instructions

This section describes how to install the PL/Vision Lite product and to use the PL/Vision Online Reference and the other files from the companion disk, located in the */advprog/disk/* directory of this CD.

4.3.1 What's On the Disk?

You'll see the following four files on the disk provided with this book:

disk.id
Contains label information (product name and version number)

plvlite.001
Compressed PL/Vision Lite software and online companion reference

read.me
Instructions for acquiring a non−Windows version of PL/Vision Lite

setup.exe
Executable file that installs the PL/Vision Lite program files (contained in plvlite.001).

4.3.2 Storage Requirements

To determine how much space you will need in the SYSTEM tablespace to install the PL/Vision packages, I ran the following script in my own PL/Vision account:

```
SELECT type,
       SUM (source_size) source_size,
       SUM (parsed_size) parsed_size,
       SUM (code_size) code_size
  FROM user_object_size
 WHERE name LIKE 'PLV%'
    OR name = 'P'
 GROUP BY type
 ORDER BY code_size DESC
/
```

This script accesses the USER_OBJECT_SIZE data dictionary to sum up the total size of source code (**`source_size`**), parsed code (**`parsed_size`**), and compiled code (**`code_size`**). The results are shown below (and may vary according to the final state of the code at time of publication):

```
SQL> start plvsize
SEF
TYPE                   SOURCE_SIZE          PARSED_SIZE        CODE_SIZE
--------------- ------------------ -------------------- ----------------
PACKAGE                      57840                92017            28795
PACKAGE BODY                266737                    0           312596
TABLE                            0                 2498                0
```

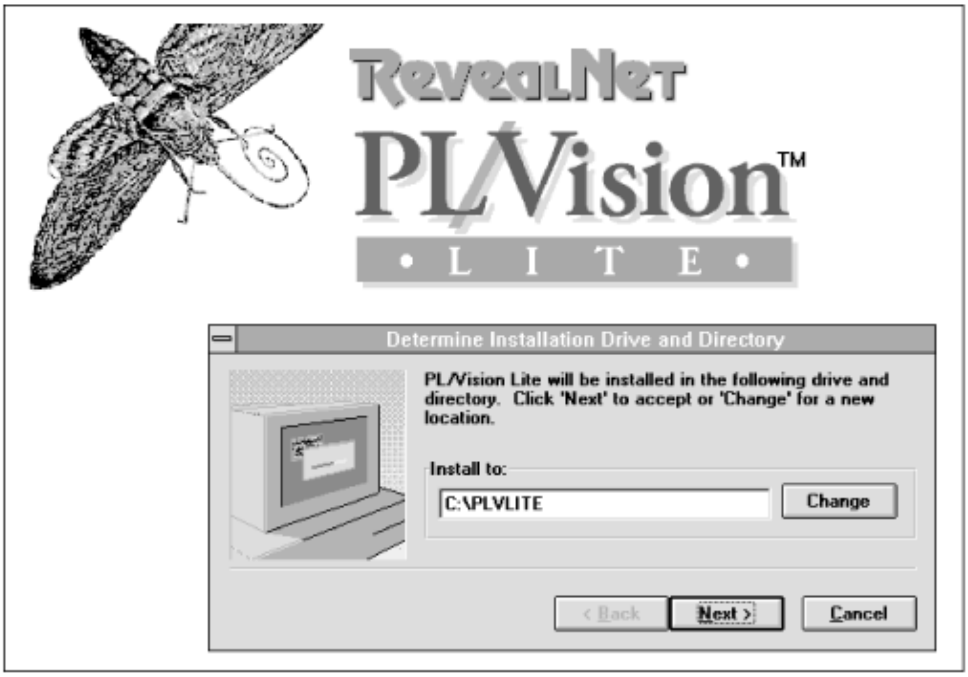
```
sum                          -------------       ------------       ----------
                                    324577              94515           341391
```

This output indicates that you will need a total of approximately 750,000 bytes in the SYSTEM tablespace to store PL/Vision.

4.3.3 Beginning the Installation

In a Windows environment, double−click on the **setup.exe** file to run the installation program. (If you are working in some other environment, see the **read.me** file for instructions.) The installation scripts will lead you through the necessary steps. The first screen you'll see is the install screen shown in Figure 4.1.

Figure 4.1: PL/Vision Lite install screen

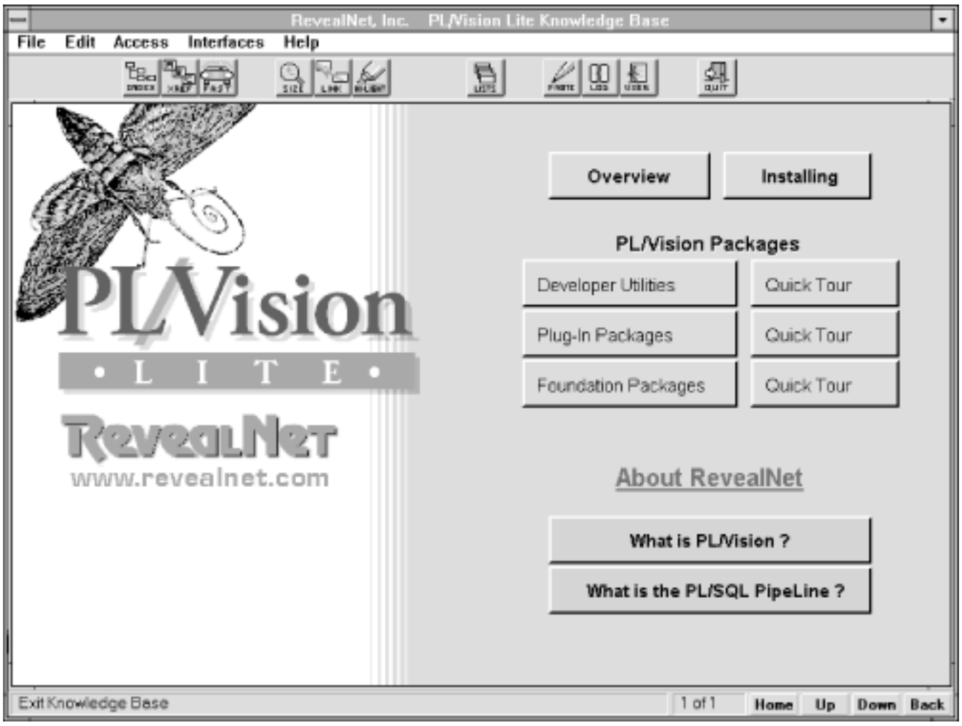


PL/Vision Lite will be installed in the following drive and directory. Click 'Next' to accept or 'Change' for a new location.

Install to:
C:\PLVLITE

The next screen will prompt you to specify the folder from which you want to be able to invoke PL/Vision. You can accept the default or specify the appropriate folder name (directory) for your own system. Once this step is complete, the icon named PL_Vision Lite Online appears in the folder you've specified. Double−click on it to start using the product. You'll see the main menu shown in Figure 4.2.

Figure 4.2: PL/Vision Lite main menu



At this point, all of the source files PL/Vision needs will be installed on your disk in the following three subdirectories.

`install`
: Contains source code for all of the packages of PL/Vision.

`test`
: Contains test scripts to execute PL/Vision code.

`use`
: Contains additional scripts to execute PL/Vision code more easily or perform other handy tasks.

Section 4.6, "Summary of Files on Disk", later in this chapter, lists these files.

You can now access the PL/Vision Lite Online Reference, as described in the next section. To actually execute PL/Vision code, however, you will need to compile the PL/Vision packages into your database. This process is described in Section 4.3.5, "Creating the PL/Vision Packages".

4.3.4 Using the PL/Vision Lite Online Reference

Now you're ready to use the PL/Vision Lite Online Reference to browse the code. There are two ways to use this reference:

1. Zoom in on the package of interest to obtain a general overview of the package or to find out the syntax for a particular element in the package.

2. View the source code for a PL/Vision package specification or body directly through the Online Reference. As you look at the directory of packages, you can press a button that will let you scroll through the PL/Vision source code.

I would recommend reading the book with the Online Reference up and running on your personal computer. As you come to a chapter about a PL/Vision package, bring up the source code for that package on your screen. That way you will be able to move instantly from a topic in the text to the code being referenced.

4.3.5 Creating the PL/Vision Packages

You can install PL/Vision in three different ways:

1. Install PL/Vision into and use it from a single Oracle account.
2. Install PL/Vision into each Oracle account that is going to take advantage of PL/Vision functionality.
3. Install PL/Vision into a single Oracle account and make its functionality available to other Oracle accounts through GRANTs and the creation of synonyms.

The first option is viable in small development environments. The second option requires lots of maintenance (you will have to reinstall or upgrade PL/Vision in each of the accounts). The third option is the one selected by most development organizations using PL/Vision.

If you do want to install PL/Vision into a single account and then share it across Oracle accounts, you should grant the SELECT ANY TABLE privilege to the PL/Vision account (a.k.a., PLVISION). If you do not take this step (shown below), you will find that several PL/Vision utilities will not work as desired.

Specifically, utilities like **`PLVvu.err`** (to display compile errors), **`PLVvu.code`** (to display stored code) and the PLVddd package (to reverse engineer DDL) will not return the desired results because of the way Oracle data dictionary views are defined. A number of the ALL_* views upon which the standard PL/Vision packages are defined restrict access to information about other accounts. The only way that PL/Vision can retrieve and display the information you will want from a shared perspective is by accessing the DBA_* data dictionary views.

So before you install PL/Vision, decide how you plan to use and share this product and grant privileges accordingly. Then take the following steps:

1. Choose or create an Oracle user account into which the PL/Vision packages and tables will be installed (hereafter referred to as the PL/Vision account). You can use a preexisting account, but you are probably better off installing all software in a "fresh" account. The recommended name for the PL/Vision account is PLVISION. You should *not* use PLV or any other name which is also a PL/Vision package name. Any of those names could cause compile errors and inconsistent behavior.

   To install and execute PL/Vision properly you will need to able to create tables, indexes, and synonyms, create packages, and read the DBA_* data dictionary views. You can grant these roles and privileges from SQL*Plus or you can use products like Oracle Navigator in Personal Oracle7.

   At a minimum, the PL/Vision account should be granted the CONNECT and RESOURCE roles. If you are going to share PL/Vision across accounts, you will also want to grant the SELECT ANY TABLE privilege. For example, if the SYSTEM account is a DBA account with a password of MANAGER, you would take these steps in SQL*Plus:

   ```
   SQL> connect system/manager
   SQL> grant select any table to PLVISION;
   ```

You also need to make sure that the PL/Vision account has EXECUTE authority on DBMS_LOCK. To accomplish this, connect to your SYS account and issue this command:

```
GRANT EXECUTE ON DBMS_LOCK TO PLVISION;
```

where PLVISION is the name of your PL/Vision account.

2. Set the default directory in SQL*Plus to the directory containing the PL/Vision software. Connect through SQL*Plus to the PLV account.

3. Run the PL/Vision package installation script. If you are running an Oracle Server Release 7.2 or below, run the *plvinst.sql* file as follows:

   ```
   SQL> @plvinst
   ```

   This script will create all packages and write the log of this activity to the *plvinst.log* file. If you are running Oracle Server Release 7.3 or above, run the *plvins23.sql* script as follows:

   ```
   SQL> @plvins23
   ```

   The installation script will create all packages and write the log of this activity to the **`plvins23.log`** file.

The version of PL/Vision Lite installed with **`plvins23.sql`** recognizes and makes use of the UTL_FILE package to perform operating system file I/O.

The remaining installation steps apply to installing PL/Vision for use by other users in your database.

4.3.6 Granting Access to PL/Vision

If you want to share PL/Vision among other users, run the **`plvgrant.sql`** script to grant execute authority on the PL/Vision packages. To grant execute authority to all users, execute the following command:

```
SQL> @plvgrant public
```

To grant execute authority to a particular user, execute the following command:

```
SQL> @plvgrant user
```

where *user* is the name of the user to whom you want access granted. Once you have granted access to PL/Vision packages from outside the PLV account, you will also probably want to create synonyms so you don't have to reference the PLV account by name when you call a PL/Vision program.

To create public synonyms for each of the PL/Vision packages, execute the *plvpsyn.sql* script from the PLV account as follows:

```
SQL> @plvpsyn connect_string
```

where *connect_string* is the connect string of a DBA account from which public synonyms can be created. The string may consist of the user name or user name/password combination. The *plvpsyn.sql* script will connect into that account and run a generated script of CREATE PUBLIC SYNONYM statements.

To create private synonyms for each of the PL/Vision packages, execute the *plvsyn.sql* script from the PLV account as follows:

```
SQL> @plvsyn connect_string
```

where *connect_string* is the connect string of the account for which private synonyms will be created. The string may consist of the user name or user name/password combination. The *plvsyn.sql* script will connect into that account and run a generated script of CREATE SYNONYM statements for that user.

PREVIOUS | HOME | NEXT
4.2 PL/Vision Package Bundles | BOOK INDEX | 4.4 Installing Online Help for PL/Vision

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

4.4 Installing Online Help for PL/Vision

PL/Vision takes advantage of the online help architecture of PLVhlp. If you choose to install the set of parallel help packages for PL/Vision, users will be able to request help during their SQL*Plus sessions on any of the PL/Vision packages.

To install the PL/Vision help packages, follow these steps:

1. Set the default directory in SQL*Plus to the directory containing the PL/Vision software. Connect through SQL*Plus to the PLV account.

2. Run the PL/Vision help installation as follows:

   ```
   SQL> @plvinsth
   ```

You will then have created a help package for each PL/Vision package. The naming convention for these packages is **`pkg_hlp`**, where *pkg* is the PL/Vision package name. For example, the help text for the PLVio package is contained in the **`PLVio_hlp`** package. The help packages consist only of help text in the form of PL/SQL comment blocks; these packages do not have bodies.

See Section 4.5, "Using Online Help" for more information about how to access the PL/Vision online help text.

4.4.1 Special Handling for PLVdyn

As I explain in Chapter 19, *PLVdyn and PLVfk: Dynamic SQL and PL/SQL*, you may want to install a copy of PLVdyn in the account of each developer who will be using this functionality. This step will avoid many potential headaches. If you are going to do this, you should take the following steps *after* installing PL/Vision and creating all synonyms:

1. Connect to the account in which you wish to install a private copy of PLVdyn. Let's call it the DEV account.

2. Drop the private synonym for PLVdyn in your user account with the following command.

   ```
   SQL> drop synonym plvdyn;
   ```

   If you have created public synonyms, you can skip this step.

3. Install the PLVdyn package in the DEV account. Run these two scripts:

```
SQL> @plvdyn.sps
SQL> @plvdyn.spb
```

PLVdyn is now installed in and owned by the DEV account. You can also follow these same steps to move the PLVdyn1 package to a local account.

4.4.2 A PL/Vision Initialization Script for SQL*Plus

If you plan to use PL/Vision from within SQL*Plus, you will want to consider creating a *login.sql* (or modifying your current file) that sets up your session from a PL/Vision perspective. PL/Vision provides two different scripts for your use: the *login.sql* and *login23.sql* files. Use the **`login.sql`** script if you're using PL/SQL Release 2.2 or earlier. Use the *login23.sql* script if you're using PL/SQL Release 2.3 and beyond.

Here are the contents of *login.sql*:

```
set feedback off
exec plvgen.set_author ('Sam I Am');
@ssoo
set feedback on
```

where *ssoo.sql* is a script that enables output from the DBMS_OUTPUT package. The call to PLV**`gen.set_author`** defines the default author used in the headers of generated PL/SQL program units. Feel free to change the string passed to **`set_author`**.

The contents of *login23.sql* are almost the same:

```
set feedback off
exec plvgen.set_author ('Sam I Am');
exec plvfile.set_dir ('c:/plv');
@ssoo
set feedback on
```

The PLV**`file.set_dir`** program sets the default directory for file I/O. You will probably want to change the string passed to **`set_dir`**, but it is very useful to set some value on startup of your SQL*Plus session.

4.4.3 Converting Scripts to PL/SQL Programs

Since SQL*Plus is a very common interactive execution platform for PL/SQL code, I make use of it throughout the book and in scripts provided on the disk. If you do not use SQL*Plus, it is generally very easy to convert SQL*Plus scripts to PL/SQL procedures. These procedures can then be executed in your environment to achieve the same effect as the scripts. Consider the following script (found in the file *dumpemp.sql* in the **`use`** subdirectory on the companion disk):

```
BEGIN
   PLVio.settrg (PLV.pstab);
   FOR emp_rec IN
       (SELECT ename FROM emp WHERE deptno = &1)
   LOOP
      PLVio.put_line (emp_rec.ename);
   END LOOP;
   PLVio.disptrg;
END;
/
```

This script uses PLVio to display all of the employee names in the specified department. The **`&1`** in the fourth line is a substitution parameter in SQL*Plus. To make this script work in PL/SQL, you can transform the *&1* into an actual PL/SQL parameter and wrap the script inside a procedure header as follows:

```
PROCEDURE dumpemp (dept_in IN emp.deptno%TYPE)
IS
BEGIN
   PLVio.settrg (PLV.pstab);
   FOR emp_rec IN
       (SELECT ename FROM emp WHERE deptno = dept_in)
   LOOP
      PLVio.put_line (emp_rec.ename);
   END LOOP;
   PLVio.disptrg;
END;
```

4.4.4 A Warning About Partial Installation

Do not attempt to install and use just a portion of PL/Vision. Why might you even consider such a thing? You might, for example, only want to use a single package, such as PLVvu to view your compile errors more effectively. You might balk at having to install all of those PL/Vision packages. However, almost every package in PL/Vision relies on many different PL/Vision packages in order to provide you with the highest possible level of functionality.

As a result, you must install all packages, following the directions provided in this chapter, whenever you want to use any portion of the product.

4.4.5 Uninstalling PL/Vision

To uninstall PL/Vision, you need to drop all synonyms, drop PL/Vision tables, and remove the packages. You can use PL/Vision to perform some of these steps.

To remove all the PL/Vision tables from an account, connect to that account in SQL*Plus and issue this command:

```
SQL> exec PLVdyn.drop_object ('table', 'PLV%');
```

All tables beginning with "PLV" will be dropped. This will take care of PL/Vision tables. You should, of course, ensure that there are no other tables that have this same naming pattern before you execute this command.

You can also execute the following script to remove all PL/Vision tables:

```
SQL> @plvtrem
```

To remove all the public synonyms for PL/Vision packages, connect to the PLV account in SQL*Plus (or a different DBA account if PLV is not a DBA account) and issue this command:

```
SQL> @plvdpsyn <connect_string>
```

The connect string is described in the section called "Installing PL/Vision."

To drop private synonyms for each of the PL/Vision packages, execute the *plvdsyn.sql* script from the account that has the private synonyms as follows:

```
SQL> @plvdsyn
```

You are now ready to remove the packages that make up PL/Vision. To do this, connect to the PLV account and issue the following command:

```
SQL> @plvprem
```

All packages and standalone programs will then be dropped.

PREVIOUS HOME NEXT
4.3 Installation Instructions BOOK INDEX 4.5 Using Online Help

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

4.5 Using Online Help

PL/Vision takes advantage of its own PLVhlp package to provide online help for its programs. As I've mentioned before, the help text that serves as source for this information resides in a set of parallel help packages. You can use several PL/SQL procedures or SQL*Plus scripts to access the help text.

To view the top−level help for a PL/Vision package, call the *help.sql* script (installed in the **use** subdirectory) as follows:

```
SQL> @help PKG (or SQL> exec PKG.help)
```

where PKG is the name of the package. So to see top−level help for PLVio, enter:

```
SQL> @help plvio (or SQL> exec plvio.help)
```

The case of your package name is not important. You will then see an overview of the package, a list of other topics available for this package and an index of programs defined in the package.

If the text exceeds the length of a PLVhlp page (the default is set to 25 lines), you will see this line after the end of the help text:

```
...more...
```

If there is more help text, you can then enter this command to see the next page:

```
SQL> @more (or SQL> exec plvhlp.more)
```

The format of the help index (also known as the help listing) is as follows:

```
N − program1_name         M − program2_name
```

where *N* is a number and **program_name** is the name of the program. You can use the number or the program name to get more information about that particular program or topic, as explained in the next section.

4.5.1 Zooming in on help text

You have two ways to zoom in on a particular topic within the current package's help text (the current package or program is defined by the parameter you pass in the call to *help.sql*): the *zoom.sql* and *zoomn.sql* scripts (both in the PL/Vision **use** subdirectory).

To zoom in on a topic by name you must first have executed the *help.sql* script (or the help procedure for the package of interest) to set the current program. Then you execute the zoom script, specifying the topic for which you want information:

```
SQL> @zoom put_line
```

If you do not want to have to type a program name or topic, you can also zoom in by a number from the help index. Again, you must first have executed the *help.sql* script to set the current program. Then you execute the zoom script, providing the topic number for which you want information. Suppose that my help index looks like this:

```
Listing for PACKAGE PLVio
1 - put_line        2 - get_line
3 - setsrc          4 - settrg
```

You can see information about the *setsrc* (set source) procedure by entering the following command at the SQL*Plus prompt:

```
SQL> @zoomn 3
```

> ***NOTE:*** Many of these scripts will execute only within the SQL*Plus environment. Others are more generic SQL or PL/SQL utilities and can be used in any environment that executes SQL statements and PL/SQL anonymous blocks.

PREVIOUS 4.4 Installing Online Help for PL/Vision | HOME BOOK INDEX | NEXT 4.6 Summary of Files on Disk

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8*i* Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

4.6 Summary of Files on Disk

The tables in this section list the scripts contained in subdirectories created by the installation procedure. These are in addition to the individual filenames of the PL/Vision packages listed earlier; note that all filenames are in dot notation, where the first component is the package name (e.g,. **`PLVvu.code`**).

4.6.1 Contents of the install Subdirectory

File	Description of Contents
`plv*.spb`	Bodies of the PL/Vision packages.
`plv*.sps`	Specifications of the PL/Vision packages.
`plvdata.sql`	Creates the data structures needed to use PL/Vision. Most of this script consists of a series of INSERT statements to populate the **`PLV_token`** and **`PLV_token_type`** tables.
`plvdpsyn.sql`	Drops all public synonyms for PL/Vision objects.
`plvdsyn.sql`	Drops all private synonyms for PL/Vision objects in a specified account.
`plvgrant.sql`	Grants access to all PL/Vision objects to the specified account or to PUBLIC.
`plvins23.sql`	Installs PL/Vision for all PL/SQL Releases 2.3 and above (the f indicates support for File I/O).
`plvinst.sql`	Installs PL/Vision for all PL/SQL Releases 2.2 and below.
`plvinsth.sql`	Installs PL/Vision online help text.
`plvprem.sql`	Removes all PL/Vision packages and code elements.
`plvpsyn.sql`	Creates public synonyms for all PL/Vision packages.
`plvsize.sql`	Displays the size of PL/Vision stored code by accessing the USER_OBJECT_SIZE data dictionary.
`plvsyn.sql`	Creates private synonyms for all PL/Vision packages for the specified account.
`plvtrem.sql`	Removes all PL/Vision tables.

4.6.2 Contents of the test Subdirectory

File	Description of Contents
`*.tst`	The disk contains a series of test scripts for many of the PL/Vision packages. They are generally named **`PKG.tst`** where PKG is the name of the package. Examples are **`PLVtrc.tst`** and **`PLVexc.tst`**. You can use these as a starting point for executing and trying out the PL/Vision packages.
`isnum.spp`	A package containing multiple implementations of a function that returns TRUE if the string is a number, FALSE otherwise.
`isnum.tst`	A test script to analyze the performance of the various functions in **`isnum.spp`**.

`lower.spp`	A package used to test the conversion of code to upper− and lowercase using the PLVcase package.
`mthtotal.sf`	A stored function which uses PLVfile to locate a specific line in a file and then return a value extracted from that line.
`PLVexc1.spp`	The first version of PLVexc the author developed to provide high−level exception−handling capabilities. It is interesting to compare this iteration with the final version to see how the capabilities of PLVexc grew increasingly abstract and declarative.
`showasci.sql`	Simple script to show the contents of the ASCII code table for the specified range of numbers.
`showhelp.all`	Code used to implement a prototype for an online help mechanism.
`spiral.all`	All of the different iterations of code that evolved in Chapter 3, *The PL/SQL Development Spiral*
`testpkg.sql`	Tests the overhead required to retrieve a value from a packaged global versus a local variable.
`timerep.sql`	A script that compares the performance of a string−repeating function for different implementations.
`upcexc.spp`	Example of an application−specific exception−handling package built over the more generic PLVexc package.

4.6.3 Contents of the use Subdirectory

File	Description of Contents
`code.sql`	Shortcut for executing **`PLVvu.code`** to see the source code for a stored object.
`creind.sql`	Creates a single index using dynamic SQL.
`dispfile.sql`	Displays the contents of a file using UTL_FILE and DBMS_OUTPUT (well, actually using the PL/Vision packages that in turn use those builtin packages).
`dumpemp.sql`	Demonstration of use of PLVio to dump the contents of the **`emp`** table to a PL/SQL table (the PLVio target repository).
`dumpprog.sql`	Dumps an program stored in the database out to an operating system file.
`dumpprog.sql`	Dumps the source code for a program into an operating system file using the PLVfile package.
`dynps.sql`	Demonstration of need to declare data structures as globals (in a package specification) if you are going to reference that data in a dynamically constructed PL/SQL block.
`dynvar.sql`	Demonstration of code required to dynamically create a package containing a global variable, including use of PLVtmr to calculate performance of this action.
`errm.sql`	Shortcut to execute **`PLV.errm`** to display the error message for an error code.
`execall.sql`	Grants execute authority on the specified program unit to PUBLIC and then creates a public synonym for that program.
`func.sql`	Shortcut to generate a function using the **`PLVgen.func`** program generator.

gendesc.sql	Demonstration of use of **PLVobj.loopexec** and dynamic PL/SQL to generate a script that outputs a DESC statement for each specified object.
gentkn.sql	Script that generates INSERT statements for the **PLV_token** and **PLV_token_type** tables.
haverr.sql	Shows all the modules that have entries in USER_ERRORS. A good quick way to see if an installation script completed successfully.
help.sql	Displays the top−level help for the specified program.
inctlg.sql	Shows the contents of the **PLVctlg** table for a particular program.
inexc.sql	Script to display contents of default PL/Vision exception log.
inline.sql	Displays all lines of code from the specified program that contains a particular string, relying mainly on PLVobj.
inline2.sql	Another version of inline that relies on PLVio to do the same job and requires much less code.
inlog.sql	Script to display the contents of the default PL/Vision log.
inrfrnc.sql	Displays the contents of the **PLVrfrnc** table for a particular program.
insrc.sql	Quick glance at the contents of **PLV_source**, the default repository for source code when writing to a database table.
intree.sql	Displays all elements which are dependent on the specified object, as is currently stored in the **PLVrfrnc** table.
listing.sql	Displays the element listing or index from the PLVhlp online help for a given package.
login.sql	Sample startup script for SQL*Plus (for PL/SQL Release 2.2 or earlier).
login23.sql	Sample startup script for SQL*Plus (for PL/SQL Release 2.3 or later).
modprs.sql	Shortcut script to generate and display the PL/SQL parser (via PLVprsps) of the specified program unit. The second argument of the script allows you to specify the type of identifiers to be retained in the parse. See the PLVprsps specification for the valid options for the second parameter.
more.sql	Shows the next page of online help using the PLVhlp facility.
nameres.sql	Provides a easy−to−call interface to the DBMS_UTILITY.NAME_RESOLVE builtin procedure.
nametoke.sql	Shortcut script to execute the DBMS_UTILITY.NAME_TOKENIZE builtin and see the results.
now.sql	Displays the current date and time.
plvcat.sql	Script to call **PLVcat.module** to catalogue all PL/Vision packages.
recomp.sql	Generates a SQL*Plus script to recompile and show errors for any program whose status is currently set to INVALID.

sepb.sql	Shortcut for SHOW ERRORS PACKAGE BODY, which displays the errors for the specified package body.
setcase.sql	Uses the PLVcase package to set or convert the case of one or more programs as specified. The script reads the source code from the data dictionary, writes it out to an operating system file, and then CREATE OR REPLACEs it back into the database.
sherr.sql	Shortcut for a call to the **PLVvu.err** procedure. Accepts a single argument, the program for which you want to display errors.
showerr.sp	An early version of the **PLVvu.err** procedure; this stored procedure displays error information.
showerr.sql showerr1.sp showerr2.sp	Different implementations of scripts that provide an alternative to the SHOW ERRORS command of SQL*Plus
showlog.sql	Shows the contents of the PL/SQL table−based PL/Vision log for the specified context entered on the current day.
showobj1.sql	Example of use of PLVobj cursor−related elements to query and display all the objects that match the specified string.
showobj2.sql	Another version of "show objects" that accomplishes the same task with a higher−level PLVobj element, **vu2pstab**.
showsrc.sql	Quick and simple display of the specified lines (start and end range) source code for a particular program. You can also use **PLVvu.code** to take advantage of the full range of features of that utility.
ssoo.sql	Executes Set ServerOutput On (hence, "ssoo") and sets the buffer size to one megabyte (the maximum).
vu.sql	A SQL*Plus shortcut script to execute **PLVvu.code** to view your stored code.
vuindex.sql	Allows you to view all indexes and their columns for the specified table.
zoom.sql	Displays PLVhlp online help text for a specified topic.
zoomn.sql	Displays PLVhlp online help text for a specified topic by number.

PREVIOUS HOME NEXT

4.5 Using Online Help BOOK INDEX 5. PL/Vision Package Specifications

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

5. PL/Vision Package Specifications

Contents:

This chapter pulls together all the specifications of the packages of PL/Vision. Organized in alphabetical order by package name, these specifications give you the basic information you need to know to call the PL/Vision programs properly in your own applications.

5.1 Common Package Elements

In providing online help, all of the PL/Vision packages take advantage of the PLVhlp utility. This is accomplished by including a help procedure in each package; the header of this procedure is as follows:

```
        PROCEDURE help (topic_in IN VARCHAR2 := NULL)
```

You can therefore obtain "top−level" help for any documented package in PL/Vision via the following command (where *PLVpkg* is the name of the package):

```
        SQL> exec PLVpkg.help
```

PREVIOUS
4.6 Summary of Files on Disk

HOME
BOOK INDEX

NEXT
5.2 p: a DBMS_OUTPUT Substitute

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

5.2 p: a DBMS_OUTPUT Substitute

The **p** (Put) package provides a powerful, flexible substitute (and one that requires minimal typing) for Oracle's builtin DBMS_OUTPUT.PUT_LINE package. See Chapter 7, *p: A Powerful Substitute for DBMS_OUTPUT* for details.

5.2.1 Toggling output from the p package

PROCEDURE turn_on;

Turn on output from the **p.l** procedure (the default). Equivalent to calling DBMS_OUTPUT.ENABLE to enable output from DBMS_OUTPUT.

PROCEDURE turn_off;

Turn off output from the **p.l** procedure. This setting can be overridden in a particular call to **p.l** by specifying TRUE for the final **show** argument.

5.2.2 Setting the line separator

PROCEDURE set_linesep (linesep_in IN VARCHAR2);

Sets the character(s) to be recognized as the line separator in your PL/SQL code (default is **=**). A line that consists only of the line separator will be displayed by **p.l** as a blank line.

FUNCTION linesep RETURN VARCHAR2;

Returns the line separator character(s).

5.2.3 Setting the line prefix

PROCEDURE set_prefix (prefix_in IN VARCHAR2 := c_prefix);

Sets the character(s) used as a prefix to the text displayed by the **p.l** procedure.

The **c_prefix** constant is defined in the p package as CHR(8).

FUNCTION prefix RETURN VARCHAR2;>

Returns the prefix character(s).

5.2.4 The overloadings of the l procedure

```
PROCEDURE l
   (date_in IN DATE, mask_in IN VARCHAR2 := PLV.datemask,
    show_in IN BOOLEAN := FALSE);

PROCEDURE l (number_in IN NUMBER, show_in IN BOOLEAN := FALSE);

PROCEDURE l (char_in IN VARCHAR2, show_in IN BOOLEAN := FALSE);
PROCEDURE l
   (char_in IN VARCHAR2, number_in IN NUMBER, show_in IN BOOLEAN := FALSE);

PROCEDURE l
   (char_in IN VARCHAR2, date_in IN DATE,
    mask_in IN VARCHAR2 := PLV.datemask, show_in IN BOOLEAN := FALSE);

PROCEDURE l (boolean_in IN BOOLEAN, show_in IN BOOLEAN := FALSE);

PROCEDURE l
   (char_in IN VARCHAR2, boolean_in IN BOOLEAN,
    show_in IN BOOLEAN := FALSE);

PROCEDURE l
   (file_in IN UTL_FILE.FILE_TYPE, show_in IN BOOLEAN := FALSE);
```

The version of **`l`** that displays the contents of a UTL_FILE file handle can only be used in PL/SQL Releases 2.3 and beyond (and is only installed when you run the **`plvinstf.sql`** script).

If you pass TRUE for the **`show_in`** argument, you will always see output from that call to **`p.l`** −− even if you have called **`p.turn_off`** earlier in the session (but only if you have enabled output from DBMS_OUTPUT).

PREVIOUS 5.1 Common Package Elements | HOME BOOK INDEX | NEXT 5.3 PLV: Top−Level Constants and Functions

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

PREVIOUS
Chapter 5
PL/Vision Package
Specifications
NEXT

5.3 PLV: Top−Level Constants and Functions

The PLV (PL/Vision) package provides a single collection point for constants and basic functions used throughout the PL/Vision library of packages. See Chapter 6, *PLV: Top−Level Constants and Functions* for details.

5.3.1 PL/Vision constants

```
dbtab CONSTANT VARCHAR2(2) := 'DB';
pstab CONSTANT VARCHAR2(2) := 'PS';
file CONSTANT VARCHAR2(2) := 'F';
string CONSTANT VARCHAR2(2) := 'S';
stdout CONSTANT VARCHAR2(2) := 'SO';
```

Names of different repositories supported within PL/Vision. These are mostly used by PLVio and by users of PLVio to set up the source and target repositories for PL/SQL code.

```
c_datemask CONSTANT VARCHAR2(100) :=
'FMMonth DD, YYYY FMHH24:MI:SS'
```

The default date format mask for PL/Vision.

5.3.2 Anchoring datatypes

```
plsql_identifier VARCHAR2(100) := 'IRRELEVANT';
max_varchar2 VARCHAR2(32767) := 'IRRELEVANT';
vcmax_len CONSTANT INTEGER := 32767;
```

The **plsql_identifier** variable offers a standard format for the declaration of any variables that will hold PL/SQL identifiers, such as table and column names.

The **max_varchar2** variable offers a standard format for the declaration of any variables that require the maximum possible size for VARCHAR2 variables, which is 32,767 bytes and also reflected by the value of the **vcmax_len** constant.

5.3.3 Setting the date format mask

```
PROCEDURE set_datemask (datemask_in IN VARCHAR2 := c_datemask);
```

Sets the string used as the default date format mask within PL/Vision.

```
FUNCTION datemask RETURN VARCHAR2;
```

Returns the string used as the default date format mask within PL/Vision.

5.3.4 Setting the NULL substitution value

```
PROCEDURE set_nullval (nullval_in IN VARCHAR2);
```

Sets the string used as the substitution value for NULLs within PL/Vision.

```
FUNCTION nullval RETURN VARCHAR2;
```

Returns the current NULL substitution value.

5.3.5 Assertion routines

```
assertion_failure EXCEPTION;
```

Exception raised by the various assertion routines when the assertion fails.

```
PROCEDURE assert (bool_in IN BOOLEAN, stg_in IN VARCHAR2 := NULL);
```

Does nothing if the Boolean argument evaluates to TRUE. Otherwise (for FALSE *or* NULL), it raises the **`assertion_failure`** exception and displays the message.

This same behavior holds for the other assertion routines shown below.

```
PROCEDURE assert_notnull
(val_in IN VARCHAR2, stg_in IN VARCHAR2 := NULL);
PROCEDURE assert_notnull
(val_in IN DATE, stg_in IN VARCHAR2 := NULL);
PROCEDURE assert_notnull
(val_in IN NUMBER, stg_in IN VARCHAR2 := NULL);
PROCEDURE assert_notnull
(val_in IN BOOLEAN, stg_in IN VARCHAR2 := NULL);
```

Overloadings for the NOT NULL assertion logic.

```
PROCEDURE assert_inrange
(val_in IN DATE,
start_in IN DATE := SYSDATE, end_in IN DATE := SYSDATE+1,
stg_in IN VARCHAR2 := NULL, truncate_in IN BOOLEAN := TRUE);
PROCEDURE assert_inrange
(val_in IN NUMBER, start_in IN NUMBER, end_in IN NUMBER,
stg_in IN VARCHAR2 := NULL);
```

Overloadings of "in range" assertions for both date and numeric information.

5.3.6 Miscellaneous programs

```
FUNCTION boolstg (bool_in IN BOOLEAN, stg_in IN VARCHAR2 := NULL)
RETURN VARCHAR2;
```

Returns a string representing the value of the Boolean argument: TRUE if the Boolean argument is TRUE, FALSE if FALSE, and NULL if NULL.

```
FUNCTION errm (code_in IN INTEGER := SQLCODE) RETURN VARCHAR2;
```

Returns the error message provided by SQLERRM. Encapsulation inside this function allows SQLERRM to be referenced inside a SQL statement.

```
FUNCTION now RETURN VARCHAR2;
```

Returns the current date and time using the current PL/Vision date format mask.

```
PROCEDURE pause (secs_in IN INTEGER);
```

Pauses your PL/SQL program for the specified number of seconds.

PREVIOUS | HOME | NEXT
5.2 p: a DBMS_OUTPUT Substitute | BOOK INDEX | 5.4 PLVcase: PL/SQL Code Conversion

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8*i* Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

5.4 PLVcase: PL/SQL Code Conversion

The PLVcase (PL/Vision CASE) package converts the case of PL/SQL source code so that it conforms to the UPPER−lower method (reserved words in upper−case, application−specific identifiers in lower−case). See Chapter 18, *PLVcase and PLVcat: Converting and Analyzing PL/SQL Code* for details.

5.4.1 Package constants

```
c_usecor CONSTANT VARCHAR2(3) := 'COR';
```

The constant used in calls to module (see below) to indicate that a CREATE OR REPLACE should be appended to the source code for the program unit.

```
c_nousecor CONSTANT VARCHAR2(4) := 'NCOR';
```

The constant used to tell module to *not* append the CREATE OR REPLACE to the program unit.

5.4.2 Case−converting programs

```
FUNCTION token (token_in IN VARCHAR2, pkg_in IN VARCHAR2 := NULL)
RETURN VARCHAR2;
```

Converts the case of a single token according to the UPPER−lower method.

```
PROCEDURE line
(line_in IN OUT PLVio.line_type,
line_out IN OUT PLVio.line_type,
found_out OUT BOOLEAN);
```

Converts the case of a single line of source code according to the UPPER−lower method (it calls **`PLVcase.token`** for each token in the string).

```
FUNCTION string (string_in IN VARCHAR2) RETURN VARCHAR2;
```

Converts the case of a string according to the UPPER−lower method. It formats the string as necessary for a call to the **`PLVcase.line`** procedure.

```
PROCEDURE string (string_inout IN OUT VARCHAR2);
```

Procedure version of the **`string`** function. It hands you back your own **`string`** variable with the case of the tokens converted.

```
PROCEDURE module
(module_in IN VARCHAR2,
cor_in IN VARCHAR2 := c_usecor,
last_module_in IN BOOLEAN := TRUE);
```

Converts the case of a single program unit according to the UPPER−lower method.

```
PROCEDURE modules (module_spec_in IN VARCHAR2 := NULL);
```

Converts the case of multiple program units according to the UPPER−lower method.

PREVIOUS | HOME | NEXT
5.3 PLV: Top−Level Constants and Functions | BOOK INDEX | 5.5 PLVcat: PL/SQL Code Cataloguing

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

5.5 PLVcat: PL/SQL Code Cataloguing

The PLVcat (PL/Vision CATalogue) package catalogues PL/SQL source code so that you can analyze the contents of your program for cross−references, dependencies, and so on. See Chapter 18 for details.

5.5.1 Cataloguing package contents

PROCEDURE module (module_in IN VARCHAR2);

Scans the contents of the specified module (currently only package specifications are supported) and writes the list of its contents to the **PLVrfrnc** table.

PROCEDURE modules (module_in IN VARCHAR2);

Performs same task as the **module** procedure, but for multiple program units. You can, in other words, provide an argument for **module_in** that contains wildcards.

5.5.2 Identifying references in stored code

PROCEDURE refnonkw (module_in IN VARCHAR2);

Scans the contents of the specified program unit and writes to the **PLVctlg** table all references to non−keyword identifiers.

PROCEDURE refbi (module_in IN VARCHAR2);

Generates the list of builtin functions and packages that are referenced within the specified program unit. This list is then written to the **PLVctlg** table.

PREVIOUS HOME NEXT
5.4 PLVcase: PL/SQL Code Conversion BOOK INDEX 5.6 PLVchr: Operations on Single Characters

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8*i* Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

5.6 PLVchr: Operations on Single Characters

The PLVchr (PL/Vision CHaRacter) package provides information about single characters in a string. See the companion disk for details.

5.6.1 PLVchr constants

```
blank_char CONSTANT CHAR(1) := ' ';
space_char CONSTANT CHAR(1) := ' ';
quote1_char CONSTANT CHAR(1) := '''';
quote2_char CONSTANT CHAR(2) := '''''';
tab_char CONSTANT CHAR(1) := CHR(9);
newline_char CONSTANT CHAR(1) := CHR(10);
```

Named constants to use in place of hard−coded literals. Sure, there is more typing involved. But at least you don't have to mess with single quotes, and the code is a lot more readable.

```
c_nonprinting CONSTANT CHAR(1) := 'n';
c_digit CONSTANT CHAR(1) := '9';
c_letter CONSTANT CHAR(1) := 'a';
c_other CONSTANT CHAR(1) := '*';
c_all CONSTANT CHAR(1) := '%';
```

Action codes for use in PLVchr programs that allow you to specify the category of character that you want to work with.

5.6.2 Character type functions

```
FUNCTION is_quote (char_in IN VARCHAR2) RETURN BOOLEAN;
FUNCTION is_blank (char_in IN VARCHAR2) RETURN BOOLEAN;
```

Functions to encapsulate hard−coded checks for contents of the strings.

```
FUNCTION is_nonprinting (code_in IN INTEGER) RETURN BOOLEAN;
FUNCTION is_nonprinting (letter_in IN VARCHAR2) RETURN BOOLEAN;
```

Returns TRUE if the character (or, overloaded as this function is, the ASCII code) is a non−printing character.

```
FUNCTION is_digit (code_in IN INTEGER) RETURN BOOLEAN;
FUNCTION is_digit (letter_in IN VARCHAR2) RETURN BOOLEAN;
```

Returns TRUE if the character (or, overloaded as this function is, the ASCII code) is a digit (0 through 9).

```
FUNCTION is_letter (letter_in IN VARCHAR2) RETURN BOOLEAN;
FUNCTION is_letter (code_in IN INTEGER) RETURN BOOLEAN;
```

Returns TRUE if the character (or, overloaded as this function is, the ASCII code) is a letter (a−z or A−Z).

```
FUNCTION is_other (code_in IN INTEGER) RETURN BOOLEAN;
FUNCTION is_other (letter_in IN VARCHAR2) RETURN BOOLEAN;
```

Returns TRUE if the character (or ASCII code) is not a letter, digit, or nonprinting character.

5.6.3 Other functions and procedures

```
FUNCTION char_name (letter_in IN VARCHAR2) RETURN VARCHAR2;
FUNCTION char_name (code_in IN INTEGER) RETURN VARCHAR2;
```

Returns the name of the provided character. This name is actually a standard abbreviation for the character, such as **`NL`** for new line. The name of a printable character is simply the character itself. You can pass either a character or an integer code to see the name.

```
FUNCTION quoted1 (string_in IN VARCHAR2) RETURN VARCHAR2;
FUNCTION quoted2 (string_in IN VARCHAR2) RETURN VARCHAR2;
```

Each function returns a string wrapped inside the number of single quote marks needed to allow the string to be evaluated to a string surrounded by one and two single quote marks, respectively.

```
FUNCTION stripped (string_in IN VARCHAR2, char_in IN VARCHAR2)
RETURN VARCHAR2;
```

Strips a string of all instances of the specified characters. This function is a frontend to TRANSLATE.

```
PROCEDURE show_string
(string_in IN VARCHAR2, flags_in IN VARCHAR2 := c_all);
```

Displays the ASCII code and its associated character for each character in the specified string. You can request to view only certain kinds of characters.

```
PROCEDURE show_table
(start_code_in IN INTEGER := 1,
end_code_in IN INTEGER := NULL);
```

Displays the ASCII code and its associated character for all codes within the specified start−end range.

PREVIOUS | HOME | NEXT
5.5 PLVcat: PL/SQL Code Cataloguing | BOOK INDEX | 5.7 PLVcmt: Commit Processing

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

5.7 PLVcmt: Commit Processing

The PLVcmt (PL/Vision CoMmiT) package provides a programmatic interface to the execution of commits, rollbacks, and the setting of savepoints. See Chapter 20, *PLVcmt and PLVrb: Commit and Rollback Processing* for details.

5.7.1 Controlling commit activity

PROCEDURE turn_on;
: Enables commit processing in PLVcmt. This is the default.

PROCEDURE turn_off;
: Disables commit processing in PLVcmt. When this program is called in the current session, the COMMIT statement will not be executed.

FUNCTION committing RETURN BOOLEAN;
: Returns TRUE if commit processing is being performed by PLVcmt.

5.7.2 Logging commit activity

PROCEDURE log;
: Requests that, whenever a COMMIT is performed, a message be sent to the PL/Vision log.

PROCEDURE nolog;
: Do not log a message with the COMMIT.

FUNCTION logging RETURN BOOLEAN;
: Returns TRUE if currently logging the fact that a commit was performed by PLVcmt.

5.7.3 Performing commits

PROCEDURE increment_and_commit (context_in IN VARCHAR2 := NULL);
: Increments the counter and commits if a commit point has been reached.

PROCEDURE perform_commit(context_in IN VARCHAR := NULL);
: The PLVcmt package's version of COMMIT. I could probably get away with calling this program **commit**, but I avoid using keywords even when the compiler doesn't seem to get confused.

5.7.4 Managing the commit counter

PROCEDURE commit_after (count_in IN INTEGER);
: Sets the break point at which a commit is performed. In other words, when the package−based counter reaches the specified number, issue a COMMIT. The default is to commit after the counter reaches 1.

PROCEDURE init_counter;

Initializes the PLVcmt counter referenced by the **increment_and_commit** program to perform incremental commits.

PREVIOUS | HOME | NEXT
5.6 PLVchr: Operations on Single Characters | BOOK INDEX | 5.8 PLVddd: DDL Syntax Dump

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

5.8 PLVddd: DDL Syntax Dump

The PLVddd (PL/Vision Dump Data Definition language) package dumps DDL syntax from a particular schema to allow you to recreate database objects easily in other schemas. See the companion disk for details.

5.8.1 Including the schema

```
PROCEDURE inclschema;
```

Turns on the inclusion of schema names before each created object. This is the DEFAULT position.

```
PROCEDURE noinclschema;
```

Turns off the showing of the schema names before each created object.

```
FUNCTION including_schema RETURN BOOLEAN;
```

Returns TRUE if including the schema.

5.8.2 Including the storage parameter

```
PROCEDURE inclsp;
```

Turns on the inclusion of the storage parameters after appropriate objects.

```
PROCEDURE noinclsp;
```

Turns OFF the inclusion of storage parameters after appropriate objects. This is the DEFAULT position.

```
FUNCTION including_sp RETURN BOOLEAN;
```

Function returning TRUE if storage parameters are included and FALSE if they are not.

5.8.3 Dumping the DDL

```
PROCEDURE tbl (owner_in IN VARCHAR2, table_in IN VARCHAR2 := '%');
```

Dumps DDL for tables including named column, check constraints, and storage information.

```
PROCEDURE idx
(owner_in IN VARCHAR2,
name_in IN VARCHAR2 := '%',
table_in IN VARCHAR2 := '%');
```

Dumps DDL for single indexes or all indexes on tables.

```
PROCEDURE pky
(owner_in IN VARCHAR2,
name_in IN VARCHAR2 := '%',
table_in IN VARCHAR2 := '%');
```

Dumps DDL for single primary keys or all primary keys on tables.

```
PROCEDURE fky
(owner_in IN VARCHAR2,
name_in IN VARCHAR2 := '%',
table_in IN VARCHAR2 := '%');
```

Dumps DDL for single foreign keys or all foreign keys on tables.

```
PROCEDURE syn
(synonym_owner_in IN VARCHAR2,
name_in IN VARCHAR2 := '%',
object_owner_in IN VARCHAR2 := '%',
object_in IN VARCHAR2 := '%');
```

Dumps DDL for single synonyms or all synonyms for a table.

```
PROCEDURE vw (owner_in IN VARCHAR2, name_in IN VARCHAR2 := '%');
```

Dumps DDL for views.

```
PROCEDURE trig
(owner_in IN VARCHAR2,
name_in IN VARCHAR2 := '%',
table_in IN VARCHAR2 := '%');
```

Dumps DDL for single triggers or all triggers on tables.

```
PROCEDURE plsql (owner_in IN VARCHAR2, name_in IN
VARCHAR2 := '%');
```

Dumps DDL for PL/SQL code objects including functions, packages, package bodies, and procedures.

```
PROCEDURE seq (owner_in IN VARCHAR2, name_in IN VARCHAR2 := '%');
```

Dumps DDL for sequences including starting points, max/min values, etc.

```
PROCEDURE schema (owner_in IN VARCHAR2, object_in IN
VARCHAR2 := '%');
```

Dumps all of the DDL related to a specified object. If the object is a table, for example, it can generate all indexes, keys, triggers, synonyms, and views for that table as well.

PREVIOUS | HOME | NEXT
5.7 PLVcmt: Commit Processing | BOOK INDEX | 5.9 PLVdyn: Dynamic SQL Operations

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

5.9 PLVdyn: Dynamic SQL Operations

The PLVdyn (PL/Vision DYNamic SQL) package provides a high−level interface to Oracle's builtin DBMS_SQL package. See Chapter 19, *PLVdyn and PLVfk: Dynamic SQL and PL/SQL* for details.

5.9.1 Tracing PLVdyn activity

`PROCEDURE showsql (start_with_in IN VARCHAR2 := NULL);`

Requests that the string being parsed dynamically be displayed. You can specify the string that should start the displayed text.

`PROCEDURE noshowsql;`

Turns off the display of the dynamic SQL string.

`FUNCTION showing RETURN BOOLEAN;`

Returns TRUE if currently showing the dynamic SQL.

5.9.2 Controlling execution of dynamic SQL

`PROCEDURE execsql;`

Requests that calls to **`PLVdyn.execute`** call the underlying DBMS_SQL.EXECUTE builtin.

`PROCEDURE noexecsql;`

Requests that **`PLVdyn.execute`** *not* actually execute the specified cursor.

`FUNCTION executing RETURN BOOLEAN;`

Returns TRUE if currently executing the dynamic SQL.

5.9.3 Bundled, low−level operations

```
FUNCTION open_and_parse
(string_in IN VARCHAR2,
mode_in IN INTEGER := DBMS_SQL.NATIVE) RETURN INTEGER;
```

Combines the open and parse operations into a single function call.

`PROCEDURE execute (cur_inout IN INTEGER);`

A passthrough to the DBMS_SQL.EXECUTE function. By using **`PLVdyn.execute`**, you give yourself the flexibility to turn off execution without modifying your code.

```
PROCEDURE execute_and_fetch
(cur_inout IN INTEGER, match_in IN BOOLEAN := FALSE);
```

A passthrough to the DBMS_SQL.EXECUTE_AND_FETCH function. By using this procedure, you give yourself the flexibility to turn off execution without modifying your code.

```
PROCEDURE execute_and_close (cur_inout IN OUT INTEGER);
```

Combines the execute and close operations into a single call.

```
PROCEDURE parse_delete
(table_in IN VARCHAR2, where_in IN VARCHAR2,
cur_out OUT INTEGER);
```

Performs the parse step of DBMS_SQL for a DELETE string constructed from the arguments in the parameter list.

5.9.4 Data Definition Language operations

```
PROCEDURE ddl (string_in IN VARCHAR2);
```

Executes any DDL statement by performing an OPEN, then a PARSE. This program forces a commit in your session, as when any DDL command is given.

```
PROCEDURE drop_object
(type_in IN VARCHAR2, name_in IN VARCHAR2,
schema_in IN VARCHAR2 := USER);
```

Provides a generic, powerful interface to the DDL DROP command. You can drop individual or multiple objects.

```
PROCEDURE truncate
(type_in IN VARCHAR2, name_in IN VARCHAR2,
schema_in IN VARCHAR2 := USER);
```

Truncates either a table or a cluster as specified.

```
PROCEDURE compile
(stg_in IN VARCHAR2, show_err_in IN VARCHAR2 := PLV.noshow);
```

Executes a CREATE OR REPLACE of the program contained in the first argument, **`stg_in`**. You can also request that errors from this compile be immediately displayed with a call to the **`PLVvu.err`** procedure.

```
PROCEDURE compile
(table_in IN PLVtab.vc2000_table,
lines_in IN INTEGER,
show_err_in IN VARCHAR2 := PLV.noshow);
```

Another version of dynamic CREATE OR REPLACE that reads the source code for the program from the PL/SQL table.

```
FUNCTION nextseq (seq_in IN VARCHAR2, increment_in IN INTEGER := 1)
RETURN INTEGER;
```

Returns the next value from the specified sequence. Can retrieve the immediate next value or the *n*th next value. Use of this function avoids direct reference to the DUAL table.

5.9.5 Data Manipulation Language operations

```
PROCEDURE dml_insert_select
(table_in IN VARCHAR2, select_in IN VARCHAR2);
```

Issues an INSERT−SELECT statement based on the arguments provided.

```
PROCEDURE dml_delete
(table_in IN VARCHAR2, where_in IN VARCHAR2 := NULL);
```

Deletes all rows specified by the WHERE clause from the table argument.

```
PROCEDURE dml_update
(table_in IN VARCHAR2,
column_in IN VARCHAR2,
value_in IN VARCHAR2|NUMBER|DATE,
where_in IN VARCHAR2 := NULL);
```

Overloaded to support string, numeric, and date values, **`dml_update`** performs a single−column UPDATE as specified by the arguments.

5.9.6 Executing dynamic PL/SQL

```
PROCEDURE plsql (string_in IN VARCHAR2);
```

Executes any PL/SQL code. This procedure automatically packages your string inside a BEGIN−END block and terminates it with a semicolon.

5.9.7 Miscellaneous programs

```
PROCEDURE disptab
(table_in IN VARCHAR2,
where_in IN VARCHAR2 := NULL,
string_length_in IN INTEGER := 20,
date_format_in IN VARCHAR2 := PLV.datemask,
num_length_in IN INTEGER := 10);
```

Displays the requested contents of any database table. Good example of the kind of code required to perform Method 4 dynamic SQL.

```
FUNCTION plsql_block (string_in IN VARCHAR2) RETURN VARCHAR2;
```

Returns a string that is a valid PL/SQL block for dynamic PL/SQL execution.

```
FUNCTION placeholder
(string_in IN VARCHAR2, start_in IN INTEGER := 1)
RETURN VARCHAR2;
```

Locates and returns the *n*th placeholder for bind variables in strings.

```
FUNCTION tabexists (table_in IN VARCHAR2) RETURN BOOLEAN;
```

Returns TRUE if the specified table exists.

```
PROCEDURE time_plsql
(stg_in IN VARCHAR2, repetitions_in IN INTEGER := 1);
```

Calculates the overhead required to execute a dynamically constructed anonymous PL/SQL block.

PREVIOUS | HOME | NEXT
5.8 PLVddd: DDL Syntax Dump | BOOK INDEX | 5.10 PLVexc: Exception Handling

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

5.10 PLVexc: Exception Handling

The PLVexc (PL/Vision EXCeption handling) package provides generic and powerful exception−handling capabilities. See Chapter 22, *Exception Handling* for details.

5.10.1 Package constants

```
c_go CONSTANT CHAR(1) := 'C';
```

Requests that your program continue (ignore the error). Explained in more detail below.

```
c_recNgo CONSTANT CHAR(2) := 'RC';
```

Requests that your program record the error and then continue. Explained in more detail below.

```
c_stop CONSTANT CHAR(1) := 'H';
```

Requests that your program be halted if this exception occurs. Explained in more detail below.

```
c_recNstop CONSTANT CHAR(2) := 'RH';
```

Requests that your program record the error and then halt. Explained in more detail below.

5.10.2 Package−based exceptions

```
process_halted EXCEPTION;
```

Package−specific exception raised when you request a "halt" action in the handler programs.

```
no_such_table EXCEPTION;
PRAGMA EXCEPTION_INIT (no_such_table, −942);
```

Predefined system exception for error ORA−942. Saves other developers from dealing with the EXCEPTION_INIT pragma.

```
snapshot_too_old EXCEPTION;
PRAGMA EXCEPTION_INIT (snapshot_too_old, −1555);
```

Predefined system exception for error ORA−1555. Saves other developers from dealing with the EXCEPTION_INIT pragma.

5.10.3 Logging exception−handling activity

```
PROCEDURE log;
```

Requests that whenever a PLVexc handler is called, a message is sent to the PL/Vision log.

```
PROCEDURE nolog;
```

Do not log the handling action when the exception is recorded and handled. with the COMMIT.

```
FUNCTION logging RETURN BOOLEAN;
```

Returns TRUE if currently logging PLVexc−based exception handling.

5.10.4 Displaying exceptions

PROCEDURE show;

Requests that error information be displayed to your screen using the **p.l** procedure.

PROCEDURE noshow;

Turns off display of the error information.

FUNCTION showing RETURN BOOLEAN;

Returns TRUE if PLVexc is currently showing errors.

5.10.5 Rolling back on exception

PROCEDURE rblast;

Requests that a rollback be issued to the most recent savepoint before writing error information to the log (the default).

PROCEDURE rbdef;

Requests that a rollback be issued to the default PLVlog savepoint before writing error information to the log (the default).

PROCEDURE norb;

Turns off issuing of rollback before logging of the error information.

FUNCTION rb RETURN VARCHAR2;

Returns TRUE if PLVexc is currently issuing a rollback.

5.10.6 Exception handlers

```
PROCEDURE handle
(context_in IN VARCHAR2,
err_code_in IN INTEGER,
handle_action_in IN VARCHAR2,
msg_in IN VARCHAR2 := SQLERRM);
```

Low−level, generic exception−handling program. This program is called by all other PLVexc handlers, which are overloaded for error number and message.

```
PROCEDURE recNgo (msg_in IN VARCHAR2 := NULL);
PROCEDURE recNgo (err_code_in IN INTEGER);
```

High−level exception handler that records and then ignores the error.

```
PROCEDURE go (msg_in IN VARCHAR2 := NULL);
PROCEDURE go (err_code_in IN INTEGER);
```

High−level exception handler that ignores the error, but gives you the opportunity to log or display the exception.

```
PROCEDURE recNstop (msg_in IN VARCHAR2 := NULL);
PROCEDURE recNstop (err_code_in IN INTEGER);
```

High−level exception handler that records the error and then causes the current program to halt.

```
PROCEDURE stop (msg_in IN VARCHAR2 := NULL);
PROCEDURE stop (err_code_in IN INTEGER);
```

High−level exception handler that causes the current program to halt.

5.10.7 Bailing out program execution

`PROCEDURE bailout;`

Starts the bailout process; the current exception will be propagated out of all exception sections that use PLVexc, regardless of the action handled.

`PROCEDURE nobailout;`

Turns off the bailout process. PLVexc will not propagate the exception past all PLVexc exception handlers.

`FUNCTION bailing_out RETURN BOOLEAN;`

Returns TRUE if PLVexc is currently set to bail out when it encounters a bailout error.

`PROCEDURE clear_bailouts;`

Registers a specific error number as a bailout error.

5.10.8 Managing the list of bailout errors

`PROCEDURE clear_bailouts;`

Clears the PLVexc list of bailout errors.

`PROCEDURE bailout_on (err_code_in IN INTEGER);`

Adds an error code to the list that PLVexc treats as bailout errors.

`PROCEDURE nobailout_on (err_code_in IN INTEGER);`

Removes an error code from the list that PLVexc treats as bailout errors.

`FUNCTION bailout_error (err_code_in IN INTEGER) RETURN BOOLEAN;`

Returns TRUE if the specified error is a bailout error.

PREVIOUS | HOME | NEXT
5.9 PLVdyn: Dynamic SQL Operations | BOOK INDEX | 5.11 PLVfile: Operating System I/O Manager

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

5.11 PLVfile: Operating System I/O Manager

The PLVfile (PL/Vision FILE) package manages operating system I/O by providing a layer of code around Oracle's builtin UTL_FILE package. See Chapter 13, *PLVfile: Reading and Writing Operating System Files* for details.

5.11.1 Package constants and exceptions

```
max_line_size CONSTANT INTEGER := 1000;
```

The maximum size of a line allowed to be read or written with PLVfile.

```
max_line VARCHAR2(1000);
```

I had to "hard code" the 1000 again in this declaration because you must supply a *literal* when you declare a length for a VARCHAR2 string. Predefined variable you can use to anchor declarations of local variables in your own programs that will hold the maximum length line allowed in PLVfile.

```
c_append CONSTANT VARCHAR2(1) := 'A';
c_read CONSTANT VARCHAR2(1) := 'R';
c_write CONSTANT VARCHAR2(1) := 'W';
```

The different types of file access allowed with the UTL_FILE builtin package (**A** = append to existing lines in file, **R** = read−only from file and **W** = write over existing contents of file).

```
c_unixdelim CONSTANT VARCHAR2(1) := '/';
c_dosdelim CONSTANT VARCHAR2(1) := '\';
```

Predefined operating system directory/path delimiters for UNIX and MS−DOS.

5.11.2 Trace PLVfile activity

```
PROCEDURE show;
```

Turns on the trace of PLVfile activity.

```
PROCEDURE noshow;
```

Turns off the trace of PLVfile activity (default setting).

```
FUNCTION showing RETURN BOOLEAN;
```

Returns TRUE if you are currently tracing PLVfile activity.

5.11.3 Setting the operating system delimiter

```
PROCEDURE set_delim (delim_in IN VARCHAR2);
```

Sets the string to be used as the operating system delimiter (the character that goes between the file location and the file name).

```
FUNCTION delim RETURN VARCHAR2;
```

Returns the current operating system delimiter.

5.11.4 Setting the default directory or location

PROCEDURE set_dir (dir_in IN VARCHAR2);

Sets the default directory for the file you are managing with PLVfile. If you specify the directory with **set_dir**, you will not have to provide it in each call to PLVfile programs.

FUNCTION dir RETURN VARCHAR2;

Returns the current default directory.

5.11.5 Creating files

FUNCTION fcreate
(loc_in IN VARCHAR2, file_in IN VARCHAR2, line_in IN VARCHAR2)
RETURN UTL_FILE.FILE_TYPE;

Specify file location and name separately, as well as the single line of text to place in the file. The **fcreate** procedure will create the file and return the handle to the file.

FUNCTION fcreate
(file_in IN VARCHAR2, line_in IN VARCHAR2 := NULL)
RETURN UTL_FILE.FILE_TYPE;

Create the file without explicitly providing the file location.

PROCEDURE fcreate
(loc_in IN VARCHAR2, file_in IN VARCHAR2, line_in IN VARCHAR2);

Create the file but do not return the handle to the file.

PROCEDURE fcreate
(file_in IN VARCHAR2, line_in IN VARCHAR2 := NULL);

Create the file without explicitly providing the file location and do not return the handle to the file.

5.11.6 Checking for file existence

FUNCTION fexists (loc_in IN VARCHAR2, file_in IN VARCHAR2)
RETURN BOOLEAN;

Provide location and name separately; function returns TRUE if PLVfile is able to open the file with read−only access.

FUNCTION fexists (file_in IN VARCHAR2) RETURN BOOLEAN;

Returns TRUE if PLVfile is able to open the specified file with read−only access.

5.11.7 Opening a file

PROCEDURE fopen
(loc_in IN VARCHAR2, file_in IN VARCHAR2, mode_in IN VARCHAR2);

Opens a file for the specified mode (location and name provided separately) and does not return the handle to the file.

PROCEDURE fopen
(file_in IN VARCHAR2, mode_in IN VARCHAR2 := c_append);

Opens a file for the specified mode and does not return the handle to the file.

```
FUNCTION fopen
(file_in IN VARCHAR2, mode_in IN VARCHAR2 := c_append)
RETURN UTL_FILE.FILE_TYPE;
```

Opens a file for the specified mode and returns the handle to the file.

```
FUNCTION fopen
(loc_in IN VARCHAR2, file_in IN VARCHAR2, mode_in IN VARCHAR2)
RETURN UTL_FILE.FILE_TYPE;
```

Opens a file for the specified mode (location and name provided separately) and returns the handle to the file.

5.11.8 Closing a file

```
PROCEDURE fclose (file_in IN UTL_FILE.FILE_TYPE);
```

Closes the specified file.

```
PROCEDURE fclose_all;
```

Closes all open files.

5.11.9 Reading from a file

```
PROCEDURE get_line
(file_in IN UTL_FILE.FILE_TYPE,
line_out IN OUT VARCHAR2,
eof_out OUT BOOLEAN);
```

Retrieves the next line from the specified file (by file handle). Returns a flag indicating whether the end of the file has been reached.

```
FUNCTION line (file_in IN VARCHAR2, line_num_in IN INTEGER)
RETURN VARCHAR2;
```

Returns the *n*th line from the specified file. This program opens, reads from, and closes the file.

```
FUNCTION infile
(loc_in IN VARCHAR2,
file_in IN VARCHAR2,
text_in IN VARCHAR2,
nth_in IN INTEGER := 1,
start_line_in IN INTEGER := 1,
end_line_in IN INTEGER := 0,
ignore_case_in IN BOOLEAN := TRUE)
RETURN INTEGER;
```

UTL_FILE−version of INSTR. Finds the *n*th appearance of a string (**`text_in`**) within the specified range of lines.

```
FUNCTION infile
(file_in IN VARCHAR2,
text_in IN VARCHAR2,
nth_in IN INTEGER := 1,
start_line_in IN INTEGER := 1,
end_line_in IN INTEGER := 0,
ignore_case_in IN BOOLEAN := TRUE)
RETURN INTEGER;
```

UTL_FILE−version of INSTR. Finds the *n*th appearance of a string (**`text_in`**) within the specified range of lines. In this version you do not have to provide the location of the file separately from the name.

5.11.10 Writing to a file

PROCEDURE put_line
(file_in IN UTL_FILE.FILE_TYPE,line_in IN VARCHAR2);

Adds a line to the end of the specified file (by file handle). This file must already be opened for write or append access.

PROCEDURE append_line (file_in IN VARCHAR2, line_in IN VARCHAR2);

Add a line to the end of the specified file. This program opens the file for append access, writes to the file using **put_line**, and then closes the file.

5.11.11 Copying a file

PROCEDURE fcopy
(ofile_in IN VARCHAR2, nfile_in IN VARCHAR2,
start_in IN INTEGER := 1, end_in IN INTEGER := NULL);

Copies the contents of the old file (**ofile_in**) to the new file (**nfile_in**) for all lines within the specified range.

PROCEDURE file2pstab
(file_in IN VARCHAR2,
table_inout IN OUT PLVtab.vc2000_table,
rows_out OUT INTEGER);

Copies the contents of the file to the PL/SQL table.

PROCEDURE file2list (file_in IN VARCHAR2, list_in IN VARCHAR2);

Copies the contents of the file to the PLVlst list specified by the list name.

PROCEDURE pstab2file
(table_in IN PLVtab.vc2000_table,
rows_in IN INTEGER,
file_in IN VARCHAR2,
mode_in IN VARCHAR2 := c_write);

Copies the contents of the PL/SQL table to a file. You can open the file in either write mode or append mode (in which case the rows are added to the current contents of the file).

5.11.12 Displaying the contents of a file

PROCEDURE display
(file_in IN UTL_FILE.FILE_TYPE,
header_in IN VARCHAR2 := NULL,
start_in IN INTEGER := 1,
end_in IN INTEGER := NULL);

Displays the contents of the file (specified by file handle) using the **p.l** procedure. This version of **display** assumes that the file has been opened.

PROCEDURE display
(file_in IN VARCHAR2,
header_in IN VARCHAR2 := NULL,
start_in IN INTEGER := 1,
end_in IN INTEGER := NULL);

Displays the contents of the file (specified by file name using the **`p.l`** procedure. This version of **`display`** assumes that the file must be opened before reading the contents.

5.11.13 Miscellaneous operations

`PROCEDURE parse_name`
`(file_in IN VARCHAR2, loc_out IN OUT VARCHAR2,`
`name_out IN OUT VARCHAR2);`

Parses a file specification (directory, name, and extension) into two separate strings: the location or directory and the file name itself.

PREVIOUS | HOME | NEXT
5.10 PLVexc: Exception Handling | BOOK INDEX | 5.12 PLVfk: Foreign Key Interface

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

5.12 PLVfk: Foreign Key Interface

The PLVfk (PL/Vision Foreign Key) package is a high−level, easy−to−use interface to look up foreign key information in your tables. See Chapter 19 for details.

5.12.1 Package Constants

```
c_prefix CONSTANT VARCHAR2(1) := 'P';
```

Specifies that the column abbreviation is to be used as a prefix.

```
c_suffix CONSTANT VARCHAR2(1) := 'S';
```

Specifies that the column abbreviation is to be used as a suffix.

```
c_no_change CONSTANT VARCHAR2(10) := 'NO CHANGE';
```

Used to indicate that no change is to be made to the string value.

```
c_int_no_change CONSTANT INTEGER := 0;
```

Used to indicate that no change is to be made to the INTEGER value.

5.12.2 Setting the PLVfk configuration

```
PROCEDURE set_vclen (length_in IN INTEGER);
```

Sets the default VARCHAR2 length for the foreign key named retrieved by the **PLVfk.name** function.

```
PROCEDURE set_id_default
(string_in IN VARCHAR2 := c_no_change,
type_in IN VARCHAR2 := c_no_change);
```

Sets the default value to be used as the suffix or prefix of the name for the ID column.

```
PROCEDURE set_nm_default
(string_in IN VARCHAR2 := c_no_change,
type_in IN VARCHAR2 := c_no_change);
```

Sets the default value to be used as the suffix or prefix of the name for the name column.

5.12.3 Looking up the name

```
FUNCTION name
   (fk_id_in IN INTEGER,
    fk_table_in IN VARCHAR2,
    fk_id_col_in IN VARCHAR2 := c_no_change,
    fk_nm_col_in IN VARCHAR2 := c_no_change,
    max_length_in IN INTEGER := c_int_no_change,
    where_clause_in IN VARCHAR2 := NULL)
RETURN VARCHAR2;
```

Retrieves the name for the specified table and ID.

5.12.4 Looking up the ID

```
FUNCTION id
   (fk_nm_in IN VARCHAR2,
    fk_table_in IN VARCHAR2,
    fk_id_col_in IN VARCHAR2 := c_no_change,
    fk_nm_col_in IN VARCHAR2 := c_no_change,
    max_length_in IN INTEGER := c_int_no_change,
    where_clause_in IN VARCHAR2 := NULL)
RETURN INTEGER;
```

Retrieves the ID (primary key) for the specified table and name or descriptor.

PREVIOUS | HOME | NEXT
5.11 PLVfile: Operating System I/O Manager | BOOK INDEX | 5.13 PLVgen: PL/SQL Code Generator

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

5.13 PLVgen: PL/SQL Code Generator

The PLVgen (PL/Vision GENerator) package provides a set of procedure you can use to generate your own PL/SQL code. See Chapter 16, *PLVgen: Generating PL/SQL Programs* for details.

5.13.1 Package constants

```
c_indent CONSTANT INTEGER := 0;
```

The default initial indentation of generated code.

```
c_incr_indent CONSTANT INTEGER := 3;
```

The default incremental indentation of generated code.

```
c_literal CONSTANT CHAR(1) := '=';
```

The character used to indicate that the default value for the string function is not to be evaluated before placing in the function definition.

```
c_def_length CONSTANT INTEGER := 100;
```

The default length for a string function's local variable.

```
c_none CONSTANT VARCHAR2(1) := 'N';
```

Indicates that no blank lines are to be placed before or after the current line of code.

```
c_before CONSTANT VARCHAR2(1) := 'B';
```

Indicates that a blank line is to be placed before the current line of code.

```
c_after CONSTANT VARCHAR2(1) := 'A';
```

Indicates that a blank line is to be placed after the current line of code.

```
c_both CONSTANT VARCHAR2(2) := 'BA';
```

Indicates that a blank line is to be placed both before and after the current line of code.

5.13.2 Setting the indentation

```
PROCEDURE set_indent
(indent_in IN NUMBER,
incr_indent_in IN NUMBER := c_incr_indent);
```

Sets the initial and incremental indentation.

```
FUNCTION indent RETURN NUMBER;
```

Returns the current value for initial indentation.

```
FUNCTION incr_indent RETURN NUMBER;
```

Returns the current value for incremental indentation.

5.13.3 Setting the author

PROCEDURE set_author (author_in IN VARCHAR2);
Assigns a value for the author string used in program headers.

FUNCTION author RETURN VARCHAR2;
Returns the current author string.

5.13.4 Toggles affecting generated code

PLVgen offers a large selection of toggles or on−off switches, which you can use to modify the content of code generated by this package. Each toggle has a "turn on" procedure, a "turn off" procedure, and a function returning the current state of the toggle (on or off).

PROCEDURE usetrc;
PROCEDURE nousetrc;
FUNCTION using_trc RETURN BOOLEAN;
Controls inclusion of the PLVtrc startup and terminate procedures.

PROCEDURE useexc;
PROCEDURE nouseexc;
FUNCTION using_exc RETURN BOOLEAN;
Controls inclusion of PLVexc exception handlers in exception sections of programs.

PROCEDURE usehdr;
PROCEDURE nousehdr;
FUNCTION using_hdr RETURN BOOLEAN;
Controls inclusion of program headers in packages, procedures, and functions.

PROCEDURE usecmnt;
PROCEDURE nousecmnt;
FUNCTION using_cmnt RETURN BOOLEAN;
Controls inclusion of comment lines in generated code.

PROCEDURE usehlp;
PROCEDURE nousehlp;
FUNCTION using_hlp RETURN BOOLEAN;
Controls inclusion of help text stubs and generation of the **help** procedure in packages.

PROCEDURE usecor;
PROCEDURE nousecor;
FUNCTION using_cor RETURN BOOLEAN;
Controls inclusion of code required to CREATE OR REPLACE program units.

PROCEDURE useln;
PROCEDURE nouseln;
FUNCTION usingln RETURN BOOLEAN;
Controls inclusion of line numbers in prefix of generated code.

PROCEDURE usemin;
Turns off all the above toggles.

PROCEDURE usemax;
Turns on all the above toggles.

5.13.5 Help generators

PROCEDURE helpproc
(prog_in IN VARCHAR2 := NULL, indent_in IN INTEGER := 0);

Generates a procedure that gives main−topic help for the specified program unit.

PROCEDURE helptext (context_in IN VARCHAR2 := PLVhlp.c_main);

Generates a comment block in the correct format to be used as online help text with the PLVhlp package.

5.13.6 Generating a package

PROCEDURE pkg (name_in IN VARCHAR2);

Generates the skeleton structure for a package's specification and body.

5.13.7 Generating a procedure

PROCEDURE proc
(name_in IN VARCHAR2,
params_in IN VARCHAR2 := NULL,
exec_in IN VARCHAR2 := NULL,
incl_exc_in IN BOOLEAN := TRUE,
indent_in IN INTEGER := 0,
blank_lines_in IN VARCHAR2 := c_before);

Generates a procedure of the specified name. You can also provide a parameter list and one or more executable lines. Finally, you can decide to include an exception section, indent the code, and perform blank−line processing.

5.13.8 Generating functions

A function has a RETURN datatype. PLVgen allows you to generate string, numeric, date, and Boolean functions. You can also supply literal and symbol default values. As a result, the **func** procedure is overloaded as shown:

PROCEDURE func
(name_in IN VARCHAR2,
datadesc_in VARCHAR2,
defval_in IN VARCHAR2 := NULL,
length_in IN INTEGER := c_def_length,
incl_exc_in IN BOOLEAN := TRUE);

Generates a string function (since the datatype for the **datdesc_in** parameter is VARCHAR2).

PROCEDURE func
(name_in IN VARCHAR2,
datadesc_in datatype,
defval_in IN datatype := NULL,
incl_exc_in IN BOOLEAN := TRUE);

Generates a function of the specified **datatype**, which is either NUMBER, DATE, or BOOLEAN. Notice that the default has the same datatype as the **datadesc_in** parameter. This is a default value that is evaluated as a literal.

PROCEDURE func
(name_in IN VARCHAR2,

```
datadesc_in datatype,
defval_in IN VARCHAR2,
incl_exc_in IN BOOLEAN := TRUE);
```

Generates a function of the specified **datatype**, which is either NUMBER, DATE, or BOOLEAN. Notice that the default in this version is a string. When you use this format, the default value is treated as an expression that is *not* evaluated.

5.13.9 Generating get−and−set routines

Get−and−set routines provide a programmatic layer of code around a private data structure. As a result, the get−and−sets or "gas" generators have associated with them a datatype. PLVgen allows you to generate string, numeric, date, and Boolean get−and−sets. You can also supply literal and symbol default values. As a result, the **gas** procedure is overloaded with the following flavors:

```
PROCEDURE gas
(name_in IN VARCHAR2,
valtype_in VARCHAR2,
defval_in IN VARCHAR2 := NULL,
length_in IN INTEGER := c_def_length);
```

Generates a string function (since the datatype for the **datdesc_in** parameter is VARCHAR2).

```
PROCEDURE gas
(name_in IN VARCHAR2,
valtype_in datatype,
defval_in IN datatype := NULL);
```

Generates get−and−sets of the specified **datatype**, which is either NUMBER, DATE, or BOOLEAN. Notice that the default has the same datatype as the **datadesc_in** parameter. This is a default value that is evaluated as a literal.

```
PROCEDURE gas
(name_in IN VARCHAR2, valtype_in datatype,
defval_in IN VARCHAR2);
```

Generates get−and−sets of the specified **datatype**, which is either NUMBER, DATE, or BOOLEAN. Notice that the default in this version is a string. When you use this format, the default value is an expression that is *not* evaluated.

```
PROCEDURE toggle (name_in IN VARCHAR2 := NULL);
```

Generates a variation of get−and−set based on a Boolean toggle. If you do not give a name, **turn_on** and **turn_off** are used as the on−off procedure names.

5.13.10 Miscellaneous code generators

```
PROCEDURE curdecl
(cur_in IN VARCHAR2,
ind_in IN INTEGER := 0,
table_in IN VARCHAR2 := NULL,
collist_in IN VARCHAR2 := NULL,
gen_rec_in IN BOOLEAN := TRUE);
```

Generates a cursor declaration with the SQL statement formatted for maximum readability.

```
PROCEDURE cfloop (table_in IN VARCHAR2);
```

Generates a cursor FOR loop and framework for a cursor declaration.

```
PROCEDURE recfnd (table_in IN VARCHAR2);
```

Generates a function that returns TRUE if a record is found, FALSE otherwise.

PROCEDURE timer (plsql_in IN VARCHAR2);

Generates a function that returns TRUE if a record is found, FALSE otherwise.

PREVIOUS | HOME | NEXT

5.12 PLVfk: Foreign Key Interface | BOOK INDEX | 5.14 PLVhlp: Online Help Architechture

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

5.14 PLVhlp: Online Help Architechture

The PLVhlp (PL/Vision HeLP) package provides an architecture with which you can build online help for your own PL/SQL programs. See Chapter 17, *PLVhlp: Online Help for PL/SQL Programs* for details.

5.14.1 Package constants

c_main CONSTANT CHAR(4) := 'HELP';

The keyword used to designate the main help for a program. This is the default kind of help to be shown.

c_examples CONSTANT VARCHAR2(30) := 'EXAMPLES';

The keyword used to designate the section of help displaying examples for a program. Other keywords can be added to the package to support other kinds of sections.

5.14.2 Setting the page size

PROCEDURE set_pagesize (pagesize_in IN NUMBER);

Sets the number of lines of help text to be displayed before a pause. The default is 25.

FUNCTION pagesize RETURN NUMBER;

Returns the number of lines of help text to be displayed before a pause.

5.14.3 Help text stub generators

*FUNCTION help_start (context_in IN VARCHAR2 := NULL)
RETURN VARCHAR2;*

Returns the string needed to start a comment to be used as online help text.

*FUNCTION help_end (context_in IN VARCHAR2 := NULL)
RETURN VARCHAR2;*

Returns the string needed to end a comment to be used as online help text.

5.14.4 Displaying online help

*PROCEDURE show (context_in IN VARCHAR2, part_in IN
VARCHAR2 := c_main);*

Displays the first page of help for the specified context.

PROCEDURE more;

Displays the next page of help, if there is any.

5.13 PLVgen: PL/SQL Code Generator

5.15 PLVio: Input/Output Processing

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

5.15 PLVio: Input/Output Processing

The PLVio (PL/Vision Input/Output) package consolidates all of the logic required to read from and write to repositories for PL/SQL source code. See Chapter 12, *PLVio: Reading and Writing PL/SQL Source Code* for details.

5.15.1 Package constants and exceptions

```
c_name PLV.plsql_identifier%TYPE := 'name';
c_type PLV.plsql_identifier%TYPE := 'type';
c_text PLV.plsql_identifier%TYPE := 'text';
c_line PLV.plsql_identifier%TYPE := 'line';
c_schema PLV.plsql_identifier%TYPE := 'owner';
```

The default names for the columns of the database table repository. You can override these names in calls to **`setsrc`** and **`settrg`**.

```
c_notset CONSTANT VARCHAR2(1) := 'U';
```

The value used by PLVio to detect when a repository (source or target) have not yet been set.

```
insert_failure EXCEPTION;
```

Exception raised when PLVio is unable to insert or put a line to the target repository.

5.15.2 Package records

```
TYPE line_type IS RECORD
(text VARCHAR2(2000) := NULL,
len INTEGER := NULL,
pos INTEGER := 1,
line INTEGER := 0, /* line # in original */
line# INTEGER := 0, /* line # for new */
is_blank BOOLEAN := FALSE,
eof BOOLEAN := FALSE,
indent INTEGER := 0,
unindent BOOLEAN := FALSE,
continuation BOOLEAN := FALSE,
blank_line_before BOOLEAN := FALSE);
```

The **`line_type`** record TYPE defines the structure for a line datatype in PLVio.

```
empty_line line_type;
```

Predefined empty line that can be used to initialize a line's fields.

5.15.3 Source and target repository type functions

```
FUNCTION file_source RETURN BOOLEAN;
FUNCTION pstab_source RETURN BOOLEAN;
FUNCTION dbtab_source RETURN BOOLEAN;
FUNCTION string_source RETURN BOOLEAN;
```

Returns TRUE if the current source repository matches that indicated by the name, FALSE otherwise.

```
FUNCTION file_target RETURN BOOLEAN;
FUNCTION pstab_target RETURN BOOLEAN;
FUNCTION dbtab_target RETURN BOOLEAN;
FUNCTION string_target RETURN BOOLEAN;
FUNCTION stdout_target RETURN BOOLEAN;
```

Returns TRUE if the current target repository matches that indicated by the name, FALSE otherwise.

```
FUNCTION nosrc RETURN BOOLEAN;
FUNCTION notrg RETURN BOOLEAN;
```

These functions return TRUE if the source and target repositories, respectively, have not yet been set.

```
FUNCTION srctype RETURN VARCHAR2;
FUNCTION trgtype RETURN VARCHAR2;
```

These functions return the current source and target repository types. The values returned can be matched against constants in the PLV package.

5.15.4 Managing the source repository

```
PROCEDURE setsrc
(srctype_in IN VARCHAR2,
name_in IN VARCHAR2 := 'user_source',
name_col_in IN VARCHAR2 := c_name,
type_col_in IN VARCHAR2 := c_type,
line#_col_in IN VARCHAR2 := c_line,
text_col_in IN VARCHAR2 := c_text,
schema_col_in IN VARCHAR2 := NULL);
```

This procedure sets the source repository. You provide the repository type, the name, and then, if a database table, the names of the columns in the table.

```
PROCEDURE initsrc
(starting_at_in IN INTEGER,
ending_at_in IN INTEGER,
where_in IN VARCHAR2 := NULL);
PROCEDURE initsrc
(starting_at_in IN VARCHAR2 := NULL,
ending_at_in IN VARCHAR2 := NULL,
where_in IN VARCHAR2 := NULL);
```

The **initsrc** procedures initialize the source after it has been set. You can provide additional information in the call to **initsrc** to restrict which rows are retrieved from the source, including a WHERE clause and start−end line numbers or strings.

```
PROCEDURE usrc
(starting_at_in IN VARCHAR2 := NULL,
ending_at_in IN VARCHAR2 := NULL,
where_in IN VARCHAR2 := NULL);
PROCEDURE usrc
```

```
(starting_at_in IN INTEGER,
ending_at_in IN INTEGER,
where_in IN VARCHAR2 := NULL);
```

The two **usrc** procedures set the source repository to the USER_SOURCE data dictionary view and then initialize the source with a call to **initsrc**, passing along the arguments provided to it (notice that they match those of **initsrc**).

```
PROCEDURE asrc
(starting_at_in IN VARCHAR2 := NULL,
ending_at_in IN VARCHAR2 := NULL,
where_in IN VARCHAR2 := NULL);
PROCEDURE asrc
(starting_at_in IN INTEGER,
ending_at_in IN INTEGER,
where_in IN VARCHAR2 := NULL);
```

The two **asrc** procedures set the source repository to the ALL_SOURCE data dictionary view and then initialize the source with a call to **initsrc**, passing along the arguments provided to it (notice that they match those of **initsrc**).

```
FUNCTION srcselect RETURN VARCHAR2;
```

Returns the current SELECT statement associated with the source repository. This is only relevant when the source type is a database table (**PLV.dbtab**).

```
PROCEDURE closesrc;
```

Closes the source repository. If a database table, the cursor is closed. If a file, the file is closed.

5.15.5 Managing the source WHERE clause

When the source type is a database table, you can manipulate the WHERE clause which identifies or restricts those rows that are read from the source table. The following constants and programs all have an impact on the WHERE clause.

```
c_first CONSTANT VARCHAR2(1) := 'F';
c_last CONSTANT VARCHAR2(1) := 'L';
c_before CONSTANT VARCHAR2(1) := 'B';
c_after CONSTANT VARCHAR2(1) := 'A';
```

These constants indicate the type of match to be performed when using the **line_with** and **set_line_limit** procedures.

```
PROCEDURE set_srcwhere (where_in IN VARCHAR2 := NULL);
```

Sets the source repository WHERE clause, either by replacing it or providing additional elements.

```
PROCEDURE rem_srcwhere;
```

Removes any additional elements that have been added to the WHERE clause.

```
FUNCTION line_with
(text_in IN VARCHAR2,
loc_type_in IN VARCHAR2 := c_first,
after_in IN INTEGER := NULL)
RETURN INTEGER;
FUNCTION line_with
(text_in IN VARCHAR2,
loc_type_in IN VARCHAR2 := c_first,
after_in IN VARCHAR2,
after_loc_type_in IN VARCHAR2 := c_first)
RETURN INTEGER;
```

The **line_with** functions return the line number associated with the values passed to them. They answer questions like "What is the first line in the PLVvu package containing `IF'?" and "What is the last line in the PLV package containing `CONSTANT' that comes after the string `VARCHAR2'?"

```
PROCEDURE set_line_limit
(line_in IN INTEGER, loc_type_in IN VARCHAR2 := c_first);
```

Adds an element to the WHERE clause of the source repository restricting the text retrieved by a line number.

```
PROCEDURE rem_line_limit (line_in IN INTEGER);
```

Use this procedure to remove from the WHERE clause an element added by **set_line_limit**.

5.15.6 Managing the target repository

```
target_table PLVtab.vc2000_table;
```

The PL/SQL table that contains the code when the target is set to PL/SQL table.

```
target_row BINARY_INTEGER;
```

The number of rows in the target PL/SQL table.

```
PROCEDURE settrg
(trgtype_in IN VARCHAR2,
name_in IN VARCHAR2 := 'PLV_source',
target_name_col_in IN VARCHAR2 := 'name',
trgtype_col_in IN VARCHAR2 := 'type',
target_line#_col_in IN VARCHAR2 := 'line',
target_text_col_in IN VARCHAR2 := 'text');
```

Sets the target and, if a database table, the structure of the table containing the text. This program also calls **inittrg** (not so with the source repository).

```
PROCEDURE disptrg
(header_in IN VARCHAR2 := NULL,
start_in IN INTEGER := 1,
end_in IN INTEGER := target_row,
type_in IN VARCHAR2 := trgtype);
```

Displays the contents of the target repository. The second and third arguments apply to PL/SQL tables and file repositories only. The fourth argument allows you to override the current target type to display the contents of another type repository.

```
PROCEDURE inittrg;
```

Initializes the target repository; this is called by **settrg** so there is very little reason to execute this directly.

```
FUNCTION trgstg RETURN VARCHAR2;
```

Returns the string target repository. Separate lines of text in the repository are separated by a **CHR(10)** character.

```
PROCEDURE closetrg;
```

Closes the target repository.

```
PROCEDURE clrtrg
(program_name_in IN VARCHAR2 := NULL,
program_type_in IN VARCHAR2 := NULL);
```

Clears the specified repository. If a database table, the rows are deleted. If a string or PL/SQL table, the repository is set to NULL.

5.15.7 Reading and writing lines

The whole point of PLVio is to read from the source repository and/or write to the target repository. In PLVio lingo, this means that you get a line from the source and put a line to the target.

```
PROCEDURE initline
(line_inout IN OUT line_type,
text_in IN VARCHAR2 := NULL,
len_in IN INTEGER := NULL,
pos_in IN INTEGER := 1,
line#_in IN INTEGER := 0,
is_blank_in IN BOOLEAN := FALSE,
eof_in IN BOOLEAN := FALSE);
```

Initializes a line record (defined with the PLV**io.line_type** record TYPE) with the values provided in the parameter list.

```
PROCEDURE get_line
(line_inout IN OUT line_type,
curr_line#_in IN INTEGER := NULL);
```

Gets a line from the source repository and deposits it in the line record provided in the argument list.

```
FUNCTION rest_of_line
(line_in IN line_type, pos_in IN INTEGER := line_in.pos)
RETURN VARCHAR2;
```

Returns the rest of the line that has not yet been scanned, based on the current position in the line (provided by the second argument).

```
PROCEDURE put_line (line_in IN line_type);
```

Puts a line record in the target repository.

```
PROCEDURE put_line
(string_in IN VARCHAR2, line#_in IN INTEGER := NULL);
```

Puts a string in the target repository. Use this version of **put_line** when you are not otherwise working with a record defined with the **line_type** record TYPE and simply have a string to move to the repository.

5.15.8 Saving and restoring repository settings

```
PROCEDURE savesrc;
```

Requests that the current settings for the source repository be saved and then restored upon close of the (new) source (the default).

```
PROCEDURE nosavesrc;
```

Requests that saves and restores not be performed.

```
FUNCTION saving_src RETURN BOOLEAN;
```

Returns TRUE if saves and restores are being performed.

```
PROCEDURE savetrg;
```

Requests that the current settings for the target repository be saved and then restored upon close of the (new) target (the default).

`PROCEDURE nosavetrg;`

Requests that saves and restores for the target not be performed.

`FUNCTION saving_trg RETURN BOOLEAN;`

Returns TRUE if saves and restores for the target are being performed.

`PROCEDURE restoresrc;`

Restores the source repository to its previous value.

`PROCEDURE restoretrg;`

Restores the target repository to its previous value.

5.15.9 Miscellaneous PLVio programs

`PROCEDURE src2trg (close_in IN BOOLEAN := TRUE);`

Transfers the contents of the source repository directly to the target repository.

5.15.10 Tracing PLVio activity

`PROCEDURE display;`

Requests that PLVio actions be displayed as they occur.

`PROCEDURE nodisplay;`

Requests that PLVio actions not be displayed as they occur.

`FUNCTION displaying RETURN BOOLEAN;`

Returns TRUE if PLVio is displaying its actions.

PREVIOUS | HOME | NEXT
5.14 PLVhlp: Online Help Architechture | BOOK INDEX | 5.16 PLVlex: Lexical Analysis

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

5.16 PLVlex: Lexical Analysis

The PLVlex (PL/Vision LEXical analysis) package provides generic string−parsing extensions to PL/SQL; these extensions include an awareness of the syntax and delimiters of the PL/SQL language. See the companion disk for details.

5.16.1 Analyzing PL/SQL string content

```
FUNCTION is_delimiter
(character_in IN VARCHAR2, exclude_in IN VARCHAR2 := NULL)
RETURN BOOLEAN;
```

Returns TRUE if the string is a PL/SQL delimiter.

```
FUNCTION is_oneline_comment (token_in IN VARCHAR2) RETURN BOOLEAN;
```

Returns TRUE if the string is a single−line comment indicator (a double hyphen).

```
FUNCTION starts_multiline_comment (token_in IN VARCHAR2)
RETURN BOOLEAN;
```

Returns TRUE if the string is equal to `/*`, which signals the start of a multiline or block comment.

```
FUNCTION ends_multiline_comment (token_in IN VARCHAR2)
RETURN BOOLEAN;
```

Returns TRUE if the string is equal to `*/`, which signals the end of a multiline or block comment.

5.16.2 Scanning PL/SQL strings

```
FUNCTION next_atom_loc
(string_in IN VARCHAR2, start_loc_in IN NUMBER)
RETURN NUMBER;
```

Returns the location of the beginning of the next PL/SQL atomic in the string.

```
PROCEDURE get_next_atomic
(line_in IN VARCHAR2,
start_pos_in IN VARCHAR2,
atomic_out OUT VARCHAR2,
new_start_pos_out OUT INTEGER,
line_len_in IN INTEGER := NULL);
```

Gets the next PL/SQL atomic from the string. This procedure builds upon **`PLVlex.next_atom_loc`** and several low−level PLVprs functions to scan the string from the perspective of a line of PL/SQL source code.

```
PROCEDURE get_next_token
(line_in IN VARCHAR2,
start_pos_in IN VARCHAR2,
token_out IN OUT VARCHAR2,
new_start_pos_out IN OUT INTEGER,
skip_spaces_in IN BOOLEAN,
line_len_in IN INTEGER := NULL,
full_qualified_name_in IN BOOLEAN := FALSE);
```

Gets the next PL/SQL token (which could be composed of multiple atomics) from the string.

PREVIOUS | HOME | NEXT
5.15 PLVio: Input/Output Processing | BOOK INDEX | 5.17 PLVlog: Logging Facility

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8*i* Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

5.17 PLVlog: Logging Facility

The PLVlog (PL/Vision LOGging) package provides a powerful, generic logging facility for PL/SQL packages. See Chapter 21, *PLVlog and PLVtrc: Logging and Tracing* for details.

5.17.1 Package constants

`c_file CONSTANT VARCHAR2(100) := 'PLV.log';`

The default name of the file contains the PL/Vision log when writing to an operating system file. This can be –– and usually would be –– overridden with your own file name. It is only applicable if you are using PL/SQL Release 2.3 or above.

`c_noaction CONSTANT PLV.plsql_identifier%TYPE := '*NO ROLLBACK*';`

Name for rollback activity to not perform any rollback.

`c_none CONSTANT PLV.plsql_identifier%TYPE := '*FULL*';`

Name to indicate that a full rollback should occur.

`c_default CONSTANT PLV.plsql_identifier%TYPE := '*DEFAULT*';`

Name to indicate that a rollback should occur to the default savepoint.

`c_last CONSTANT PLV.plsql_identifier%TYPE := '*PLVRB-LAST*';`

Name to indicate that a rollback should occur to the last savepoint recorded by PLVrb.

```
c_PLVlogsp CONSTANT PLV.plsql_identifier%TYPE :=
'PLVlog_savepoint';
```

The default savepoint issued after a write to the log.

5.17.2 Controlling logging activity

`PROCEDURE turn_on;`

Turns on the logging activity; calls to **`put_line`** write information to the log (default).

`PROCEDURE turn_off;`

Turns off the log mechanism.

`FUNCTION tracing RETURN BOOLEAN;`

Returns TRUE if the log is active.

5.17.3 Selecting the log type

```
PROCEDURE sendto
(type_in IN VARCHAR2, file_in IN VARCHAR2 := NULL);
```

Generic program to indicate the type of log (the repository to which information will be sent). The valid types are stored in the PLV package. If you choose **`PLV.file`**, you need to also provide a file name. You can also set the log type by calling one of the following procedures.

`PROCEDURE to_pstab;`

Requests that the information be sent to a PL/SQL table.

`PROCEDURE to_dbtab;`

Requests that the information be sent to a database table.

`PROCEDURE to_file (file_in IN VARCHAR2);`

Requests that the information be sent to an operating system file.

`PROCEDURE to_stdout;`

Requests that the information be sent to standard output.

`FUNCTION logtype RETURN VARCHAR2;`

Returns the current log target type. This type will be one of the repository constants defined in the PLV package.

5.17.4 Writing to the log

```
PROCEDURE put_line
(context_in IN VARCHAR2,
code_in IN INTEGER,
string_in IN VARCHAR2 := NULL,
create_by_in IN VARCHAR2 := USER,
rb_to_in IN VARCHAR2 := c_default,
override_in IN BOOLEAN := FALSE);
```

Writes a line to the PLVlog repository. This version of **`put_line`** allows you to specify a full set of values for the log record.

```
PROCEDURE put_line
(string_in IN VARCHAR2,
rb_to_in IN VARCHAR2 := c_default,
override_in IN BOOLEAN := FALSE);
```

This version of **`put_line`** keeps to an absolute minimum what you have to/want to provide to write a line to the log.

5.17.5 Reading the log

```
PROCEDURE get_line
(row_in IN INTEGER,
context_out OUT VARCHAR2,
code_out OUT INTEGER,
string_out OUT VARCHAR2,
create_by_out OUT VARCHAR2,
create_ts_out OUT DATE);
```

Reads a row from the PL/SQL table log, parses the contents, and returns the individual values in the separate OUT arguments of the parameter list.

`PROCEDURE display (header_in IN VARCHAR2 := 'PL/Vision Log');`

Displays the contents of the current PLVlog log (either in the database table or in the PL/SQL table).

5.17.6 Managing the log

```
PROCEDURE clear_pstab;
```

Empties the PL/SQL table associated with the PLVlog mechanism.

```
FUNCTION pstab_count RETURN INTEGER;
```

Returns the number of rows filled in the PLVlog PL/SQL table.

```
PROCEDURE set_dbtab
(table_in IN VARCHAR2 := 'PLV_log',
context_col_in IN VARCHAR2 := 'context',
code_col_in IN VARCHAR2 := 'code',
text_col_in IN VARCHAR2 := 'text',
create_ts_col_in IN VARCHAR2 := 'create_ts',
create_by_col_in IN VARCHAR2 := 'create_by');
```

Determines which database table is to be used by PLVlog for logging. PLVlog relies on dynamic SQL, so you can at runtime specify the table and column names for this table.

```
PROCEDURE fclose;
```

Closes the operating system file if you have chosen a file for the log repository.

```
PROCEDURE ps2db;
```

Transfers contents of PL/SQL log to the PLVlog database table.

5.17.7 Rolling back in PLVlog

```
PROCEDURE do_rollback;
```

Turns on the issuing of a ROLLBACK before an INSERT into the log.

```
PROCEDURE nodo_rollback;
```

Turns off the issuing of a ROLLBACK before an INSERT into the log.

```
FUNCTION rolling_back RETURN BOOLEAN;
```

Returns TRUE if PLVlog is issuing a ROLLBACK before an INSERT.

```
PROCEDURE rb_to (savepoint_in IN VARCHAR2 := c_none);
```

Sets the default savepoint used both for the ROLLBACK command before a log insert and the SAVEPOINT command after the log insert.

```
PROCEDURE rb_to_last;
```

Sets the savepoint used both for the ROLLBACK command before a log insert to the last savepoint known to PLVrb.

```
PROCEDURE rb_to_default;
```

Sets the default savepoint used both for the ROLLBACK command before a log insert to the PLVlog default savepoint.

```
PROCEDURE set_sp (savepoint_in IN VARCHAR2);
```

Sets the name of the savepoint to be set after the call to **`put_line`** to write a line to the log.

PREVIOUS | HOME | NEXT
5.16 PLVlex: Lexical Analysis | BOOK INDEX | 5.18 PLVlst: List Manager

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8*i* Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

5.18 PLVlst: List Manager

The PLVlst (PL/Vision LiST manager) package provides a generic list manager for PL/SQL built on PL/SQL tables. See the companion disk for details.

5.18.1 Package exceptions

```
list_undefined EXCEPTION;
```

Raised when you attempt to perform a PLVlst operation on an undefined list.

```
list_full EXCEPTION;
```

Raised when you attempt to add another item to a list which is already full (currently limited to maximum of 10,000 items in a list).

```
out_of_bounds EXCEPTION;
```

Exception raised when you try to reference an item in the list that is not defined.

5.18.2 Creating and destroying lists

```
PROCEDURE make (list_in IN VARCHAR2);
```

Allocates storage for a new list of the specified name.

```
FUNCTION is_made (list_in IN VARCHAR2) RETURN BOOLEAN;
```

Returns TRUE if the specified list already exists.

```
PROCEDURE destroy (list_in IN VARCHAR2);
```

Destroys the list, freeing up associated memory.

5.18.3 Modifying list contents

```
PROCEDURE appenditem (list_in IN VARCHAR2, item_in IN VARCHAR2);
```

Adds an item to the end of the specified list.

```
PROCEDURE insertitem
(list_in IN VARCHAR2, position_in IN NUMBER,
item_in IN VARCHAR2);
```

Inserts an item at the specified position in the list.

```
PROCEDURE prependitem (list_in IN VARCHAR2, item_in IN VARCHAR2);
```

Inserts an item at the first position of the list.

```
PROCEDURE replaceitem
(list_in IN VARCHAR2, position_in IN NUMBER,
item_in IN VARCHAR2);
```

Replaces the *n*th item in the list with the new item.

PROCEDURE deleteitem (list_in IN VARCHAR2, item_in IN VARCHAR2);

Deletes the first occurrence of the specified item from the list.

PROCEDURE deleteitem (list_in IN VARCHAR2, position_in IN NUMBER);

Deletes the item at the specified location in the list.

5.18.4 Analyzing list contents

FUNCTION getitem (list_in IN VARCHAR2, position_in IN NUMBER);

Returns the *n*th item from the list.

*FUNCTION getposition (list_in IN VARCHAR2, item_in IN VARCHAR2)
RETURN NUMBER;*

Returns the position of the specified item in the list.

FUNCTION nitems (list_in IN VARCHAR2) RETURN INTEGER;

Returns the number of items currently in the list.

PROCEDURE display (list_in IN VARCHAR2);

Displays the contents of the specified list.

PREVIOUS | HOME | NEXT
5.17 PLVlog: Logging Facility | BOOK INDEX | 5.19 PLVmsg: Message Handling

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

5.19 PLVmsg: Message Handling

The PLVmsg (PL/Vision MeSsaGe) package consolidates various kinds of message text in a single PL/SQL−based repository. See Chapter 9, *PLVmsg: Single−Sourcing PL/SQL Message Text* for details.

5.19.1 Restricting use of text

```
PROCEDURE restrict;
```

Restricts text for Oracle error numbers to be retrieved from a call to SQLERRM (the default).

```
PROCEDURE norestrict;
```

Directs PLVmsg to retrieve message text only from the PL/SQL table.

```
FUNCTION restricting RETURN BOOLEAN;
```

Describes current state of restrict toggle: TRUE if restricting text to SQLERRM, FALSE otherwise.

5.19.2 Managing and accessing message text

```
FUNCTION text (num_in IN INTEGER := SQLCODE) RETURN VARCHAR2;
```

Returns the text stored in the PL/SQL table of the PLVmsg package for the specified row number.

```
PROCEDURE add_text (num_in IN INTEGER, text_in IN VARCHAR2);
```

Adds text to the PL/SQL table of the PLVmsg package at the specified row number.

```
PROCEDURE load_from_dbms
(table_in IN VARCHAR2,
where_clause_in IN VARCHAR2 := NULL,
code_col_in IN VARCHAR2 := 'error_code',
text_col_in IN VARCHAR2 := 'error_text');
```

Loads the PL/SQL table of the PLVmsg package from the specified table using DBMS_SQL. You can specify the table name, optional WHERE clause, and even the names of the columns.

```
FUNCTION min_row RETURN BINARY_INTEGER;
```

Returns the lowest row number in use by the PLVmsg PL/SQL table. This is necessary for PL/SQL tables in PL/SQL Releases 2.2 and below.

```
FUNCTION max_row RETURN BINARY_INTEGER;
```

Returns the highest row number in use by the PLVmsg PL/SQL table. This is necessary for PL/SQL tables in PL/SQL Releases 2.2 and below.

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8*i* Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

5.20 PLVobj: Object Interface

The PLVobj (PL/Vision OBJect) package provides a programmatic interface to the PL/SQL objects stored in the ALL_OBJECTS data dictionary view. See Chapter 20 for details.

5.20.1 Tracing PLVobj activity

```
PROCEDURE display;
```

Turns on display of information about activity occurring in PLVobj.

```
PROCEDURE nodisplay;
```

Turns off display of PLVobj activity.

```
FUNCTION displaying RETURN BOOLEAN;
```

Returns TRUE if showing activity in PLVobj.

5.20.2 General constants and exceptions

```
no_name_specified EXCEPTION;
```

Exception raised when you try to perform an operation but have not specified the name of the object (the "current object" has not been set).

```
c_pkg_spec CONSTANT VARCHAR2(1) := 'S';
c_pkg_body CONSTANT VARCHAR2(1) := 'B';
c_entire_pkg CONSTANT VARCHAR2(2) := 'SB';
c_proc CONSTANT VARCHAR2(2) := 'P';
c_func CONSTANT VARCHAR2(2) := 'F';
```

Names for the different types of program units. You can use these in calls to **`set_type`** or simply pass the literal values as part of a single type−name string.

```
c_procedure CONSTANT VARCHAR2(30) := 'PROCEDURE';
c_function CONSTANT VARCHAR2(30) := 'FUNCTION';
c_synonym CONSTANT VARCHAR2(30) := 'SYNONYM';
c_package CONSTANT VARCHAR2(30) := 'PACKAGE';
c_package_body CONSTANT VARCHAR2(30) := 'PACKAGE BODY';
```

Full names of program unit types as they are found in ALL_OBJECTS.

5.20.3 Setting the current object

```
PROCEDURE setcurr
(name_in IN VARCHAR2, type_in IN VARCHAR2 := NULL);
```

Sets the current object for other PLVobj modules. When you call **`setcurr`**, you set the schema, name, and type for the object. You can also call the individual programs listed below to set a single part of the current object.

PROCEDURE set_schema (schema_in IN VARCHAR2 := USER);
Sets the schema for the current object.

PROCEDURE set_type (type_in IN VARCHAR2);
Sets the type for the current object.

PROCEDURE set_name (name_in IN VARCHAR2);
Sets the name for the current object.

5.20.4 Accessing the current object

FUNCTION currname RETURN VARCHAR2;
Returns the name of the current object.

FUNCTION currtype RETURN VARCHAR2;
Returns the type of the current object.

FUNCTION currschema RETURN VARCHAR2;
Returns the schema of the current object.

FUNCTION fullname RETURN VARCHAR2;
Returns the full name of the current object (the different elements concatenated together).

PROCEDURE showcurr (show_header_in IN BOOLEAN := TRUE);
Displays the full name of the current object.

5.20.5 Interfacing with the PLVobj cursor

PROCEDURE open_objects;
Opens the PLVobj cursor for the current object settings.

PROCEDURE fetch_object;
PROCEDURE fetch_object
(name_out OUT VARCHAR2, type_out OUT VARCHAR2);
Two overloaded versions to fetch the next row from the PLVobj cursor. The first version fetches the next object into the current object. The second version allows you to fetch the next object into local variables, leaving the current object unchanged.

FUNCTION more_objects RETURN BOOLEAN;
Returns TRUE if the last fetch from the PLVobj cursor returned a record.

PROCEDURE close_objects;
Closes the PLVobj cursor.

5.20.6 Programmatic cursor FOR loop elements

```
PROCEDURE loopexec
   (module_in IN VARCHAR2,
    exec_in IN VARCHAR2 := c_show_object,
    placeholder_in IN VARCHAR2 := c_leph,
    name_format_in IN VARCHAR2 := c_modspec);
```

The **loopexec** procedure simulates a cursor FOR loop through a programmatic interface using dynamic PL/SQL. You can modify the behavior of **loopexec** through the use of the following constants.

```
c_leph CONSTANT VARCHAR2(10) := ':rowobj';
```

The default Loop Exec PlaceHolder string.

```
c_show_object CONSTANT VARCHAR2(100) := 'p.l (:rowobj)';
```

The default action for **`loopexec`**, which is to display the set of objects that are fetched by the cursor.

```
c_modspec CONSTANT VARCHAR2(1) := 'S';
c_modname CONSTANT VARCHAR2(1) := 'N';
```

Named constants for the two different formats for object names manipulated by **`loopexec`**: S for module specification and N for module name.

```
v_letab PLVtab.vc2000_table;
v_lerowind INTEGER;
```

The PL/SQL table and row count variable used to store all the objects retrieved by the programmatic cursor FOR loop, **`loopexec`**.

5.20.7 Saving and restoring PLVobj settings

```
PROCEDURE savecurr;
```

Saves the current object to private variables so that it can be restored.

```
PROCEDURE restcurr;
```

Restore the current object from the saved setting.

5.20.8 Miscellaneous PLVobj programs

```
PROCEDURE vu2pstab
(module_in IN VARCHAR2,
table_out OUT PLVtab.vc2000_table,
num_objects_inout IN OUT INTEGER);
```

Copies the set of objects identified by the PLVobj cursor to a PL/SQL table.

```
PROCEDURE convobj
(name_inout IN OUT VARCHAR2,
type_inout IN OUT VARCHAR2,
schema_inout IN OUT VARCHAR2);
```

Converts a single object string (which can have a complex format such as *type:schema.name*)

```
PROCEDURE bindobj
(cur_in IN INTEGER,
name_col_in IN VARCHAR2 := 'name',
type_col_in IN VARCHAR2 := 'type',
schema_col_in IN VARCHAR2 := NULL);
```

Encapsulates calls to DBMS_SQL.BIND_VARIABLE to allow binding of the different elements of the current object into the specified cursor. You can bind all three elements as a subset; binding will only occur for those arguments that have non−NULL values.

```
PROCEDURE convert_type (type_inout IN OUT VARCHAR2);
```

Converts a variety of abbreviations for program unit types into the strings employed in the ALL_OBJECTS data dictionary view. The string "BODY", for example, is converted to the full "PACKAGE BODY".

PREVIOUS HOME NEXT
BOOK INDEX

5.19 PLVmsg: Message Handling

5.21 PLVprs: String Parsing

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

5.21 PLVprs: String Parsing

The PLVprs (PL/Vision PaRSe) package offers a set of programs which provide generic and flexible string parsing capabilities. See Chapter 10, *PLVprs, PLVtkn, and PLVprsps: Parsing Strings* for details.

5.21.1 Package constants

```
c_ignore_case CONSTANT VARCHAR2(1) := 'I';
c_respect_case CONSTANT VARCHAR2(1) := 'R';
```

Use these constants to indicate whether you want case to be ignored or respected in the current operation.

```
c_all CONSTANT VARCHAR(3) := 'ALL';
c_word CONSTANT VARCHAR(4) := 'WORD';
c_delim CONSTANT VARCHAR(5) := 'DELIM';
```

The different types of atomics; **c_all** indicates "all atomics"; **c_word** indicates "words only"; **c_delim** indicates "delimiters only".

```
std_delimiters CONSTANT VARCHAR2 (50) :=
'!@#$%^&*()-_=+\|`~{{]};:''",<.>/?' ||
PLVchr.newline_char || PLVchr.tab_char || PLVchr.space_char;
```

The standard set of delimiter characters.

```
plsql_delimiters CONSTANT VARCHAR2 (50) :=
'!@%^&*()-=+\|`~{{]};:''",<.>/?' ||
PLVchr.newline_char || PLVchr.tab_char || PLVchr.space_char;
```

The set of delimiters for the PL/SQL language; this list is a bit different from the **std_delimiters**. The underscore and pound sign characters, for example, are not delimiters in PL/SQL.

5.21.2 Wrapping long strings into paragraphs

```
PROCEDURE wrap
(text_in IN VARCHAR2,
line_length IN INTEGER,
paragraph_out IN OUT PLVtab.vc2000_table,
num_lines_out IN OUT INTEGER);
```

Wraps the string provided by **text_in** into separate lines with a maximum specified length, each line of which is stored in consecutive rows in a PL/SQL table.

```
FUNCTION wrapped_string
(text_in IN VARCHAR2,
line_length IN INTEGER := 80,
prefix_in IN VARCHAR2 := NULL)
```

```
RETURN VARCHAR2;
```

Returns a long string wrapped into a series of lines separated by newline characters. This version of wrap avoids the need to define and manipulate a PL/SQL table.

```
PROCEDURE display_wrap
(text_in IN VARCHAR2,
line_length IN INTEGER := 80,
prefix_in IN VARCHAR2 := NULL);
```

Displays the wrapped version of **`text_in`** using the **`p.l`** procedure (and DBMS_OUTPUT.PUT_LINE).

5.21.3 Analyzing string contents

```
FUNCTION next_atom_loc
(string_in IN VARCHAR2,
start_loc_in IN NUMBER,
direction_in IN NUMBER := +1,
delimiters_in IN VARCHAR2 := std_delimiters)
RETURN INTEGER;
```

Returns the location in the string of the next atomic. You provide the starting location of the scan, the direction of the scan (usually +1 or −1, but you can provide other values as well), and the delimiters to be used in the scan.

```
FUNCTION numatomics
(string_in IN VARCHAR2,
count_type_in IN VARCHAR2 := c_all,
delimiters_in IN VARCHAR2 := std_delimiters)
RETURN INTEGER;
```

Returns the number of atomics in a string, where the definition of an atomic is provided by the count type (all or word or delimiter) and the set of delimiters.

```
FUNCTION nth_atomic
(string_in IN VARCHAR2,
nth_in IN NUMBER,
count_type_in IN VARCHAR2 := c_all,
delimiters_in IN VARCHAR2 := std_delimiters)
RETURN VARCHAR2;
```

Returns the *n*th atomic in a string, where the definition of an atomic is provided by the count type (all or word or delimiter) and the set of delimiters.

```
FUNCTION numinstr
(string_in IN VARCHAR2,
substring_in IN VARCHAR2,
ignore_case_in IN VARCHAR2 := c_ignore_case)
RETURN INTEGER;
```

Returns the number of times a substring occurs in a string. You can choose to perform a search that is case−sensitive or that ignores case.

5.21.4 Parsing strings

```
PROCEDURE string
(string_in IN VARCHAR2,
atomics_list_out OUT PLVtab.vc2000_table,
num_atomics_out IN OUT NUMBER,
```

delimiters_in IN VARCHAR2 := std_delimiters);

Parses a string into atomics that are loaded into a PL/SQL table. You decide which characters will serve as the delimiters.

PROCEDURE string
(string_in IN VARCHAR2,
atomics_list_out IN OUT VARCHAR2,
num_atomics_out IN OUT NUMBER,
delimiters_in IN VARCHAR2 := std_delimiters);

Parses a string into atomics stored in a string with each atomic separated by a vertical bar. Once again, you decide which characters will serve as the delimiters.

PROCEDURE display_atomics
(string_in IN VARCHAR2,
delimiters_in IN VARCHAR2 := std_delimiters);

Displays the individual atomics in a string, as defined by the provided list of delimiters.

PREVIOUS	HOME	NEXT
5.20 PLVobj: Object Interface	BOOK INDEX	5.22 PLVprsps: PL/SQL Source Code Parsing

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

5.22 PLVprsps: PL/SQL Source Code Parsing

The PLVprsps (PL/Vision PaRSe PL/SQL) package is a more specialized string parser than PRSprs; it parses PL/SQL source code into its separate atomics. See Chapter 10 for details.

5.22.1 Package constants

The following constants are used to specify the types of tokens to be preserved when PL/SQL source code is parsed.

`c_all_tokens CONSTANT VARCHAR2(1) := 'A';`
: Specifies "all tokens".

`c_kw_tokens CONSTANT VARCHAR2(1) := 'K';`
: Specifies "keywords only".

`c_nonkw_tokens CONSTANT VARCHAR2(1) := 'N';`
: Specifies "non−keywords only" or application−specific identifiers.

`c_bi_tokens CONSTANT VARCHAR2(1) := 'B';`
: Specifies keywords that are builtins.

5.22.2 Specifying tokens of interest

`PROCEDURE keep_all;`
: Specifies that when a PL/SQL string is parsed, all tokens are to be kept (stored in a PL/SQL table).

`PROCEDURE keep_kw;`
: Specifies that when a PL/SQL string is parsed, only keywords are to be kept.

`PROCEDURE keep_nonkw;`
: Specifies that when a PL/SQL string is parsed, only non−keywords are to be kept.

`PROCEDURE keep_bi;`
: Specifies that when a PL/SQL string is parsed, only keywords that are builtins are to be kept.

`PROCEDURE nokeep_all;`
: Specifies that when a PL/SQL string is parsed, all tokens are to be ignored.

`PROCEDURE nokeep_kw;`
: Specifies that when a PL/SQL string is parsed, all keywords are to be ignored.

`PROCEDURE nokeep_nonkw;`
: Specifies that when a PL/SQL string is parsed, only non−keywords are to be ignored.

```
PROCEDURE nokeep_bi;
```

Specifies that when a PL/SQL string is parsed, only builtin keywords are to be ignored.

5.22.3 Parsing PL/SQL source code

```
PROCEDURE init_table
(tokens_out IN OUT PLVtab.vc2000_table,
num_tokens_out IN OUT INTEGER);
```

Initializes (to empty) a PL/SQL table that will hold the parsed tokens.

```
PROCEDURE plsql_string
(line_in IN VARCHAR2,
tokens_out IN OUT PLVtab.vc2000_table,
num_tokens_out IN OUT INTEGER,
in_multiline_comment_out IN OUT BOOLEAN);
```

Parses a single line of code −− a PL/SQL string −− and deposits the separate tokens in the PL/SQL table. It also passes back a Boolean flag to indicate whether, at the end of this string, the code is within a multiline comment block.

```
PROCEDURE module
(module_in IN VARCHAR2 := NULL,
tokens_out IN OUT PLVtab.vc2000_table,
num_tokens_out IN OUT INTEGER);
```

Parses all the lines of code for the specified program.

```
PROCEDURE module
(tokens_out IN OUT PLVtab.vc2000_table,
num_tokens_out IN OUT INTEGER);
```

Parses all the lines of code for the current object as specified by the PLVobj package.

PREVIOUS
5.21 PLVprs: String Parsing

HOME
BOOK INDEX

NEXT
5.23 PLVrb: Rollback Processing

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

5.23 PLVrb: Rollback Processing

The PLVrb (PL/Vision RollBack) package provides a programmatic interface to rollback activity in PL/SQL. See Chapter 20 for details.

5.23.1 Controlling rollback activity

`PROCEDURE turn_on;`

Enables rollback processing in PLVrbPLVcmt. This is *not* the default.

`PROCEDURE turn_off;`

Disables rollback processing in PLVrbPLVcmt. When this is called in the current session, the ROLLBACK statement will not be executed (the default).

`FUNCTION rolling_back RETURN BOOLEAN;`

Returns TRUE if rollback processing is being performed by PLVrbPLVcmt.

5.23.2 Logging rollback activity

`PROCEDURE log;`

Requests that whenever a ROLLBACK is performed, a message is sent to the PL/Vision log.

`PROCEDURE nolog;`

Do not log a message with the ROLLBACK.

`FUNCTION logging RETURN BOOLEAN;`

Returns TRUE if currently logging the fact that a rollback was performed by PLVrbPLVcmt.

5.23.3 Performing rollbacks

`PROCEDURE perform_rollback (context_in IN VARCHAR2 := NULL);`

Issues a ROLLBACK command.

```
PROCEDURE rollback_to
(sp_in IN VARCHAR2, context_in IN VARCHAR2 := NULL);
```

Issues a ROLLBACK to the specified savepoint.

`PROCEDURE rb_to_last (context_in IN VARCHAR2 := NULL);`

Issues a ROLLBACK to the last savepoint specified in a call to **`set_savepoint`**.

5.23.4 Managing savepoints

`PROCEDURE set_savepoint (sp_in IN VARCHAR2);`

Sets a savepoint by soft−coded string, rather than the usual hard−coded savepoint identifier. This savepoint is set to the "last savepoint" recorded by PLVrbPLVcmt.

`FUNCTION lastsp RETURN VARCHAR2;`

Returns the name of the last savepoint.

`PROCEDURE reset_savepoints;`

Clears the stack of savepoints maintained by PLVrb. This is called by PLVrbPLVcmt after a **`commit`** is performed.

PREVIOUS | HOME | NEXT
5.22 PLVprsps: PL/SQL Source Code Parsing | BOOK INDEX | 5.24 PLVstk: Stack Manager

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

5.24 PLVstk: Stack Manager

The PLVstk (PL/Vision STacK manager) package is a generic manager for both first−in−first−out (FIFO) and last−in−last−out (LIFO) stacks; it is built on PLVlst. See the companion disk for details.

5.24.1 Package constants

```
defstk CONSTANT VARCHAR2(5) := 'stack';
```

The name of the default stack.

```
lifo CONSTANT VARCHAR2(4) := 'LIFO';
```

Indicates that you are working with a last−in−first−out stack. Used in calls to **pop**.

```
fifo CONSTANT VARCHAR2(4) := 'FIFO';
```

Indicates that you are working with a first−in−first−out stack. Used in calls to **pop**.

5.24.2 Creating and destroying stacks

```
PROCEDURE make
(stack_in IN VARCHAR2 := defstk,
overwrite_in IN BOOLEAN := TRUE);
```

Allocates storage for a stack of up to 1,000 items with the specified name. By default, if the stack already exists it will be reinitialized to an empty stack.

```
PROCEDURE destroy (stack_in IN VARCHAR2 := defstk);
```

Releases all memory associated with this stack.

5.24.3 Modifying stack contents

```
PROCEDURE push
(item_in IN VARCHAR2, stack_in IN VARCHAR2 := defstk);
```

Pushes an item onto the specified stack.

```
PROCEDURE pop
(value_out IN OUT VARCHAR2,
stack_in IN VARCHAR2 := defstk,
stack_type_in IN VARCHAR2 := lifo);
```

Pops an item off the top (LIFO) or bottom (FIFO) of the stack.

5.24.4 Analyzing stack contents

```
FUNCTION nitems (stack_in IN VARCHAR2 := defstk)
RETURN INTEGER;
```

Returns the number of items currently in the stack.

```
FUNCTION itemin (stack_in IN VARCHAR2, item_in IN VARCHAR2)
RETURN BOOLEAN;
```

Returns TRUE if the specified item is found in the stack.

5.24.5 Tracing Stack Activity

```
PROCEDURE show
(stack_in IN VARCHAR2 := defstk,
show_contents_in IN BOOLEAN := FALSE);
```

Requests that pre−action status of stack be displayed for the specified stack (or all).

```
PROCEDURE noshow;
```

Turns off display of pre−action status.

```
FUNCTION showing RETURN BOOLEAN;
```

Returns TRUE if showing pre−action status.

```
PROCEDURE verify
(stack_in IN VARCHAR2 := defstk,
show_contents_in IN BOOLEAN := FALSE);
```

Requests that post−action status of stack be displayed for the specified stack (or all).

```
PROCEDURE noverify;
```

Turns off display of post−action status.

```
FUNCTION verifying RETURN BOOLEAN;
```

Returns TRUE if showing post−action status.

PREVIOUS | HOME | NEXT
5.23 PLVrb: Rollback Processing | BOOK INDEX | 5.25 PLVtab: Table Interface

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

5.25 PLVtab: Table Interface

The PLVtab (PL/Vision TABle) package makes it easier to declare, use, and display the contents of PL/SQL tables by providing predefined PL/SQL table types and programs. See Chapter 8, *PLVtab: Easy Access to PL/SQL Tables* for details.

5.25.1 Predefined table TYPEs

Since these table TYPES are already defined in the PLVtab package, you can use them to declare your own PL/SQL tables −− and not deal with the cumbersome syntax.

```
TYPE boolean_table IS TABLE OF BOOLEAN INDEX BY BINARY_INTEGER;
TYPE date_table IS TABLE OF DATE INDEX BY BINARY_INTEGER;
TYPE integer_table IS TABLE OF INTEGER INDEX BY BINARY_INTEGER;
TYPE number_table IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
TYPE vc30_table IS TABLE OF VARCHAR2(30) INDEX BY BINARY_INTEGER;
TYPE vc60_table IS TABLE OF VARCHAR2(60) INDEX BY BINARY_INTEGER;
TYPE vc80_table IS TABLE OF VARCHAR2(80) INDEX BY BINARY_INTEGER;
TYPE vc2000_table IS TABLE OF VARCHAR2(2000) INDEX BY BINARY_INTEGER;
TYPE vcmax_table IS TABLE OF VARCHAR2(32767) INDEX BY BINARY_INTEGER;
```

5.25.2 The empty PL/SQL tables

An empty PL/SQL table is a structure in which no rows have been defined. The only way to delete all the rows from a PL/SQL table is to assign an empty table to that table. You can use these predefined PL/SQL tables to accomplish this task easily.

```
empty_boolean boolean_table;
empty_date date_table;
empty_integer integer_table;
empty_number number_table;
empty_vc30 vc30_table;
empty_vc60 vc60_table;
empty_vc80 vc80_table;
empty_vc2000 vc2000_table;
empty_vcmax vcmax_table;
```

5.25.3 Toggle for showing header

`PROCEDURE showhdr;`

Requests that a header be displayed with the contents of the table (the header text is passed in the call to the **`display`** procedure). This is the default.

`PROCEDURE noshowhdr;`

Turns off the display of the header text with the table contents.

`FUNCTION showing_header RETURN BOOLEAN;`

Returns TRUE if the header is being displayed.

5.25.4 Toggle for showing row numbers

PROCEDURE showrow;
: Requests that the row number be displayed with the row contents.

PROCEDURE noshowrow;
: Turns off display of the row number (the default).

FUNCTION showing_row RETURN BOOLEAN;
: Returns TRUE if the row number is being displayed.

5.25.5 Toggle for showing blanks in row

PROCEDURE showblk;
: Requests that blank lines be displayed. A NULL row will display as the **`p.fornull`** NULL substitution value. A line consisting only of blanks will be displayed as the word BLANKS.

PROCEDURE noshowblk;
: Requests that blank lines be displayed as blank lines.

FUNCTION showing_blk RETURN BOOLEAN;
: Returns TRUE if showing the contents of blank lines.

5.25.6 Setting the row prefix

PROCEDURE set_prefix (prefix_in IN VARCHAR2 := NULL);
: Sets the character(s) used as a prefix to the text displayed before the value in the table's row (default is NULL).

FUNCTION prefix RETURN VARCHAR2;
: Returns the current prefix character(s).

5.25.7 Saving and restoring settings

PROCEDURE save;
: Saves the current settings for the toggles to private variables in the package.

PROCEDURE restore;
: Restores the settings for the toggles from the saved values.

5.25.8 Displaying a PLVtab table

```
PROCEDURE display
(tab_in IN boolean_table|date_table|number_table|integer_table,
end_in IN INTEGER,
hdr_in IN VARCHAR2 := NULL,
start_in IN INTEGER := 1,
failure_threshold_in IN INTEGER := 0,
increment_in IN INTEGER := +1);
PROCEDURE display
(tab_in IN
vc30_table|vc60_table|vc80_table|vc2000_table|vcmax_table,
end_in IN INTEGER,
hdr_in IN VARCHAR2 := NULL,
start_in IN INTEGER := 1,
failure_threshold_in IN INTEGER := 0,
increment_in IN INTEGER := +1);
```

The **`display`** procedure is overloaded nine times, for a variety of datatypes. The first version above shows the overloading for non−string datatypes. The second version shows all the different types of string PL/SQL tables supported by the **`PLVtab.display`** procedure.

PREVIOUS | HOME | NEXT
5.24 PLVstk: Stack Manager | BOOK INDEX | 5.26 PLVtkn: Token Table Interface

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

5.26 PLVtkn: Token Table Interface

The PLVtkn (PL/Vision ToKeN) package provides an interface to the **PLV_token** table; the package examines this table to determine whether an identifier is a keyword. See Chapter 10 for details.

5.26.1 Package constants

```
c_any CONSTANT CHAR(1) := '%';
c_syntax CONSTANT CHAR(1) := 'X';
c_builtin CONSTANT CHAR(1) := 'B';
c_symbol CONSTANT CHAR(1) := 'S';
c_exception CONSTANT CHAR(1) := 'E';
c_datatype CONSTANT CHAR(1) := 'D';
c_datadict CONSTANT CHAR(2) := 'DD';
c_sql CONSTANT CHAR(3) := 'SQL';
```

Each of these constants specify a different type of token or reserved word in the PL/SQL language (except for **c_any**, of course). They match the values found in the PLV_**token_type** table. They are used by various programs in the PLVtkn package.

```
c_plsql CONSTANT CHAR(1) := '%'; /* = c_builtin */
c_od2k CONSTANT CHAR(4) := 'OD2K';
c_of CONSTANT CHAR(2) := 'OF';
c_or CONSTANT CHAR(2) := 'OR';
c_og CONSTANT CHAR(2) := 'OG';
```

These constants identify different categories or sets of reserved words for the PL/SQL language. The identifier SHOW_ALERT is, for example, a builtin in Oracle Forms, but has no significance in the stored PL/SQL code. Currently PL/Vision is aware of stored PL/SQL and Oracle Forms reserved words only.

5.26.2 The keyword record TYPE

```
TYPE kw_rectype IS RECORD
(token PLV_token%token,
token_type PLV_token%token_type,
is_keyword BOOLEAN);
```

A PL/SQL record TYPE defining a record holding information about a token.

5.26.3 Determining type of token

```
FUNCTION is_keyword
(token_in IN VARCHAR2, type_in IN VARCHAR2 := c_any)
RETURN BOOLEAN;
```

General function that returns TRUE if the specified token is a keyword or a reserved word in the PL/SQL language. You can, with the **type_in** argument, ask if the token is a particular type of token. You can also call any of the following more narrowly defined functions:

```
FUNCTION is_syntax (token_in IN VARCHAR2) RETURN BOOLEAN;
FUNCTION is_builtin (token_in IN VARCHAR2) RETURN BOOLEAN;
FUNCTION is_symbol (token_in IN VARCHAR2) RETURN BOOLEAN;
FUNCTION is_datatype (token_in IN VARCHAR2) RETURN BOOLEAN;
FUNCTION is_exception (token_in IN VARCHAR2) RETURN BOOLEAN;
```

Each of these functions returns TRUE if the specified token is an identifier of the type indicated by the name of the function.

5.26.4 Retrieving keyword information

```
PROCEDURE get_keyword (token_in IN VARCHAR2, kw OUT kw_rectype);
```

Retrieves all the information about the token provided in the parameter list.

PREVIOUS | HOME | NEXT
5.25 PLVtab: Table Interface | BOOK INDEX | 5.27 PLVtmr: Program Performance Analyzer

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

5.27 PLVtmr: Program Performance Analyzer

The PLVtmr (PL/Vision TiMeR) package allows you to measure the elapsed time of PL/SQL code; it provides a programmatic layter around the GET_TIME function of Oracle's DBMS_UTILITY package. See Chapter 14, *PLVtmr: Analyzing Program Performance* for details.

5.27.1 Control execution of timings

PROCEDURE turn_on;

Enables the timing package. All PLVtmr programs will execute as requested.

PROCEDURE turn_off;

Disables PLVtmr. Calls to timing programs will be ignored.

5.27.2 Setting the repetition value

PROCEDURE set_repeats (repeats_in IN NUMBER);

Sets the number of repetitions for all timing loops implemented inside the PLVtmr package (**calibrate**, **func**, **currsucc**, **currfail**).

FUNCTION repeats RETURN NUMBER;

Returns the current repetition value.

5.27.3 Setting the factoring value

PROCEDURE set_factor (factor_in IN NUMBER);

Sets the number of iterations when calling PLVtmr in a loop. This value allows PLVtmr to show total and individual elapsed times.

FUNCTION factor RETURN NUMBER;

Returns the current number of iterations.

5.27.4 Capturing the current timestamp

PROCEDURE capture (context_in IN VARCHAR2 := NULL);

Calls **capture** to capture the current timestamp (usually the start time of an elapsed time calculation). You can provide an optional context for the timing.

5.27.5 Retrieving and displaying elapsed time

FUNCTION elapsed RETURN NUMBER;

Returns the number of hundredths of seconds elapsed since last call to **capture**.

```
FUNCTION elapsed_message
(prefix_in IN VARCHAR2 := NULL,
adjust_in IN NUMBER := 0,
reset_in IN BOOLEAN := TRUE,
reset_context_in IN VARCHAR2 := NULL)
RETURN VARCHAR2;
```

Returns a standard message format around the value returned by **`elapsed`**.

```
PROCEDURE show_elapsed
(prefix_in IN VARCHAR2 := NULL,
adjust_in IN NUMBER := 0,
reset_in IN BOOLEAN := TRUE);
```

Displays the elapsed time message returned by **`elapsed_message`**.

5.27.6 Calibration and timing scripts

PLVtmr offers a set of predefined scripts and calibration programs to test comparative performances. You might find these particular programs useful; you might simply follow their example to construct your own.

```
PROCEDURE calibrate;
```

Calculates a *base timing* -- the amount of time required to execute the NULL statement the current number of repetitions (set through **`set_repetitions`**).

```
FUNCTION base_timing RETURN NUMBER;
```

Returns the current base timing.

```
PROCEDURE func;
```

Calculates the overhead of a function call.

```
PROCEDURE cursucc;
```

Compares the performance of implicit and explicit cursors when retrieving a row successfully.

```
PROCEDURE curfail;
```

Compares the performance of implicit and explicit cursors when failing to retrieve a row.

PREVIOUS | HOME | NEXT
5.26 PLVtkn: Token Table Interface | BOOK INDEX | 5.28 PLVtrc: Trace Facility

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8*i* Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

5.28 PLVtrc: Trace Facility

The PLVtrc (PL/Vision TRaCe) package provides a generic trace facility for PL/SQL applications for use in debugging. See Chapter 21 for details.

5.28.1 Package constants

`c_top_pos CONSTANT INTEGER := 0;`

> Name of position of top module in the call stack. The argument you would pass to the **`PLVtrc.module`** function to retrieve the topmost program in the execution call stack.

`c_last_pos CONSTANT INTEGER := 2;`

> Name of position of most recent module in call stack. The argument you would pass to the **`PLVtrc.module`** function to retrieve the last program executed before the call to **`PLVtrc.module`**.

5.28.2 Controlling trace activity

`PROCEDURE turn_on;`

> Turns on the trace, enabling output from calls to the programs described below.

`PROCEDURE turn_off;`

> Turns off the trace.

`FUNCTION tracing RETURN BOOLEAN;`

> Returns TRUE if the trace is active.

5.28.3 Writing to the PL/Vision log

`PROCEDURE log;`

> Turns on logging of trace message to the PL/Vision log (see the PLVlog package), in addition to displaying the trace.

`PROCEDURE nolog;`

> Turns off logging (the default).

`FUNCTION logging RETURN BOOLEAN;`

> Returns TRUE if logging of trace messages is currently turned on.

5.28.4 Displaying current module

`PROCEDURE dispmod;`

> Turns on display of current module when showing the trace message.

```
PROCEDURE nodispmod;
```

Turns off display of current module (the default).

```
FUNCTION displaying_module RETURN BOOLEAN;
```

Returns TRUE if PLVtrc is displaying the current module.

5.28.5 Accessing the PL/SQL call stack

```
FUNCTION ps_callstack RETURN VARCHAR2;
```

Returns the string generated by a call to DBMS_UTILITY.FORMAT_CALL_STACK.

```
FUNCTION ps_module (pos_in IN INTEGER := c_last_pos)
RETURN VARCHAR2;
```

Returns the *n*th module (specified by the user) in the PL/SQL execution call stack.

5.28.6 Tracing PL/SQL code execution

```
PROCEDURE startup
(module_in IN VARCHAR2, string_in IN VARCHAR2 := NULL);
```

Executes the first line of your programs. Maintains a PLVtrc execution stack, which is then available to other PL/Vision packages.

```
PROCEDURE terminate (string_in IN VARCHAR2 := NULL);
```

Executes as the last line of your programs. Removes the current program from the PLVtrc execution stack.

```
FUNCTION currmod RETURN VARCHAR2;
```

Returns the name of the current module as set by calls to the procedure **`PLVtrc.startup`**.

```
FUNCTION prevmod RETURN VARCHAR2;
```

Returns the name of the previous module as set by calls to the procedure **`PLVtrc.terminate`**.

5.28.7 Displaying an activity trace

The **`show`** procedure is heavily overloaded for different datatypes and combinations of datatypes, along the lines of the **`p.l`** procedure. In all cases, the information you pass to **`show`** is, well, shown if you have turned on the PLVtrc facility.

```
PROCEDURE show (stg1 IN VARCHAR2);
PROCEDURE show (bool1 IN BOOLEAN);
PROCEDURE show (num1 IN NUMBER);
PROCEDURE show
(date1 IN DATE, mask_in IN VARCHAR2 := PLV.datemask);
```

Single value shows.

```
PROCEDURE show (stg1 IN VARCHAR2, num1 IN NUMBER);
PROCEDURE show (stg1 IN VARCHAR2, bool1 IN BOOLEAN);
PROCEDURE show
(stg1 IN VARCHAR2, date1 IN DATE,
mask_in IN VARCHAR2 := PLV.datemask);
```

Two−value combination shows.

```
PROCEDURE show (stg1 IN VARCHAR2, num1 IN NUMBER, num2 IN NUMBER);
PROCEDURE show
```

```
(stg1 IN VARCHAR2, num1 IN NUMBER, bool1 IN BOOLEAN);
PROCEDURE show
(stg1 IN VARCHAR2, num1 IN NUMBER, date1 IN DATE,
mask_in IN VARCHAR2 := PLV.datemask);
```

Three−value combination shows.

```
PROCEDURE action
(string_in IN VARCHAR2 := NULL,
counter_in IN INTEGER := NULL,
prefix_in IN VARCHAR2 := NULL);
```

Displays a trace of a particular action; you provide a context, numeric code, and string for the action.

5.28.8 Accessing the PLVtrc execution call stack (ECS)

```
PROCEDURE showecs;
```

Displays the PLVtrc−maintained execution call stack.

```
PROCEDURE clearecs;
```

Clears the PLVtrc execution call stack.

```
FUNCTION ecs_string RETURN VARCHAR2;
```

Returns a string which contains the current PLVtrc execution call stack (**`ecs`**), with each module separated by a newline character.

```
FUNCTION module (pos_in IN INTEGER := c_top_pos) RETURN VARCHAR2;
```

Returns the *n*th module in the PLVtrc execution call stack.

PREVIOUS | HOME | NEXT
5.27 PLVtmr: Program Performance Analyzer | BOOK INDEX | 5.29 PLVvu: Code and Error Viewing

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built−in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built−ins Pocket Reference

5.29 PLVvu: Code and Error Viewing

The PLVvu (PL/Vision View) package provides a set of programs that allow you to view both stored source code and compile errors. See Chapter 15, *PLVvu: Viewing Source Code and Compile Errors* for details.

5.29.1 Package constants

```
c_overlap INTEGER := 5;
```

The default and initial value for the number of lines to overlap around a line with a compile error.

5.29.2 Setting the overlap

```
PROCEDURE set_overlap (size_in IN INTEGER := c_overlap);
```

Call this procedure to change the size of overlap from the default described above. If you call **`set_overlap`** without any value, the size is set back to the default.

```
FUNCTION overlap RETURN INTEGER;
```

Returns the current number of overlap lines.

5.29.3 Displaying stored code

```
PROCEDURE code
(name_in IN VARCHAR2 := NULL,
start_in IN INTEGER := 1,
end_in IN INTEGER := NULL,
header_in IN VARCHAR2 := 'Code');
```

Displays some or all of the lines of source code for the specified object.

```
PROCEDURE code_after
(name_in IN VARCHAR2 := NULL,
start_with_in IN VARCHAR2,
num_lines_in IN INTEGER := overlap,
nth_in IN INTEGER := 1);
```

Displays some or all of the lines of source code for the specified object that come after the *n*th appearance of the specified string.

5.29.4 Displaying compile errors

```
PROCEDURE err
(name_in IN VARCHAR2 := NULL,
overlap_in IN INTEGER := overlap);
```

Displays the compile errors for the specified object, along with the immediately surrounding code. You can provide an override to the current number of overlap lines in the second argument.

`PROCEDURE err (name_in IN VARCHAR2, type_in IN VARCHAR2);`

Displays the compile errors for the specified object, along with the immediately surrounding code. In this version of **`err`**, you supply the program name and the program type separately (and rely on the default overlap). This is available primarily for backward compatibility to earlier versions of PLVvu.

PREVIOUS | HOME | NEXT
5.28 PLVtrc: Trace Facility | BOOK INDEX | III. Building Block Packages

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

6. PLV: Top−Level Constants and Functions

Contents:

Let's start our exploration into PL/Vision with the PLV package, which can be considered the lowest−level package or perhaps even the "miscellaneous" package. It provides a single collection point for constants and basic functions used throughout PL/Vision. Whenever I thought of something useful that was needed in more than one package but that did not belong in any particular functional package, it ended up in PLV.

The PLV (PL/Vision) package offers the following:

- *A NULL substitution value.* You can set in PLV the string you want to use in place of NULL for display purposes.
- *A product−wide date mask.* You can set a date mask that will be used throughout PL/Vision when converting and displaying dates. You can also use this mask yourself in SQL and PL/SQL.
- *A set of assertion routines.* These programs help you construct more robust applications by letting you easily verify assumptions.
- *Miscellaneous utilities.* You can obtain the current date and time, pause your PL/SQL program, obtain the error message for an Oracle error number, and more.
- *A set of constants used throughout PL/Vision.* These named constants help users of PL/Vision (and packages within PL/Vision) avoid hard−coding literals.
- *Predefined datatypes.* PL/Vision uses these datatypes as anchors (with the %TYPE declaration attribute) for declaring other variables. You might use them, too, or you might simply follow the example in your own applications.

The following sections show how to use each of the different elements in the PLV package.

6.1 Null Substitution Value

It has been my experience that when I display a variable that is NULL, I want to see some evidence of that fact. In fact, I have found that in several places in PL/Vision I want to be able to substitute a NULL value for some other string. Rather than hard code that value into my package, I added the ability in the PLV package to decide what the substitution value would be with the **`set_nullval`** procedure:

```
PROCEDURE set_nullval (nullval_in IN VARCHAR2);
```

The default value for NULL substitutions is the string NULL. You can obtain the current setting of the substitution value by calling the **`nullval`** function:

```
FUNCTION nullval RETURN VARCHAR2;
```

The **`p`** package makes use of the substitute value for NULLs. When you call **`p.l`** to display text that RTRIMs to a NULL, it will automatically display the string that you have specified for substitution. In the following example, I set the NULL substitution value to N/A and then demonstrate its use in a call to the **`p.l`** procedure.

```
SQL> exec PLV.set_nullval ('N/A');
SQL> VARIABLE v NUMBER;
SQL> exec p.l (:v);
N/A
```

PREVIOUS | HOME | NEXT

III. Building Block Packages | BOOK INDEX | 6.2 Setting the PL/Vision Date Mask

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

6.2 Setting the PL/Vision Date Mask

To standardize the way that date information is displayed inside PL/Vision, the PLV package maintains a PL/Vision date mask. This mask is used in the **p**, PLVtrc, PLVtab, and PLVlog packages to convert dates to strings.

The default date mask for PL/Vision is stored in the **c_datemask** constant and has this value:

```
FMMonth DD, YYYY HH24:MI:SS
```

The FM prefix is a toggle that requests suppression of all padded blanks and zeroes.

You can change the date mask with a call to **set_datemask**, whose header is:

```
PROCEDURE set_datemask (datemask_in IN VARCHAR2 := c_datemask)
```

Since the default value for **set_datemask** is the default date mask for PL/Vision, you can also reset the date mask to the default by calling **set_datemask** without any arguments.

You can retrieve the date mask (which is to say, you can use the date mask yourself) by calling the **datemask** function:

```
FUNCTION datemask RETURN VARCHAR2;
```

The following calls to **set_datemask** and the **datemask** function illustrate the behavior of these programs.

```
SQL> exec p.l(sysdate);
May 17, 1996 13:41:56
```

Change the date mask to show only month and year:

```
SQL> exec PLV.set_datemask ('Month YYYY');
SQL> exec p.l(sysdate);
May       1996
```

Change the date mask to suppress those extra spaces:

```
SQL> exec PLV.set_datemask ('fmMonth YYYY');
SQL> exec p.l(sysdate);
May 1996
```

Now return the date mask back to the default:

```
SQL> exec PLV.set_datemask
SQL> exec p.l(sysdate);
May 17, 1996 13:42:37
```

The following query uses the **`datemask`** function inside SQL to view the date and time of stored information (the **`PLV`** package makes this function accessible in SQL by including a RESTRICT_REFERENCES pragma):

```
SQL> SELECT TO_CHAR (hiredate, PLV.datemask)
  2     FROM emp
  3    WHERE deptno = 10;

TO_CHAR(HIREDATE,PLV.DATEMASK)
--------------------------------------------------
November 17, 1981 09:18:44
June 9, 1981 11:11:32
January 23, 1982 17:01:00
```

PREVIOUS HOME NEXT
6.1 Null Substitution Value BOOK INDEX 6.3 Assertion Routines

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

6.3 Assertion Routines

PL/Vision provides a set of generic routines you can use in your own programs to assert the validity of your program's assumptions. Just about every piece of software you write makes assumptions about the data it manipulates. For example, parameters may have only certain values or be within a certain range; a string value should have a certain format; an underlying data structure is assumed to have been created. It's fine to have such rules and assumptions, but it is also very important to verify or "assert" that none of the rules is being violated.

The cleanest way to perform this task is to call a prebuilt assertion routine (see *Chapter 20* in *Oracle PL/SQL Programming*). The PLV package offers a variety of procedures to allow you to validate assumptions in the most natural possible manner. In all cases, if the assumption is violated the assertion program will take up to two actions:

1. *Display a message if provided.* This string is optional and the default for the string is NULL.

2. *Raise the* **assertion_failure** *exception.* You can then handle this exception in the program that called the assertion routine, or you can let the exception terminate that program and propagate to the enclosing block.

The PLV assertion routines come in the following flavors:

Procedure Name	Description
assert	Generic assertion routine. You pass it a Boolean expression or value and assert tests to see if that expression is TRUE.
assert_inrange	Generic assertion routine to check date and numeric ranges. You provide the value to be checked along with start and end values. If the value (either date or number) does not fall within the specified range, **assert_inrange**
assert_notnull	Generic assertion routine to check that the specified value is NOT NULL.

The **assert** procedure is the most generic of the assertion routines. It is called, in fact, by the other assertion routines. The header for **assert** is as follows:

```
PROCEDURE assert
   (bool_in IN BOOLEAN, stg_in IN VARCHAR2 := NULL);
```

6.3.1 Using the assert Procedure

Let's take a look at how to incorporate assertion routines in your code, and then examine the impact. The following procedure translates a code into a description string. There are only three valid codes, an assumption that is validated by the call to **PLV.assert**:

```
CREATE OR REPLACE FUNCTION calltype (code_in IN VARCHAR2)
   RETURN VARCHAR2
IS
   retval VARCHAR2(100) := NULL;
BEGIN
   PLV.assert
      (code_in IN ('E', 'C', 'I'), 'Enter E C or I...');
   IF code_in = 'E'
   THEN retval := 'EMERGENCY';
   ELSIF code_in = 'C'
   THEN retval := 'COMPLAINT';
   ELSIF code_in = 'I'
   THEN retval := 'INFORMATION';
   END IF;
   RETURN retval;
END calltype;
/
```

Notice that I pass a complex Boolean expression as an argument to the assert routine. This may seem odd at first glance, but you will get used to it quickly. A program's Boolean argument can be a literal, a variable, or an expression.

Now we will try using the **`calltype`** function by embedding it in calls to the **`p.l`** procedure so that we can see the results.

In the first call to **`calltype`** below, I pass a valid code and **`p.l`** displays the correct returned description. In the second call to **`calltype`**, I pass an invalid code, J. As a result, the assertion routine displays the message as specified in the function and then raises the exception, which goes unhandled.

```
SQL> exec p.l(calltype('E'));
EMERGENCY
SQL> exec p.l(calltype('J'));
Enter E C or I...
ERROR at line 1:
ORA−06510: PL/SQL: unhandled user−defined exception
```

The error is displayed as a "user−defined exception" because **`PLV.assert`** raised the **`assertion_failure`** exception, which is not a system exception. You can trap for that exception as shown below:

```
BEGIN
   p.l (calltype ('J'));
EXCEPTION
   WHEN PLV.assertion_failure
   THEN
      p.l ('Invalid call type code');
END;
```

6.3.2 Asserting NOT NULL

The other assertion routines are designed to handle specific kinds of assertions that programmers must commonly handle. The **`assert_notnull`** routine, for example, allows you to easily make sure that an argument to a program is NOT NULL.

Without an assertion routine, you will write variations of code like this over and over again in your programs:

```
IF code_in IS NULL
THEN
   p.l ('The code cannot be null!');
   RAISE VALUE_ERROR;
ELSE
```

```
      act_on_code (code_in);
   END IF;
```

With **`PLV.assert_notnull`**, you simply attempt to assert the rule. If the code "passes," you move on to your action:

```
PLV.assert_notnull (code_in);
act_on_code (code_in);
```

You save on the typing and your indentation flattens out, thereby improving the readability of your program.

PLV offers four overloadings of **`assert_notnull`**, so you can pass it Booleans, strings, dates, and numbers.

6.3.3 Asserting "In Range"

The range assertion routines will probably save you the most code and provide a higher level of coverage of problem data. PLV offers two overloaded **`assert_inrange`** programs: one for dates and one for numbers.

The date range assertion routine header is:

```
PROCEDURE assert_inrange
   (val_in IN DATE,
    start_in IN DATE := SYSDATE,
    end_in IN DATE := SYSDATE+1,
    stg_in IN VARCHAR2 := NULL,
    truncate_in IN BOOLEAN := TRUE);
```

The first three arguments should be clear: You provide the value you want to check, as well as the start and end dates. Notice that the default start date is SYSDATE or "now" (at midnight) and the default end date is SYSDATE+1 or "tomorrow" (at midnight). The fourth argument, **`stg_in`**, is the optional string for display.

The fifth parameter, **`truncate_in`**, allows you to specify whether or not you want the end−point dates to be truncated. When a date is truncated (with the default mask, which is your only option in **`assert_inrange`**), the time portion is removed. The default setting for this argument is to perform truncation. I offer this option because in many cases when developers want to perform date range checks, they do not want to have to deal with the time component. That aspect of a date variable can, in fact, cause "obviously correct" dates to fail.

The default values of **`assert_inrange`** for dates is designed to allow you to assert with a minimum of typing that a date falls on the current day. Consider this call to the assertion program:

```
IF PLV.assert_inrange (v_hiredate)
THEN
   ...
```

If no other arguments are specified, then **`PLV`** checks to see if

```
v_hiredate BETWEEN TRUNC (SYSDATE) AND TRUNC (SYSDATE+1)
```

which, given the way TRUNC works, asks: "Is **`hiredate`** between midnight of last night and midnight of the coming night?" In other words, does **`v_hiredate`** fall anywhere during the current day?

The numeric **`assert_inrange`** is more straightforward. As you can see from the header below, there is no truncate argument. It simply checks to see if the specified number falls within the specified range.

```
PROCEDURE assert_inrange
   (val_in IN NUMBER,
    start_in IN NUMBER,
    end_in IN NUMBER,
    stg_in IN VARCHAR2 := NULL);
```

The following procedure updates the salary of an employee, but only if the new salary does not exceed the maximum salary allowed in the system (returned by the personnel package **`max_salary`** function):

```
PROCEDURE update_salary
   (emp_in IN emp.empno%TYPE, newsal_in IN NUMBER)
BEGIN
   PLV.assert_inrange (newsal_in, 0, personnel.max_salary);
   UPDATE emp
      SET sal = newsal_in
    WHERE empid = emp_in;
END;
```

If you are careful and consistent in your use of assertion programs like those in the PLV package, your programs will be more robust and less likely to fail in unpredictable ways.

PREVIOUS | HOME | NEXT
6.2 Setting the PL/Vision Date Mask | BOOK INDEX | 6.4 PLV Utilities

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

6.4 PLV Utilities

PL/Vision comes with a set of utility procedures and functions. These programs offer shortcuts to executing commonly needed operations or information in PL/SQL programs. In some cases, the utility exists simply to make it possible to access the information from within a SQL statement. These programs are described below.

6.4.1 Converting Boolean to String

The **boolstg** function translates a Boolean expression into a string describing that Boolean's value. The header for **boolstg** is:

```
FUNCTION boolstg
   (bool_in IN BOOLEAN, stg_in IN VARCHAR2 := NULL)
RETURN VARCHAR2;
```

The second argument allows you to pass a string that is prefixed to the string describing the Boolean (TRUE, FALSE, or NULL). The various ways to call **PLV.boolstg** are illustrated below:

```
SQL> exec p.l(PLV.boolstg (TRUE));
TRUE
SQL> exec p.l(PLV.boolstg (TRUE, 'This is'));
This is TRUE
```

6.4.2 Obtaining the Error Message

The **errm** function provides a PL/SQL function interface to the SQLERRM builtin function. You cannot call SQLERRM in a SQL statement, which is annoying when you have error information in a SQL database table and you want to display the corresponding error message text. You want to do something like this:

```
SELECT errcode, SQLERRM (errcode)
  FROM error_log
 WHERE create_ts < SYSDATE;
```

but the SQL layer returns this error message:

```
ORA−00904: invalid column name
```

The **errm** function allows you to use SQLERRM inside SQL by hiding that builtin behind the function interface and by using the RESTRICT_REFERENCES pragma in the specification. With PLV, you change that SQL statement to:

```
SELECT errcode, PLV.errm (errcode)
  FROM error_log
 WHERE create_ts < SYSDATE;
```

and get the information you need to analyze and fix your problems.

6.4.3 Retrieving Date and Time

The **now** function is simply a quick way to display the current date and time. Its header is:

```
FUNCTION now RETURN VARCHAR2;
```

I built **PLV.now** because I got tired of typing:

```
SELECT TO_CHAR (SYSDATE, 'HH:MI:SS') FROM dual;
```

just to see the current time. With **PLV.now**, you can at any point see both the date and time from within SQL*Plus with either of these commands:

```
------------------------------------------
SQL> SELECT PLV.now from DUAL;
------------------------------------------

August 3, 1996 20:19:35

SQL> exec p.l(PLV.now);
August 3, 1996 20:20:48
```

6.4.4 Pausing Your Program

The **pause** procedure of the PLV package provides a cover for the DBMS_LOCK.SLEEP procedure; its header is:

```
PROCEDURE pause (seconds_in IN INTEGER);
```

Why bother providing this **pause** program, when it is nothing more than a call to the builtin SLEEP procedure? Most PL/SQL developers will never use the DBMS_LOCK package; few of us need to create and manipulate locks with the Oracle Lock Management services. Yet this package contains SLEEP because it is the context in which Oracle developers realized they needed this capability.

The **PLV.pause** procedure offers, at least within PL/Vision, a more rational location for this technology.

The following "infinite" loop uses **PLV.pause** to make sure that there is an hour's delay between each retrieval of data from a DBMS_PIPE named **hourly_production**.

```
LOOP
   process_line_data ('hourly_production');
   PLV.pause (60 * 60);
END LOOP;
```

PREVIOUS | HOME | NEXT
6.3 Assertion Routines | BOOK INDEX | 6.5 The Predefined Datatypes

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

6.5 The Predefined Datatypes

The PLV package provides several centrally located, predefined datatypes. These elements are used throughout PL/Vision, but you can make use of them as well.

The two variables that are to be used as predefined datatypes are the following:

```
plsql_identifier VARCHAR2(100) := 'IRRELEVANT';
max_varchar2 VARCHAR2(32767) := 'IRRELEVANT';
```

Use the **`plsql_identifier`** variable whenever you need to declare a VARCHAR2 variable or constant that holds a PL/SQL identifier, such as a table name or column name or program name. Currently these names are limited to 30 characters. That may, however, change in the future and you will find errors popping up in your utilities if you declare variables like this:

```
v_table_name VARCHAR2(30);
```

Instead, use the predefined datatype as follows:

```
v_table_name PLV.plsql_identifier%TYPE;
```

Use the **`max_varchar2`** variable whenever you need to declare a string variable to the maximum number of characters allowable in PL/SQL. Again, today that maximum size is 32,767, but this value may increase in the future. By relying on **`max_varchar2`** in your declarations and parameter definitions, you (or the supplier of PL/Vision) can change the definition in one place and, with a compile, upgrade all your code.

> ***NOTE:*** Do you notice any conflict between the declarations of these predefined datatypes and the best practices I described earlier in this book? I have declared variables in my package specification; the best practice recommends *strongly* that you always hide your data structures in the package body. At the very least, you might be thinking, I should make the **`plsql_identifier`** and **`max_varchar2`** data structures constants, so their values *cannot* be changed.
>
> Well, take a look at the default value for these variables. It doesn't matter *what* value is assigned to these variables. They are only to be used in %TYPE anchored declarations to pass on the datatype and constraint. And I couldn't make them CONSTANTs even if I wanted to (I tried that on the first pass). It turns out that you cannot anchor variables to constants; if you want to use %TYPE, you must remove the CONSTANT keyword.

6.6 The Predefined Constants

In addition to the predefined datatypes, the PLV package offers a set of constants that are used throughout PL/Vision. These constants define the type of input or output repository you want to use in a given situation. These constants are used in PLVio, PLVlog, and PLVexc. The input/output options in PL/Vision are shown in the following table.

Description	Constant
Database table	**dbtab**
PL/SQL table	**pstab**
Operating system file	**file**
VARCHAR2 string	**string**
Standard output	**stdout**

For example, if you want to define the target repository of the PLVio package to be a PL/SQL table, you will issue this command:

```
SQL> exec PLVio.settrg (PLV.pstab);
```

If you want PLVlog to write its information to an operating system file, you will execute this command:

```
SQL> exec PLVlog.tofile;
```

which in turn sets the **v_log_type** variable to the **PLV.file** constant.

Special Notes on PLV

Once you build a package like this, you will find yourself constantly adding to it. Be careful not to throw in constants and programs that really do have a place in another, less generic package.

The **boolstg**, **errm**, and **now** functions of PLV can all be used from within SQL statements, since the package specification for PLV includes calls to the RESTRICT_REFERENCES pragma.

The **errm** function is a good example of how you can use a layer of your own PL/SQL code around native PL/SQL builtins (SQLERRM) to make the information more widely accessible (i.e., callable from within SQL statements).

7. p: A Powerful Substitute for DBMS_OUTPUT

Contents:

The **p** (PL/Vision Put) package is one of the first and simplest packages I ever built. It is also one of my favorites. The concept is clear, the payback immediate and everlasting. It also demonstrates some of the key advantages of PL/SQL packages in general.

The **p** package offers a powerful, flexible substitute for the DBMS_OUTPUT.PUT_LINE builtin package (see sidebar for a quick review of DBMS_OUTPUT). This is the **l** procedure. Generally, you will use the **l** procedure of the **p** package in place of DBMS_OUTPUT.PUT_LINE to display output from within a PL/SQL program. The **p** package improves your development productivity by minimizing keystrokes (as I described in Chapter 1, *PL/SQL Packages*, I grew to detest those 20 characters and sought out names for the package and procedure that would involve the smallest amount of typing possible), but its advantages go beyond this superficial benefit.

The builtin DBMS_OUTPUT.PUT_LINE procedure, particularly as it is supported within the SQL*Plus environment, has the following complications:

- If you pass it a string that is longer than 255 bytes, the PL/SQL runtime engine raises the VALUE_ERROR exception.
- If you try to display a NULL value, PUT_LINE simply ignores your request. Not even a blank line is displayed.
- All leading blanks are trimmed from the string when displayed.[1]

  [1] In SQL*Plus 3.3, you can issue the command **`set serveroutput on format wrapped`**, and leading blanks are preserved and long lines are wrapped to the length specified in SQL*Plus. However, you still can't display more than 255 characters. [undocumented feature reported by Laurence Pit]
- PUT_LINE's overloading is quite limited. It cannot display a Boolean value, nor can it handle combinations of data.

I created the **p** package to compensate for these deficiencies −− and then I put it to use. In fact, you will find only one package in PL/Vision in which DBMS_OUTPUT.PUT_LINE is called: the **p** package. I have been very careful to always use my own substitution program for this builtin, to make sure that the robust features of **p.l** are leveraged throughout the library.

The following sections of this chapter show you how to use **p.l** to display information. They also show you how to modify the behavior of **p.l** as follows:

- Set the line prefix. This prefix allows you to preserve leading spaces.
-

Set the line separator. This character gives you a way to preserve white space (blank lines) in your source.

- Control **p.l** output. The show override argument in the **p.l** procedure gives you some added flexibility over when **p.l** will actually show you something.

7.1 Using the l Procedure

The **p.l** procedure is a pleasure to use. When you call **p.l** instead of the DBMS_OUTPUT.PUT_LINE procedure, you will never have to worry about raising the VALUE_ERROR exception. You can display values of up to 32,767 bytes! You can pass many different kinds of data to the **l** procedure and it will figure out what to do and how to best display the information.

What, you worry? No, you let the package do the worrying for you. You get to concentrate on building your application.

You use **p.l** just as you would its builtin cousin, except that the **p** package offers a much wider overloading for different types and combinations of types of data. You pass it one or more values for display purposes. You can also use the final argument of the **p.l** procedure to control when output should be displayed (see Section 7.4, "Controlling Output from p").

Here are the headers for the version of **p.l** that display a number and a string−date combination, respectively.

```
PROCEDURE l (number_in IN NUMBER, show_in IN BOOLEAN := FALSE);

PROCEDURE l
   (char_in IN VARCHAR2, date_in IN DATE,
    mask_in IN VARCHAR2 := PLV.datemask,
    show_in IN BOOLEAN := FALSE);
```

To view the salary of an employee, you simply execute:

```
p.l (emp_rec.sal);
```

To view the employee name and hire date, you execute:

```
p.l (emp_rec.ename, emp_rec.hiredate)
```

and you will see this data in this format:

```
JONES: May 12, 1981 22:45:47
```

To get the same information using the default functionality of DBMS_OUTPUT, you would have to enter something as ugly and time−consuming as this:

```
DBMS_OUTPUT.PUT_LINE (emp_rec.ename || ': ' ||
   TO_CHAR (emp_rec.hiredate, 'FMMonth DD, YYYY HH:MI:SS'))
```

Which would you rather type? That should give you a good sense of the potential productivity gains available through **p**![2]

[2] Did you ever notice how old the data in the **emp** table is? Oracle Corporation should update that demonstration table to reflect corporate growth and increased salaries...but I guess they have to worry about backward compatibility of demonstration scripts!

7.1.1 Valid Data Combinations for p.l

Table 7.1 shows the different types of data that can be passed to the **p.l** procedure.

See the **p** package specification (or the table in Chapter 5, *PL/Vision Package Specifications*) for the headers of all the corresponding versions of the **l** procedure.

Table 7.1: Valid Data Combinations for p.l

Data Combinations	Resulting Value
VARCHAR2	The string as supplied by the user.
DATE	The date converted to a string, using the specified date mask. The default date mask is provided by **PLV.datemask** −− a PL/Vision−wide setting.
NUMBER	The number converted to a string using the default format mask.
BOOLEAN	The string "TRUE" if the Boolean expression evaluates to TRUE, "FALSE" if FALSE, and the NULL substitution value if NULL.
VARCHAR2, DATE	The string concatenated to a colon, concatenated to the date (converted to a string as explained above).
VARCHAR2, NUMBER	The string concatenated to a colon, concatenated to the number (converted to a string as explained above).
VARCHAR2, BOOLEAN	The string concatenated to a colon, concatenated to the Boolean (converted to a string as explained above).

7.1.2 Displaying Dates

When you display a date using **p.l**, it uses the string returned by the **PLV.datemask** function as the format mask. The default value of the format mask is:

The DBMS_OUTPUT Package

The DBMS_OUTPUT package allows you to display information to your session's output device from within your PL/SQL program. As such, it serves as just about the only easily accessible means of debugging your PL/SQL Version 2 programs. DBMS_OUTPUT is also the package you will use to generate reports from PL/SQL scripts run in SQL*Plus.

Theoretically, you write information to the DBMS_OUTPUT buffer with calls to PUT_LINE and PUT and then extract that information for display with the GET_LINE program. In reality (in SQL*Plus, anyway), you simply call the PUT_LINE program from within your PL/SQL program and when your program finishes executing, all the text "put" to the buffer is displayed on your screen. The following SQL*Plus session gives you an idea of what you must type:

```
SQL> exec DBMS_OUTPUT.PUT_LINE ('this is great!');
this is great
```

The size of the DBMS_OUTPUT buffer can be set to a size between 2,000 bytes (the default) and 1,000,000 bytes with the ENABLE procedure. If you do not ENABLE the package, then no information will be displayed or will be retrievable from the buffer.

When using DBMS_OUTPUT in SQL*Plus, you can use the SET command to enable output from DBMS_OUTPUT and also set the size of the buffer. To enable output, you must issue this command:

```
SQL> set serveroutput on
```

To set the buffer size to a value other than 2,000, add the size clause as follows:

```
SQL> set serveroutput on size 1000000
```

I recommend that you put the SET SERVEROUTPUT command in your **login.sql** script so your session is automatically enabled for output. Remember, however, that every time you reconnect inside SQL*Plus, all of your package variables are reinitialized. So if you issue a CONNECT command in SQL*Plus, you will need to reenable DBMS_OUTPUT. The script **ssoo.sql** (on the disk) does this for you with a minimum of fuss. To enable output and set the buffer to its maximize size (1 megabyte), simply type:

```
SQL> @ssoo
```

See *Chapter 15* of *Oracle PL/SQL Programming*, for more details on DBMS_OUTPUT.

```
'FMMonth DD, YYYY FMHH24:MI:SS'
```

If you would like to change the format used to display dates, you can either specify a new format when you call **p.l** or you can change the default mask maintained by the PLV package.

To specify a different format in the call to **p.l**, simply include the mask string after the date argument. Here, for example, is the header for the version of **p.l** that displays a date:

```
PROCEDURE l
   (date_in IN DATE,
    mask_in IN VARCHAR2 := PLV.datemask,
    show_in IN BOOLEAN := FALSE);
```

So to display the name of an employee and the month/year she was hired, I can use:

```
p.l (emp_rec.ename, emp_rec.hiredate, 'Month YYYY');
```

Alternatively, I can set the default format for any date displayed in PLV with this call:

```
PLV.set_datemask ('Month YYYY');
```

and then the call to **p.l** could be simplified to:

```
p.l (emp_rec.ename, emp_rec.hiredate);
```

PREVIOUS	HOME	NEXT
6.6 The Predefined Constants	BOOK INDEX	7.2 The Line Separator

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

7.2 The Line Separator

When you compile and store code in SQL*Plus, any blank lines in your source code are discarded. This annoying "undocumented feature" wreaks havoc at compile time. If there are any compile errors, the line number stored in USER_ERRORS and returned by SHOW ERRORS almost never matches the line number of the code in your operating system file. What an annoyance!

The situation gets even worse when you really want to preserve those blank lines for readability. The **`PLVhlp`** package, for example, provides an architecture for online help. Without blank lines in the help text, it would be very difficult to make this text understandable. It would be awfully nice to be able to preserve those blank lines −− or at least make a line *appear* to be blank when displayed.

The **`p`** package recognizes this need and allows you to specify a line separator character. If a line of text passed to **`p.l`** consists only of the line separator character, it is displayed as a blank line.

You set the line separator with the **`set_linesep`** procedure, whose header is:

```
PROCEDURE set_linesep (linesep_in IN VARCHAR2);
```

You can retrieve the current line separator character with the **`linesep`** function:

```
FUNCTION linesep RETURN VARCHAR2;
```

The default line separator character is =. As a result, if I execute the following anonymous block,

```
BEGIN
   p.l ('this');
   p.l ('=');
   p.l ('is');
   p.l ('=');
   p.l ('separated!');
END;
/
```

I see this output:

```
this

is

separated!
```

7.3 The Output Prefix

PL/Vision works extensively with stored PL/SQL code −− which often has (and should have) lots of indentation to reveal the logical flow of the program. If I use the native, builtin PUT_LINE procedure to display this text, it comes out left−justified; all leading spaces are automatically trimmed by the DBMS_OUTPUT.PUT_LINE builtin. This is not a very useful way to display code.

The **p** package handles this situation by prefixing all text with a prefix string. As long as the prefix string is not composed entirely of spaces, the leading spaces in your own text will be preserved.

You can set and retrieve the value of the prefix string with these two programs:

```
PROCEDURE set_prefix (prefix_in IN VARCHAR2 := c_prefix);
FUNCTON prefix RETURN VARCHAR2;
```

The default prefix (stored in the package constant **c_prefix**) is CHR(8), which is the backspace character. This character, like many of the other nonprinting characters in the ASCII code table, displays as a black box in the Windows environment and functions well as a prefix.

You may wish to substitute a different value more appropriate to your operating system. To do this, simply call the **set_prefix** procedure as shown in the example below:

```
SQL> exec p.set_prefix ('*');
SQL> exec p.l (SYSDATE);
*May 12, 1996 22:36:55
```

If you call **set_prefix**, but do not pass a value, you will set the prefix back to its default.

PREVIOUS | HOME | NEXT
7.2 The Line Separator | BOOK INDEX | 7.4 Controlling Output from p

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

7.4 Controlling Output from p

The **p** package offers more flexibility than does DBMS_OUTPUT in determining when output should, in fact, be displayed. With DBMS_OUTPUT, you face an all or nothing scenario. If output has been enabled, you see all information passed to PUT_LINE. If you have not (in SQL*Plus) executed the verbose SET SERVEROUTPUT ON command, nothing appears on the screen.

With **`p.l`**, you can match this functionality and then go a bit beyond it as well. The **p** package provides a toggle to determine whether calls to **`p.l`** should generate output. The programs that make up this toggle are:

```
PROCEDURE turn_on;
PROCEDURE turn_off;
```

If you call **`turn_off`** to disable output from **`p.l`**, nothing will be displayed −− unless you explicitly request that the information be shown. The last parameter of every overloading of the **`l`** procedure is the "show override". If you pass TRUE, the information will always be displayed (assuming that output from DBMS_OUTPUT has been enabled). The default value for the "show override" is FALSE, meaning "do not override."

In the following sequence of calls in SQL*Plus, I manipulate the status of output in the **p** package to demonstrate how the show override argument can be used.

```
SQL> exec p.turn_off
SQL> exec p.l (SYSDATE);
SQL> exec p.l (SYSDATE, show_in => TRUE);
*May 12, 1996 22:43:51
SQL> exec p.l (SYSDATE IS NOT NULL, show_in => TRUE);
*TRUE
SQL> exec p.turn_on
SQL> exec p.l(SYSDATE);
*May 12, 1996 22:45:47
```

The **p** package could, of course, offer much more flexibility even than this variation of all or nothing. Many developers have implemented variations on this package with numeric levels that provide a much finer granularity of choice over which statements will actually display output. Given the nearness of third−party (and Oracle−supplied) debuggers for PL/SQL, however, I exercised self−restraint and focused my efforts in the **p** package on ease of use and developer productivity.

Special Notes on p

Here are some factors to consider when working with the **p** package:

- The prefix, line separator, and NULL substitution values can be up to 10 characters in length.
-

When you **turn_on** output from the **p** package, the DBMS_OUTPUT.ENABLE procedure is called with a maximum size buffer of 1 megabyte.

- Any string that is longer than 80 characters in length will be displayed in a paragraph−wrapped format at a line length of 75 characters.

- The **p** package will only send information to standard output. If you want to send text to a database table or PL/SQL table or other repository, you might consider using PLVlog or even PLVio if the text has to do with PL/SQL source code.

- The biggest challenge in implementing the **p** package was to modularize the code inside the package body so that all those overloadings of the **l** procedure do not result in chaos. I needed to avoid redundant code so that I could easily add to the overloadings as the need arose and even add new functionality to the package's display options (such as paragraph−wrapping long text, a rather recent enhancement). I accomplished this by creating a private module, **display_line**, which is called by each of the **p.l** procedures.

- The current set of overloadings of **p.l** is really quite minimal. You might want to try your hand at enhancing PL/Vision yourself by increasing the variety of datatypes one can pass to the **l** procedure. What about two numbers or a number and a date? Give it a try!

PREVIOUS | HOME | NEXT
7.3 The Output Prefix | BOOK INDEX | 8. PLVtab: Easy Access to PL/SQL Tables

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

PREVIOUS NEXT

Chapter 8

8. PLVtab: Easy Access to PL/SQL Tables

Contents:

The PLVtab (PL/Vision TABle) package offers predefined PL/SQL table types and programs to make it easier to declare, use, and display the contents of PL/SQL tables. PL/SQL tables are the closest things to arrays in PL/SQL, but there are a number of complications. First, to use a PL/SQL table, you first have to declare a table type. Then you can declare and use the PL/SQL table itself. Beyond the definition of the PL/SQL table, the fact that it is a sparse, unbounded, homogeneous data structure can lead to complications, particularly when it comes to scanning and displaying those structures (see *Chapter 10* of *Oracle PL/SQL Programming*).

By using PLVtab, you can avoid (in most cases) having to define your own PL/SQL table types. You can also take advantage of flexible, powerful display procedures to view table contents.

When you display the contents of PL/SQL tables, PLVtab allows you to:

- Show or suppress a header for the table
- Show or suppress the row numbers for the table values
- Display a prefix before each row of the table

This chapter shows how to use the different aspects of PLVtab.

8.1 Using PLVtab−Based PL/SQL Table Types

When you use PL/SQL tables, you normally perform a number of common actions, including defining the table type, declaring the table, filling up the rows, referencing the rows, and emptying the table when done. When using a PLVtab−based table, you do not have to declare the table type. Instead you simply reference the package−based type in your declaration. PLVtab predefines the following PL/SQL table TYPEs, shown in Table 8.1:

Table 8.1: Table Types Predefined in PLVtab

Type	Description
`boolean_table`	PL/SQL table of Booleans
`date_table`	PL/SQL table of dates
`integer_table`	PL/SQL table of integers
`number_table`	PL/SQL table of numbers

`vc30_table`	PL/SQL table of VARCHAR2(30) strings
`vc60_table`	PL/SQL table of VARCHAR2(60) strings
`vc80_table`	PL/SQL table of VARCHAR2(80) strings
`vc2000_table`	PL/SQL table of VARCHAR2(2000) strings
`ident_table`	PL/SQL table of VARCHAR2(100) strings; matches **`PLV.plsql_identifier`** declaration.
`vcmax_table`	PL/SQL table of VARCHAR2(32767) strings

Let's compare the "native" and PL/Vision approaches to defining PL/SQL tables. In the following anonymous block, I define a PL/SQL table of Booleans without the assistance of **`PLVtab`**.

```
DECLARE
   TYPE bool_tabtype IS TABLE OF BOOLEAN
      INDEX BY BINARY_INTEGER;
   yesno_tab bool_tabtype;
BEGIN
```

With the **`PLVtab`** package in place, all I have to is the following:

```
DECLARE
   yesno_tab PLVtab.boolean_table;
BEGIN
```

Once you have declared a table using PLVtab, you manipulate that table as you would a table based on your own table TYPE statement.

PREVIOUS HOME NEXT

7.4 Controlling Output from p BOOK INDEX 8.2 Displaying PLVtab Tables

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

8.2 Displaying PLVtab Tables

For each type of table, PLVtab provides a **display** procedure to show the contents of the table. As a result, there are nine, overloaded versions of the **display** procedure. The headers for each of these programs are the same, except for the datatype of the first parameter (the kind of table to be displayed).

Here, for example, is the specification of the procedure to display a date table:

```
PROCEDURE display
 (tab_in IN date_table,
  end_in IN INTEGER,
  hdr_in IN VARCHAR2 := NULL,
  start_in IN INTEGER := 1,
  failure_threshold_in IN INTEGER := 0,
  increment_in IN INTEGER := +1);
```

As you can see, there are lots of parameters, and that means lots of flexibility in specifying what rows are displayed and the format of the display. Here is an explanation of the various arguments:

Argument	Description
tab_in	The PL/SQL table you want to display. The table type must be one of those predefined in PLVtab.
end_in	The last row you want displayed. This is required. Until PL/SQL Release 2.3 there is no way for PLVtab to know the total number of rows defined in the table. As you will see below, you can also specify the starting row, which defaults to 1.
hdr_in	The header you want displayed before the individual rows are written out using the **p.l** procedure.
start_in	The first row you want displayed. The default value is 1. This is placed after the **end_in** argument because in almost every case it will not need to be specified.
failure_threshold_in	The number of times the display program can reference an undefined row in the table before it stops trying any more. Remember: PL/SQL tables are sparse. Consecutive rows do not need to be defined, but the display program does need to move sequentially through the table to display its rows.
increment_in	The increment used to move from the current row to the next row. The default value is 1, but you could ask display to show every fifth row by passing a value of 5.

The following examples illustrate how the different arguments are used.

8.2.1 Displaying Wrapped Text

The **`display_wrap`** program of the PLVprs package takes advantage of the PLVtab package in several ways. It declares and uses a VARCHAR2(2000) table to receive the output from the **`wrap`** procedure, which wraps a long string into multiple lines, each line of which is stored in a row in the PL/SQL table. This table is then displayed with a call to the **`display`** procedure. Notice that **`display_wrap`** also turns off the PLVtab header and sets the prefix before performing the display. These toggles for PLVtab are discussed in the next section.

```
PROCEDURE display_wrap
  (text_in IN VARCHAR2,
   line_length IN INTEGER := 80,
   prefix_in IN VARCHAR2 := NULL)
IS
  lines PLVtab.vc2000_table;
  line_count INTEGER := 0;
BEGIN
  PLVtab.noshow_header;
  PLVtab.set_prefix (prefix_in);
  wrap (text_in, line_length, lines, line_count);
  PLVtab.display (lines, line_count);
END;
```

Notice that in this call to **`display`** I employ most of the defaults: a NULL header, a starting row of 1, a failure threshold of 0 (all rows should be defined), and an increment of 1. I do not want a header since I am essentially using **`display`** as a utility within another program.

8.2.2 Displaying Selected Companies

Suppose that I have populated a PL/SQL table with company names, where the row number is the primary key or company ID. I am, therefore, not filling the PL/SQL table sequentially. By keeping track of the lowest and highest row used in the table, however, I can still display all the defined rows in the PL/SQL table as shown below.

First, the package containing the data structures associated with the list of company names:

```
PACKAGE comp_names
IS
   /* The table of names. */
   list PLVtab.vc80_table;
   /* The lowest row number used. */
   lo_row BINARY_INTEGER := NULL;
   /* The highest row number used. */
   hi_row BINARY_INTEGER := NULL;
END comp_names;
```

Then various programs have been called to fill up the PL/SQL table with any number of company names. The following call to display will show all defined rows regardless of how many there are, and how many undefined rows lie between company names.

```
PLVtab.display
  (comp_names.list,
   comp_names.hi_row,
   'Selected Company Names',
   comp_names.lo_row,
   comp_names.hi_row − comp_names.lo_row);
```

Let's look at a concrete example. Row 1506 is assigned the value of ACME, while row 20200 contains the company name ArtForms. I can then make the above call to **`PLVtab.display`** and get the following results displayed on the screen:

```
Selected Company Names
ACME
ArtForms
```

You will probably be surprised to hear that it took more than 83 seconds on my Pentium 90Mhz laptop to produce these results. Why so long a delay? The **`display`** procedure displayed row 1506 and then attempted unsuccessfully 18,693 times to retrieve the rows between 1506 and 20200. Each time **`display`** referenced an undefined row, the PL/SQL runtime engine raised the NO_DATA_FOUND exception, which was ignored.

The conclusion you should draw from this example is that **`PLVtab.display`** does a great job of hiding these kinds of details, but it is still important for you to understand the architecture of PL/SQL tables. This understanding will help you explain what would otherwise be an absurdly slow response time −− and also help you decide when to take advantage of the **`PLVtab.display`** procedure. If your defined rows are dispersed widely, **`PLVtab.display`** may not be efficient enough a method to display the contents of your table.

PREVIOUS | HOME | NEXT
8.1 Using PLVtab−Based PL/SQL Table Types | BOOK INDEX | 8.3 Showing Header Toggle

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8*i* Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

8.3 Showing Header Toggle

The PLVtab package is designed to be a generic, low−level utility for working with PL/SQL tables. As such, it needs to be as flexible as possible when it comes to displaying the contents of these tables. I found that I wanted to use **`PLVtab.display`** both to:

1. Dump the contents of a table for debugging and verification purposes; and
2. Display table contents from within other PL/Vision utilities and packages.

In the first use of **`PLVtab.display`**, I could rely on the default header for the table, which is simply:

```
Contents of Table
```

since I just wanted to see the results. When I am using PLVtab from within another environment or utility, I need to be able to carefully control the format of the output. In some cases I will want to provide an alternative header, which is done through the parameter list of the display procedure. In other situations, I may want to avoid a header altogether.

The "show header" toggle offers this level of flexibility. The default/initial setting for PLVtab is to display a header with the table. You can turn off the toggle by executing the "no show" procedure as follows:

```
SQL> exec PLVtab.noshowhdr
```

In this mode, even if you provide an explicit header in your call to display, that information will be ignored. Only the row information will be displayed.

8.4 Showing Row Number Toggle

The "show row" toggle allows you to specify whether or not you want the row numbers to be displayed along with the contents of the row. The default value for this setting is no row numbers. Turn on the display of row numbers as follows:

```
SQL> exec PLVtab.showrow
```

and then when you display the contents of a table, you will see Row N = prefixed to each row value as shown in this output from **`PLVtab.display`**:

```
Row 1505 = ACME
Row 20200 = ArtForms
```

8.5 Setting the Display Prefix

The PLVtab **`set_prefix`** procedure allows you to specify a prefix that is to be displayed before the row values. This prefix is only displayed when you are *not* showing the row numbers. The default value for the prefix is NULL, which means that you don't see any prefix unless you call the **`set_prefix`** program. The header for this procedure is:

```
PROCEDURE set_prefix (prefix_in IN VARCHAR2 := NULL)
```

Since the single argument has a default value of NULL, you can set the prefix back to its default value simply by entering this command:

```
SQL> PVLtab.set_prefix;
```

The following script shows you how the prefix is set and used in the display of PLVtab table information.

```
DECLARE
   nms PLVtab.vc80_table;
   lo INTEGER;
   hi INTEGER;
BEGIN
   PLVtab.noshowrow;
   PLVtab.set_prefix ('Company Name = ');
   nms (1505) := 'ACME';
   nms (20200) := 'ArtForms';
   lo := 1505;
   hi := 20200;
   PLVtab.display (nms, hi, 'Selected Company Names', lo, hi-lo);
END;
/
Selected Company Names
Company Name = ACME
Company Name = ArtForms
```

8.6 Emptying Tables with PLVtab

For PL/SQL Releases 2.2 and earlier, the only way to delete all rows from a PL/SQL table (and release all associated memory) is to assign an empty table of the same TYPE to your structure. PLVtab offers the following set of empty tables to facilitate this process for PLVtab−based tables:

```
empty_boolean boolean_table;
empty_date date_table;
empty_integer integer_table;
empty_number number_table;

empty_vc30 vc30_table;
empty_vc60 vc60_table;
empty_vc80 vc80_table;
empty_vc2000 vc2000_table;
empty_vcmax vcmax_table;
empty_ident ident_table;
```

It is very easy to use these empty tables (of course, they are only empty if you do not define rows in those PL/SQL tables!). The following example shows a package body that has defined within it a PL/SQL table. This table is then modified and emptied by the program units defined in that same package body.

```
PACKAGE BODY paid_subs
IS
   listcount INTEGER := 0;
   namelist PLVtab.vc80_table;

   PROCEDURE addsub (name_in IN VARCHAR2) IS
   BEGIN
      namelist (listcount + 1) := name_in;
      listcount := listcount + 1;
   END;

   PROCEDURE clearlist IS
   BEGIN
      namelist := PLVtab.empty_vc80;
   END;
END paid_subs;
```

If you have PL/SQL Release 2.3, you don't have to bother with these empty tables. Instead, you can use the PL/SQL table DELETE attribute to remove the rows from the table. The following examples illustrate the power and flexibility of this syntax:

```
namelist.DELETE; -- Delete all rows.
namelist.DELETE (5); -- Delete row 5.
namelist.DELETE (5, 677); -- Delete all rows between 5 and 677.
```

This is obviously a much more desirable technique −− and it highlights a drawback to the PLVtab approach to emptying tables.

8.6.1 Improving the Delete Process

As explained above, to delete all the rows from a (PL/SQL Release 2.2 and earlier) PLVtab table, you would assign an empty table to that table. The problem with this approach is that it exposes the implementation of the delete process. You have to know about the empty table and also the aggregate assignment syntax. Worse, when you do upgrade to PL/SQL Release 2.3 or above, you have to go to each of these assignments and change the code in order to take advantage of the new attribute.

A much better approach would be for PLVtab to provide not the empty tables themselves, but procedures that do the emptying for you. Such a program is very simple and is shown below:

```
PROCEDURE empty (table_out OUT date_table) IS
BEGIN
   table_out := empty_date;
END;
```

This procedure would, of course, have to be overloaded for each table TYPE. Notice that this program uses the empty table just as you would, but that detail is hidden from view. There are two advantages to this approach:

- Now when I want to empty a table, I simply call the program as shown below:

  ```
  PLVtab.empty (my_table);
  ```

  I don't have to know about the empty tables and their naming conventions. I leave that to the package.

- When my installation upgrades to PL/SQL Release 2.3, I can take immediate advantage of the DELETE operator without changing those parts of my application that empty my tables. Instead, I can simply change the implementation of the empty procedure itself. I can implement a procedure with equivalent functionality as follows:

  ```
  PROCEDURE empty (table_out OUT date_table) IS
  BEGIN
     table_out.DELETE;
  END;
  ```

Yet I could also enhance the empty procedures of PLVtab to take full advantage of the flexibility offered by the DELETE attribute:

```
PROCEDURE empty
  (table_out OUT date_table,
   start_in IN INTEGER := NULL,
   end_in IN INTEGER := NULL)
IS
BEGIN
   table_out.DELETE
      (NVL (start_in, table_out.FIRST),
       NVL (end_in, table_out.LAST));
END;
```

Through careful assignment of default values for the arguments of this new implementation, all previous uses of the empty procedure would still be valid. Future uses could take advantage of the new arguments.[1]

[1] Why isn't this technique used in PLVtab? Well, at some point, I had to stop changing my code and instead write a book about it. You are, at least, now aware of the issue and can implement this approach yourself.

Special Notes on PLVtab

The PLVtab package supports only the table types listed in Table 8−1. You can add additional table types easily, but be sure to make each of these changes:

- Add the table TYPE statement for the desired datatype.
- Declare an "empty" table of that same table type.
- Create another version of the display procedure that accepts this table type.

PREVIOUS | HOME | NEXT
8.5 Setting the Display Prefix | BOOK INDEX | 8.7 Implementing PLVtab.display

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8*i* Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

8.7 Implementing PLVtab.display

I faced several challenges when building the **display** procedures:

- I had to create a separate procedure for each of the different table types, but I did not want to actually have separate display engines for each table type; the code would be very cumbersome and lengthy. Yet consolidating this code would also be difficult since each display procedure drew its information from a different PL/SQL table.

- With the implementation of PL/SQL tables prior to Release 2.3 of PL/SQL, it is impossible to obtain information about the state of a PL/SQL table from the runtime engine. Instead, you must keep track of the rows that have been used or be ready to handle the NO_DATA_FOUND exception.

I took care of the code redundancy problem by creating a single internal display procedure (**idisplay**) that is called by each of the public display procedures. Here is an example of the full body of the **display** procedure for date tables:

```
PROCEDURE display
   (tab_in IN date_table,
    end_in IN INTEGER,
    hdr_in IN VARCHAR2 := NULL,
    start_in IN INTEGER := 1,
    failure_threshold_in IN INTEGER := 0,
    increment_in IN INTEGER := +1)
IS
BEGIN
   idate := tab_in;
   idisplay
     (c_date, end_in, hdr_in, start_in,
      failure_threshold_in, increment_in);
   idate := empty_date;
END;
```

What is going on here? First, I copy the incoming table into a private PL/SQL table (**idate**). Then I display the contents of the **idate** table and not the user's table. Finally, I empty the internal PL/SQL table to minimize memory utilization. Notice that the **idate** table does not appear in the parameter list for **idisplay**. Instead, I pass in a constant, **c_date**, to indicate that **idisplay** should get the row values from the **idate** table.

I have, then, achieved my first objective: Use a single procedure to implement all of the different overloaded versions (this follows the diamond effect described in Chapter 4, *Getting Started with PL/Vision*). Of course, all I have really done is move the complexity down into the **idisplay** procedure. At least, however, all the complexity is concentrated into that single module.

But when you look inside the **idisplay** procedure you will find that there is actually very little complexity there either. Instead, I further buried the "how" of displaying all these different types of PL/SQL tables in another private module, **`display_row`**. Here is the main loop of the **`idisplay`** procedure:

```
WHILE in_range (current_row) AND within_threshold
LOOP
   display_row
   (type_in, failure_threshold_in, increment_in,
   count_misses, current_row, within_threshold);
END LOOP;
```

The **`display_row`** procedure takes as its first argument the type of table (**`c_date`**, **`c_number`**, etc.). It then executes an extended IF statement to determine from which table the row value should be extracted. The initial portion of this IF statement is shown below. The **`rowval`** variable is passed to **`p.l`** by the **`display_row`** procedure.

```
IF type_in = c_boolean
THEN
   rowval :=
      PLV.boolstg (iboolean (curr_row_inout), real_null_in => TRUE);

ELSIF type_in = c_date
THEN
   rowval := TO_CHAR (idate (curr_row_inout), PLV.datemask);

ELSIF type_in = c_integer
THEN
   rowval := TO_CHAR (iinteger (curr_row_inout));
```

Take some time to review the implementation of **`PLVtab.idisplay`**. On the one hand, you might think that the author is a somewhat off−balance, obsessive kind of guy. You might be right. On the other hand, you might also learn some interesting techniques for code consolidation inside packages when providing highly overloaded programs to users.

PREVIOUS	HOME	NEXT
8.6 Emptying Tables with PLVtab	BOOK INDEX	9. PLVmsg: Single−Sourcing PL/SQL Message Text

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8*i* Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

9. PLVmsg: Single−Sourcing PL/SQL Message Text

Contents:

The PLVmsg (PL/Vision MeSsaGe) package consolidates various kinds of message text in a single PL/SQL−based repository. Each message is associated with a number. You can then retrieve messages by number using the **`text`** function.

`PLVmsg` was originally designed to provide a programmatic interface to Oracle error messages and application−specific error text for error numbers in the −20,000 to −20,999 range (it is called in the **`PLVexc.handle`** program). The package is now, however, flexible enough to serve as a repository for message text of any kind.

This package allows you to:

- Assign individual text messages to specified rows in the PL/SQL table (the row is equal to the message number)
- Retrieve message text by number (which could be an error number or primary key)
- Automatically substitute your own messages for standard Oracle error messages with the restrict toggle
- Batch load message numbers and text from a database table directly into the PLVmsg PL/SQL table

This chapter shows how to use each of the different elements of the PLVmsg package.

9.1 PLVmsg Data Structures

The PL/SQL table used by PLVmsg to store message text is defined in the package body as follows:

```
msgtxt_table PLVtab.vc2000_table;
```

A PLVmsg message can therefore have a maximum of 2,000 bytes in the text.

The rows in this PL/SQL table are not filled sequentially. The rows are the message numbers and might represent Oracle error numbers, an entity's primary key values, or anything else the user passes as the message number. As a result (for PL/SQL Releases 2.2 and earlier), the PLVmsg package must keep track of the lowest and highest number rows. These values are stored in the following private variables:

```
v_min_row BINARY_INTEGER;
```

```
        v_max_row BINARY_INTEGER;
```

Note that a user of PLVmsg cannot make a direct reference to the **`msgtxt_table`** or the low and high row values; these data structures are hidden in the package body. I am, in this way, able to guarantee the integrity of the message text.

PREVIOUS | HOME | NEXT
8.7 Implementing PLVtab.display | BOOK INDEX | 9.2 Storing Message Text

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

9.2 Storing Message Text

Before your programs can retrieve messages from the PLVmsg PL/SQL table, you must place these messages in the table. You can do so in one of two ways:

1. Load individual messages with calls to the **add_text** procedure.

2. Load sets of messages from a database table with the **load_from_dbms** procedure.

9.2.1 Adding a Single Message

With **add_text**, you add specific strings to the message table at the specified row. Here is the header for **add_text**:

```
PROCEDURE add_text (err_code_in IN INTEGER, err_text_in IN VARCHAR2);
```

The following statements, for example, define message text for several error numbers set aside by Oracle Corporation for application−specific use (passed with a call to the RAISE_APPLICATION_ERROR builtin):

```
PLVmsg.add_text (−20000, 'General error');
PLVmsg.add_text (−20100, 'No department with that number.);
PLVmsg.add_text (−20200, 'Employee too young.');
```

Section 9.3, "Retrieving Message Text", later in this chapter, will show how you can extract these messages.

9.2.2 Batch Loading of Message Text

In many environments, a database table is used to store and maintain error messages, as well as other types of message text. The **load_from_dbms** procedure can be used to make this information available through the PLVmsg interface. The header for this procedure is:

```
PROCEDURE load_from_dbms
   (table_in IN VARCHAR2,
    where_clause_in IN VARCHAR2 := NULL,
    code_col_in IN VARCHAR2 := 'error_code',
    text_col_in IN VARCHAR2 := 'error_text');
```

This procedure reads the rows from the specified table and transfers them to the PL/SQL table. Recall that the PLVmsg **msgtxt_table** is not filled sequentially; the rows defined in the table are determined by the contents of the code column in the specified table.

To make the package as flexible as possible, PLVmsg relies on DBMS_SQL so that you can use whatever database table fits (or already exists) in your schema. When you call **load_from_dbms**, you tell it the name of the table and its columns, as well as an optional WHERE clause. The PLVmsg program then constructs the

SQL necessary to grab the text data. The only requirement of the table is that it has a numeric column for message numbers (used as PL/SQL table rows) and a string column for the message text.

You must, at a minimum, provide the name of the messages table. The default names of the columns are:

error_code
: The error number for the message

error_text
: The text of the error message

In the following call to **load_from_dbms**, I rely on the full set of defaults for the structure of the error table to transfer all rows from the **error_messages** table:

```
PLVmsg.load_from_dbms ('error_messages');
```

This request will work only if the **error_messages** table has columns named **error_code** and **error_text**.

In this next example, I supply customized values for all arguments:

```
PLVmsg.load_from_dbms
   ('errtxt',
    'code BETWEEN −20000 AND −20999',
    'code', 'text');
```

My table is named **errtxt** and has two columns, **code** and **text**. I further request that only the text for messages with error numbers between −20,000 and −20,999 be placed in the PLVmsg PL/SQL table. This WHERE clause implies that for all other errors, my program will rely on the message returned by SQLERRM (see the next section for more details).

You might be asking yourself: why bother with PLVmsg if you already have a database table−driven architecture for such messages? There are two key advantages:

1. With PLVmsg you will be reading the message text from memory (after the initial transfer) without having to go through the SQL layer. This will improve performance, though it will also require more memory since each user of PLVmsg will have her own copy of the messages.

2. PLVmsg is very flexible, in that you can dynamically direct your program to either the PLVmsg text or the database error message.

PREVIOUS | HOME | NEXT
9.1 PLVmsg Data Structures | BOOK INDEX | 9.3 Retrieving Message Text

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8*i* Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

9.3 Retrieving Message Text

The **`text`** function hides all the logical complexities involved in locating the correct message text and information about physical storage of text. You simply ask for the message and **`PLVmsg.text`** returns the information. That message may have come from SQLERRM or from the PL/SQL table. Your application doesn't have to address or be aware of these details. Here is the header for the **`text`** function (the full algorithm is shown in Example 9.1):

```
FUNCTION text (num_in IN INTEGER := SQLCODE) RETURN VARCHAR2;
```

You pass in a message number to retrieve the text for that message. If, on the other hand you do not provide a number, **`PLVmsg.text`** uses SQLCODE.

The following call to **`PLVmsg.text`** is, thus, roughly equivalent to displaying SQLERRM:

```
p.l (PLVmsg.text);
```

I say "roughly" because with PLVmsg you can also override the default Oracle message and provide your own text. This process is explained below.

Example 9.1: Algorithm for Choosing Message Text

```
FUNCTION text (num_in IN INTEGER := SQLCODE)
      RETURN VARCHAR2
IS
   msg VARCHAR2(2000);
BEGIN
   IF (num_in
         BETWEEN c_min_user_code AND c_max_user_code) OR
      (restricting AND NOT oracle_errnum (num_in)) OR
      NOT restricting
   THEN
      BEGIN
         msg := msgtxt_table (num_in);
      EXCEPTION
         WHEN OTHERS
         THEN
            IF oracle_errnum (num_in)
            THEN
               msg := SQLERRM (num_in);
            ELSE
               msg := 'No message for error code.';
            END IF;
      END;
   ELSE
      msg := SQLERRM (num_in);
   END IF;

   RETURN msg;
EXCEPTION
   WHEN OTHERS
```

```
      THEN
         RETURN NULL;
   END;
```

9.3.1 Substituting Oracle Messages

The following call to **`add_text`** is intended to override the default Oracle message for several rollback segment−related errors:

```
   FOR err_ind IN −1550 .. −1559
   LOOP
      PLVmsg.add_text
      (err_ind, 'Database failure; contact SysOp at x1212');
   END LOOP;
```

PREVIOUS | HOME | NEXT
9.2 Storing Message Text | BOOK INDEX | 9.4 The Restriction Toggle

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

9.4 The Restriction Toggle

Use the restriction toggle to determine whether messages for errors numbers that are legitimate Oracle error numbers will be retrieved from the PLVmsg PL/SQL table (unrestricted) or from the SQLERRM function (restricted). A legitimate Oracle error number is an integer that is negative or zero or 100 (equivalent to −1403 or "no data found").

The restriction toggle is composed of three programs:

```
PROCEDURE restrict;
PROCEDURE norestrict;
FUNCTION restricting RETURN BOOLEAN;
```

When you call the **`PLVmsg.restrict`** procedure (and this is the default setting), PLVmsg will rely on SQLERRM whenever appropriate to retrieve the message for a legitimate Oracle error number.

If you call **`norestrict`**, PLVmsg will first check the PL/SQL table of **`PLVmsg`** to see if there is an error message for that error. In unrestricted mode, therefore, you can automatically substitute standard Oracle error messages with your own text −− and be as selective as you like about the substitutions.

The **`restricting`** function will let you know the status of the **`restrict`** toggle in PLVmsg. It returns TRUE if you are restricting error messages to SQLERRM; otherwise, it will return FALSE.

Here are examples of the toggle in use:

1. In a SQL*Plus script, direct all error messages to be retrieved from the PL/SQL table, if present.

   ```
   PLVmsg.norestrict;
   transfer_data;
   ```

2. At the start of a SQL*Plus session, make sure that Oracle messages will be used whenever possible.

   ```
   SQL> exec PLVmsg.restrict;
   ```

PREVIOUS HOME NEXT
9.3 Retrieving Message Text BOOK INDEX 9.5 Integrating PLVmsg with Error Handling

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built−in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built−ins Pocket Reference

9.5 Integrating PLVmsg with Error Handling

Although PLVmsg can be used in other circumstances, PL/Vision uses it inside its exception handler package, PLVexc, and you are most likely to use it that way as well. This section shows you how to do this.

Suppose that you have taken the time to write a procedure named **showerr** to consolidate error handling. It accepts an error number−message combination and then both displays the message and records the error. If you do not make use of PLVmsg, a typical exception section might look like this:

```
EXCEPTION
   WHEN DUP_VAL_ON_INDEX
   THEN
      showerr (SQLCODE, 'Duplicate employee name.');
   WHEN OTHERS
   THEN
      IF SQLCODE = −20200
      THEN
         showerr (−20200, 'Employee too young.');
      ELSE
         showerr (SQLCODE, SQLERRM);
      END IF;
END;
```

What's the problem with this approach? I can think of several drawbacks:

- You have to do lots of typing. It took me several minutes to type out this example and I type quickly. It also provides lots of opportunities for errors.
- The developer has to know about DUP_VAL_ON_INDEX (I, for one, always get it wrong the first time; it seems that it should be IN_INDEX).
- There is some dangerous hard−coding in this section: both the −20,200 and the associated error message. What happens if you need to handle the same error in another program?

Now, suppose on the other hand that I had made use of PLVmsg. First, I would have added text to the PLVmsg repository as follows:

```
PLVmsg.add_text (−1, 'Duplicate employee name.');
PLVmsg.add_text (−20200, 'Employee too young.');
```

Sure, I had to know that ORA−00001 goes with the DUP_VAL_ON_INDEX exception, but remember that I will be writing this once for all developers on an application team. After setting these values I would also have called the **norestrict** toggle. This allows PLVmsg to override the usual error message for ORA−00001 with my own message.

```
PLVmsg.norestrict;
```

With the text in place and restrictions removed on accessing override messages, I can reduce my exception section from what you saw earlier to just this:

```
EXCEPTION
   WHEN OTHERS
   THEN
      showerr (SQLCODE, PLVmsg.text);
END;
```

When the SQLCODE is −1, **`PLVmsg.text`** is routed to the contents of the PL/SQL table in row −1 (and does not use SQLERRM). When SQLCODE is −20,200, the value in row −202000 is returned. Finally, for all other regular Oracle error numbers, PLVmsg obtains the text from SQLERRM.

The result is a dramatically cleaned−up exception section and an application in which all error text management is performed in one place: the PLVmsg repository.

9.5.1 Using PLVmsg in PL/Vision

As mentioned earlier, the PLVexc packages relies on PLVmsg to obtain error message text. The **`PLVmsg.text`** function is called by **`terminate_and_handle`**, which acts as a bridge between the high−level handlers, such as **`recNgo`**, and the low−level handle procedure. The implementation of **`terminate_and_handle`** is shown below:

```
PROCEDURE terminate_and_handle
   (action_in IN VARCHAR2,
    err_code_in IN INTEGER)
IS
BEGIN
   PLVtrc.terminate;
   handle
      (PLVtrc.prevmod, err_code_in, action_in,
       PLVmsg.text (err_code_in));
END;
```

The value passed in as **`err_code_in`** might be SQLCODE, or it might be some application−specific value. Whatever its value, **`PLVmsg.text`** translates the error number into message text and passes that to the low−level handler. The handle procedure then might display this string or store it in the PL/Vision log.

By calling **`PLVmsg.text`** at this point in the exception−handling architecture, PLVexc offers its users a lot of flexibility. Suppose that when you first built your application, you didn't have time to work on error messages. You took advantage of PLVexc, but ignored completely the PLVmsg package capabilities. In this case, **`PLVmsg.text`** acted simply as a passthrough to SQLERRM. Somewhere down the line, however, you decided to enhance the error messages for your application.

To accomplish this enhancement, you would not have to change your application. All of your exception handlers that call the high−level **`PLVexc`** exception handlers are already calling **`PLVmsg.text`**. All you have to do is store all of your message text in a database table and then call **`PLVmsg.load_from_dbms`** at a good startup point for the application (in a When−New−Form−Instance trigger in an Oracle Forms−based application or in the initialization section of a common package).

From that point on (and remember: without changing any of your code!), the new error text will be used in the application.

Special Notes on PLVmsg

Here are some factors to consider when working with PLVmsg:

- The maximum size of a message is 2,000 bytes.
- The number 100 and all negative numbers that are not between −20,000 and −20999 are considered to be Oracle error codes.
- The **`load_from_dbms`** is a useful example of the kind of code you need to write to transfer data from a database table to a PL/SQL table −− even to the extent of allowing the user to specify the relevant names. You should be able to easily adapt this PLVmsg procedure to your own purposes.

PREVIOUS | HOME | NEXT
9.4 The Restriction Toggle | BOOK INDEX | 9.6 Implementing load_from_dbms

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8*i* Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

9.6 Implementing load_ from_dbms

The **load_from_dbms** procedure serves as a good example of a program for loading number−text combinations from any database table into a PL/SQL table using dynamic SQL. Since you can specify the table name, WHERE clause, and column names, you can load message text from multiple sources and for multiple purposes. You can even copy this program, modify it, and use it in other programs.

The implementation of this procedure is shown in Example 9.2. It is explained in the next section. (continued)

Example 9.2: The Implementation of load_ from_dbms

```
PROCEDURE load_from_dbms
   (table_in IN VARCHAR2,
    where_clause_in IN VARCHAR2 := NULL,
    code_col_in IN VARCHAR2 := 'error_code',
    text_col_in IN VARCHAR2 := 'error_text')
IS
   select_string PLV.max_varchar2%TYPE :=
    'SELECT ' || code_col_in || ', ' || text_col_in ||
    '  FROM ' || table_in;

   cur INTEGER;
   error_code INTEGER;
   error_text VARCHAR2(2000);

   PROCEDURE set_minmax (code_in IN INTEGER) IS
   BEGIN
      IF min_row IS NULL OR min_row > code_in
      THEN
         v_min_row := code_in;
      END IF;

      IF max_row IS NULL OR max_row < code_in
      THEN
         v_max_row := code_in;
      END IF;
   END;
BEGIN
   IF where_clause_in IS NOT NULL
   THEN
      select_string := select_string || ' WHERE ' || where_clause_in;
   END IF;

   cur := PLVdyn.open_and_parse (select_string);
   DBMS_SQL.DEFINE_COLUMN (cur, 1, error_code);
   DBMS_SQL.DEFINE_COLUMN (cur, 2, error_text, 2000);

   PLVdyn.execute (cur);
   LOOP
      EXIT WHEN DBMS_SQL.FETCH_ROWS (cur) = 0;
      DBMS_SQL.COLUMN_VALUE (cur, 1, error_code);
      DBMS_SQL.COLUMN_VALUE (cur, 2, error_text);
      set_minmax (error_code);
      add_text (error_code, error_text);
   END LOOP;
   DBMS_SQL.CLOSE_CURSOR (cur);
END;
```

When performing dynamic SQL, you construct the SQL statement at runtime. In **load_from_dbms**, I declare and initialize my SELECT string as follows:

```
select_string PLV.max_varchar2%TYPE :=
   'SELECT ' || code_col_in || ', ' || text_col_in ||
   '  FROM ' || table_in;
```

Notice that I use all of the name arguments to the program to build the SELECT statement. There is nothing hard coded here except for the mandatory syntax elements like SELECT and FROM. That assignment (which takes place in the declaration section) covers the basic query.

What if the user provided a WHERE clause? The first line in the procedure's body adds the WHERE clause if it is not NULL:

```
IF where_clause_in IS NOT NULL
THEN
   select_string := select_string || ' WHERE ' || where_clause_in;
END IF;
```

Notice that you do not have to provide a WHERE keyword. That is inserted automatically by the program.

My string is ready to be parsed, so I call the **PLVdyn open_and_parse** procedure to take those two steps. Then I define the two columns (number and string) that are specified in the SELECT statement:

```
cur := PLVdyn.open_and_parse (select_string);
DBMS_SQL.DEFINE_COLUMN (cur, 1, error_code);
DBMS_SQL.DEFINE_COLUMN (cur, 2, error_text, 2000);
```

Now the cursor is fully defined and ready to be executed. The final step in **load_from_dbms** is to run the equivalent of a cursor FOR loop: for every record dynamically fetched, add the text to the table and update the high and low indicators:

```
PLVdyn.execute (cur);
LOOP
   EXIT WHEN DBMS_SQL.FETCH_ROWS (cur) = 0;
   DBMS_SQL.COLUMN_VALUE (cur, 1, error_code);
   DBMS_SQL.COLUMN_VALUE (cur, 2, error_text);

   set_minmax (error_code);

   add_text (error_code, error_text);
END LOOP;
DBMS_SQL.CLOSE_CURSOR (cur);
```

I fetch a row (exiting immediately if nothing is returned). I then extract the values into local variables with COLUMN_VALUE. Following that, I update the minimum and maximum row numbers and, finally, add the text to the PL/SQL table using that same **add_text** program that users of **PLVmsg** would use to add text to the table. When I am done with the loop, I close the cursor.

To make the main body of the procedure as readable as possible, I create a local procedure (**set_minmax**) to keep track of the lowest and highest row numbers used in the PL/SQL table. This is necessary in releases of PL/SQL earlier than 2.3 since there is no way to query the PL/SQL runtime engine for this information. The local procedure, **set_minmax**, also serves to hide this annoying level of detail and weakness in PL/SQL table design. When you upgrade to PL/SQL Release 2.3 or above, you can just strip out this code.

PREVIOUS
9.5 Integrating PLVmsg with Error Handling

HOME
BOOK INDEX

NEXT
10. PLVprs, PLVtkn, and PLVprsps: Parsing Strings

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

10. PLVprs, PLVtkn, and PLVprsps: Parsing Strings

Contents:

Parsing a computer program can be a very frustrating and complex experience. There are all kinds of exceptions to the rules and special rules to handle in one's logic. The only way I could handle all of these details was to deal with as narrow a portion of functionality at a time as I could. The result is a set of four different packages oriented to different levels of the parsing process. By isolating various areas of complexity into these different packages, I can keep each of the individual parsing programs brief and relatively easy to write and understand.

PL/Vision offers several different string and source code parsing packages:

PLVprs
: Generic string−parsing extensions to PL/SQL. This is the lowest level of string−parsing functionality and will prove useful in many different situations.

PLVtkn
: Interface to the **`PLV_token`** table, which contains more than 1,200 keywords of the PL/SQL language, including those for Oracle Forms. Use **`PLVtkn`** to determine if an identifier is a keyword.

PLVlex
: PL/SQL lexical analysis package. Performs similar parsing actions as those available in **`PLVprs`**, but does so with an awareness of the syntax and delimiters of the PL/SQL language.[1]

> [1] This package is not described further in the book; see the the companion disk for more information.

PLVprsps
: Highest−level package to parse PL/SQL source code (hence the **`prsps`** name) into separate atomics. Relies on the other parsing packages to get its job done.

10.1 PLVprs: Useful String Parsing Extensions

The PLVprs (PL/Vision PaRSe) package offers a set of procedures and functions that provide generic and very flexible string−parsing functionality. These programs extend the builtin string functions of PL/SQL. They can:

- Parse a string into its atomics
- Count the number of atomics in a string
- Count the frequency of a substring within a string
-

Return the *n* th atomic in a string

- Wrap a long line of text into a paragraph
- Display a wrapped line of text

You can use PLVprs within other packages and programs to analyze strings and display their contents. It is also used within PL/Vision by PLVvu and PLVdyn to display long messages.

All of our applications require manipulation of textual information. I have often encountered the need to parse and analyze those strings in order to answer requests like the following (and I am sure that you could add more to the list):

- Count the number of words in a string.
- Separate out all words and punctuation in a string into separate components.
- Return the *n*th value in a semicolon−delimited string. This is a very common situation in Oracle Forms applications, in which a developer might pack a set of values into a global variable like this: "123;5555;6623.11;".

10.1.1 Developing a General Solution

Taken separately, it is not too hard to develop a solution to any of the items on this list. If you build solutions to each individual requirement on a case−by−case basis, you will end up with an enormous volume of redundant code which you cannot easily enhance or upgrade.

Rather than construct a smattering of different, specialized routines to analyze strings, I offer with PLVprs a set of very generic and flexible functions and procedures. Yet the only way to make sure that my approach will handle many different situations and requirements is to base that approach on a general analysis of the components of a string.

In the world of PLVprs, a string is made up of a series of atomics, the smallest indivisible elements of the string. An atomic is either a word (defined as a contiguous set of letters and/or numbers) or a delimiter (any one of a set of special characters which separate or delimit words). In the English language, for example, common delimiters, a.k.a. punctuation, include the comma, semicolon, question mark, period, etc. But in a more general scenario, we have to be careful not to assume that the set of delimiters is a constant. For example, if I want to perform operations on a string of values delimited by a semicolon, then the set of delimiters for that particular operation is a single character: the semicolon.

Once you see strings as a set of atomics, of words separated by delimiters, you can reinterpret and enhance the kinds of requests listed above, such as "count all words in a string." Do you want to count all the atomics or just the words or just the delimiters? PLVprs gives you that level of flexibility.

10.1.2 Customizing the Delimiter Set

The PLVprs package interprets a string as a series of atomics: words and delimiters. But what is a delimiter? PLVprs predefines two sets of delimiters:

<table>
<tr><td>Name</td><td>Characters in Delimeter Set</td></tr>
<tr><td>std_delimeters</td><td>!@#$%^&*() _=+\|`~{{]};:"",<.>/?' plus space, tab, and newline characters</td></tr>
<tr><td>plsql_delimeters</td><td>!@%^&*() =+\|`~{{]};:"",<.>/?' plus space, tab, and newline characters</td></tr>
</table>

The only difference between these lists is that the **plsql_delimiters** set omits the underscore, dollar sign, and pound sign characters. These characters are valid symbols in a PL/SQL identifier, which should certainly be considered a word.

Would you ever need any other delimiter sets? Sure, why not? You might have a string which is packed with values separated by the vertical bar. In this situation, when you call **PLVprs.string** to separate the string into separate atomics, you will want to be able to specify your special and very short delimiter list.

PLVprs lets you specify a non−default delimiter list in the following programs:

next_atom_loc
nth_atomic
display_atomics
string
numatomics

Here is an example of how I could parse a string with atomics packed between vertical bars and ignore any other delimiters:

```
DECLARE
   atomic PLVtab.vc2000_table;
   num INTEGER;
BEGIN
   PLVprs.string ('A#−%|12345|(*&*)|0101R|', atomic, num, '|');
   PLVtab.display (atomic,num);
END;
/
```

This is the output seen after executing the above script:

```
Contents of Table
A#−%
|
12345
|
(*&*)
|
0101R
|
```

With the default set of delimiters, this string would have been broken up by **PLVprs.string** into fifteen separate atomics, rather than just eight. So don't forget to use your own delimiter list as needed to simplify your parsing and analysis jobs.

10.1.3 Parsing Strings into Atomics

The PLVprs package offers a number of programs that perform different parsing operations on strings. They are each discussed below. The implementations of most of these programs were discussed in *Oracle PL/SQL Programming*, so I will not go beyond an explanation here of how you can use these functions.

10.1.3.1 next_atom_loc function

The **`next_atom_loc`** returns the location in the specified string of the next atomic. It is used by other PLVprs programs, but is also available for use by other PL/Vision packages −− and by you. Its header is:

```
FUNCTION next_atom_loc
     (string_in IN VARCHAR2,
      start_loc_in IN NUMBER,
      direction_in IN NUMBER := +1,
      delimiters_in IN VARCHAR2 := std_delimiters)
RETURN INTEGER;
```

The **`string_in`** parameter is the string to be scanned. The **`start_loc_in`** parameter provides the starting position of the search for the start of the next atomic. The **`direction_in`** parameter indicates whether the search should move forward or backward through the string. The final argument allows you to specify the set of characters to be considered delimiters (which indicate the start of a new atomic).

The **`next_atom_loc`** function returns the location in the string of the starting point of the next atomic (from the start location). The function scans forward if **`direction_in`** is +1, otherwise it scans backwards through the string. Here is the logic to determine when the next atomic starts:

1. If the current atomic is a delimiter (that is, if the character at the **`start_loc_in`** of the string is a delimiter), then the next character starts the next atomic since all delimiters are a single character in length.

2. If the current atomic is a word (that is, if the character at the **`start_loc_in`** of the string is a letter or number), then the next atomic starts at the next delimiter. Any letters or numbers in between are part of the current atomic.

The **`next_atomic_loc`** function loops through the string one character at a time and applies these tests. It also has to check for the end of string. If it scans forward, the end of string comes when the SUBSTR that pulls out the next character returns NULL. If it scans backward, then the end of the string comes when the location is less than 0.

10.1.3.2 display_atomics procedure

The **`display_atomics`** procedure displays the atomics found in the specified string. Its header is shown below:

```
PROCEDURE display_atomics
   (string_in IN VARCHAR2,
    delimiters_in IN VARCHAR2 := std_delimiters);
```

You specify the string you want parsed and the set of characters you want to be treated as delimiters for the parsing. To make it easier to view blank lines, spaces are displayed as a description of the number of spaces present (as in "1 blank" or "6 blanks"). This feature is shown below:

```
SQL> exec PLVtab.showrow
SQL> exec PLVprs.display_atomics ('Compassion is a human thing.');
Parsed Atomics in String
Row 1 = Compassion
Row 2 = 1 blank
Row 3 = is
Row 4 = 1 blank
Row 5 = a
Row 6 = 1 blank
Row 7 = human
Row 8 = 2 blanks
Row 9 = thing
Row 10 = .
```

You can specify an alternate delimiter set in order to display the contents of a string according to the format of that string and how it is used. Compare the two calls to **`display_atomics`** below:

```
SQL> exec PLVprs.display_atomics ('1234|A$%|67YYY|(big)');
11
Parsed Atomics in String
1234
−|−
A
−$−
−%−
−|−
67YYY
−|−
−(−
big
−)−
SQL> exec PLVprs.display_atomics ('1234|A$%|67YYY|(big)', '|');
7
Parsed Atomics in String
1234
−|−
A$%
−|−
67YYY
−|−
(big)
```

In the second call, I am telling **`display_atomics`** to consider "|" as the sole delimiter in the string.

10.1.3.3 numatomics function

The **`numatomics`** function returns the number of atomics in a string. You can calculate the number of all atomics, only the words, or only the delimiters. You can specify in your call to the function what characters should be considered delimiters. The header for this function is:

```
FUNCTION numatomics
   (string_in IN VARCHAR2,
    count_type_in IN VARCHAR2 := c_all,
    delimiters_in IN VARCHAR2 := std_delimiters)
RETURN INTEGER;
```

The following examples demonstrate how to use the **`count_type_in`** argument. The first call relies on the default value of "all atomics." The next two calls pass in a specific request for type of atomic, relying on the constants provided in the package specification.

```
SQL> exec p.l (PLVprs.numatomics ('this, is%not.'))
7
SQL> exec p.l (PLVprs.numatomics ('this, is%not.', PLVprs.c_word))
3
SQL> exec p.l (PLVprs.numatomics ('this, is%not.', PLVprs.c_delim))
4
```

10.1.3.4 nth_atomic function

The **`nth_atomic`** function returns the *n* th atomic in a string. You can, for example, ask for the seventh word or the sixth delimiter. The header for **`nth_atomic`** is:

```
FUNCTION nth_atomic
   (string_in IN VARCHAR2,
    nth_in IN NUMBER,
    count_type_in IN VARCHAR2 := c_all,
    delimiters_in IN VARCHAR2 := std_delimiters)
RETURN VARCHAR2;
```

The **`nth_atomic`** function is very flexible, following the model of the builtin functions, INSTR and SUBSTR. You can scan both forward and backward through the string. If you provide a positive **`nth_in`** argument, then **`nth_atomic`** scans forward to the *n*th atomic. If, on the other hand, the **`nth_in`** value is negative, **`nth_atomic`** will scan backwards through the string to the *n*th atomic. This feature is shown in the examples below:

```
SQL> exec p.l(PLVprs.nth_atomic ('this, is%not.', 2))
,
SQL> exec p.l(PLVprs.nth_atomic ('this, is%not.', 2, PLVprs.c_word))
is
SQL> exec p.l(PLVprs.nth_atomic ('this, is%not.', 3, PLVprs.c_delim))
%
SQL> exec p.l(PLVprs.nth_atomic ('this, is%not.', −3, PLVprs.c_word))
this
```

The PLVdyn package also utilizes **`nth_atomic`** function to extract the type of program unit and the name of the program unit that is being compiled and stored by the **`compile`** procedure:

```
/* Get the first word.*/
v_name1 := PLVprs.nth_atomic (stg_in, 1, PLVprs.c_word);

/* Get the second word. */
v_name2 := PLVprs.nth_atomic (stg_in, 2, PLVprs.c_word);

/* If a package body, then get the THIRD word. */
IF UPPER (v_name1||' '||v_name2) = 'PACKAGE BODY'
THEN
   v_name1 := v_name1 || ' ' || v_name2;
   v_name2 := PLVprs.nth_atomic (stg_in, 3, PLVprs.c_word);
END IF;
```

10.1.3.5 string procedures

There are two, overloaded versions of the **`string`** procedure, which parses a string into its separate atomics. The **`string`** procedure is available to users of the PLVprs package; it is also used by programs in the PLVprs package. The **`display_atomics`** procedure, for example, calls the **`string`** procedure to parse the specified string and then calls **`PLVtab.display`** to display the table that contains the parsed atomics.

The headers for the **`string`** procedures are:

```
PROCEDURE string
   (string_in IN VARCHAR2,
    atomics_list_out OUT PLVtab.vc2000_table,
    num_atomics_out IN OUT NUMBER,
    delimiters_in IN VARCHAR2 := std_delimiters);

PROCEDURE string
   (string_in IN VARCHAR2,
    atomics_list_out IN OUT VARCHAR2,
    num_atomics_out IN OUT NUMBER,
    delimiters_in IN VARCHAR2 := std_delimiters);
```

The first argument is the string to be parsed. The second and third arguments contain the parsed output of the procedure call. The fourth argument allows you to specify an alternative set of delimiter characters.

The table version of **string** fills a PL/SQL table with the atomics of the string, one atomic per row. The VARCHAR2 version of the **string** procedure returns the atomics in a string separated by the vertical bar delimiter. This "string version" of **string** simply calls the table version and then dumps the contents of the table into a string.

10.1.3.6 numinstr function

The **numinstr** function is a good example of how you can supplement the fine, but finite set of builtin string functions with your own basic and quite reusable functions. The **numinstr** function returns the number of times a substring appears in a string. Its header is:

```
FUNCTION numinstr
   (string_in IN VARCHAR2,
    substring_in IN VARCHAR2,
    ignore_case_in IN VARCHAR2 := c_ignore_case)
RETURN INTEGER;
```

The first argument is the string, the second is the substring, and the third allows you to specify whether you want case to be ignored or respected in the search. The following examples illustrate this flexibility:

```
SQL> exec p.l (PLVprs.numinstr ('abcabC', 'c'));
2
SQL> exec p.l (PLVprs.numinstr ('abcabC', 'c', PLVprs.c_respect_case));
1
```

In the following code fragment, I call **numinstr** to determine the number of placeholders for bind variables in a dynamically constructed SQL string (a placeholder is defined as a colon followed by a PL/SQL identifier, such as **:newname** or **:bindvar1**). I then use a numeric FOR loop to issue a call to the BIND_VARIABLE procedure of the builtin DBMS_SQL package for each of the bind variables.

```
numvars := PLVprs.numinstr (sql_string, ':');
FOR bindvar_ind IN 1 .. numvars
LOOP
   DBMS_SQL.BIND_VARIABLE
     (cur_handle,
      'bindvar' || TO_CHAR (bindvar_ind),
      variables_table (bindvar_in));
END LOOP;
```

Finding the Best Solution

I believe I am a fairly good PL/SQL programmer. I can churn out hundreds of lines of complicated code that works, more or less, after just a few rounds of revisions. On the other hand, I feel I am also very much open to the possibility that others can and have done better −− and that I can learn from them. None of us has all the answers −− and some of us have more answers than others.

Let me give you an example. In my first book on PL/SQL, *Oracle PL/SQL Programming* (O'Reilly & Associates), I give every impression that I know what I am talking about −− and a big part of what I talk about is writing concise, high−quality code. In Chapter 11, *Character Functions*, I take my readers through the exercise of building a function to count the number of times a substring occurs in a string (a function not provided by PL/SQL). I end up with two implementations, the shorter of which is shown in Example 10.1.

Example 10.1: Counting the Frequency of a Substring Within a String

```
FUNCTION numinstr
   (string_in IN VARCHAR2, substring_in IN VARCHAR2)
RETURN INTEGER
IS
   substring_loc NUMBER;
   return_value NUMBER := 1;
BEGIN
   LOOP
      substring_loc :=
         INSTR (UPPER (string_in),
                UPPER (substring_in), 1, return_value);

      /* Terminate loop when no more occurrences are found. */
      EXIT WHEN substring_loc = 0;

      /* Found match, so add to total and continue. */
      return_value := return_value + 1;
   END LOOP;
   RETURN return_value - 1;
END numinstr;
```

I was quite content with this function −− until I received an email from Kevin Loney, author of *ORACLE* DBA Handbook (Oracle Press). This email very politely complimented me on my book and offered an alternative implementation for **numinstr** −− a simpler, more efficient, and more elegant implementation. As Kevin noted, he came up with this solution before the days of PL/SQL, when all he had to work with was SQL (a set−at−a−time, nonprocedural language). In his approach (shown in Example 10.2), he took advantage of the REPLACE builtin function to substitute NULL for any occurrences of the substring. He could then compare the size of the original string with the "replaced" string and use that as the basis for his calculation.

Example 10.2: Counting the Frequency of a Substring Within a String

```
FUNCTION numinstr
    (string_in IN VARCHAR2, sub_string_in IN VARCHAR2)
RETURN INTEGER
/* Divide difference of two lengths by length of substring. */
IS
BEGIN
   RETURN
     ((LENGTH (string_in) -
       NVL (LENGTH (REPLACE (string_in, sub_string_in)), 0))
     / LENGTH (sub_string_in));
END;
```

I felt a wave of embarrassment wash over me for just a moment when I first read Kevin's note. Then I regained sanity. *Of course* there are better ways of doing things. Discovering and sharing these improvements −− programming altruism −− is one of the finest aspects of our work. And programming humility −− the willingness to accept these improvements −− can make our lives much easier. If you are open to the ideas of others, then you are also open to the idea of using the work of others (with permission!). This means that you will spend much less time coding something that is already available −− you will avoid reinventing the wheel. Kevin's implentation for `numinstr` (enhanced to ignore or respect case) is now in version I provide in PLVprs.

10.1.4 Wrapping Strings into Paragraphs

How many times have you been confronted with the need to display a long string (defined, essentially, as anything longer than 80 characters) in a format which can be read easily? Unfortunately, PL/SQL does not have any paragraph−wrapping capabilities built into it. All you have is DBMS_OUTPUT.PUT_LINE, which accepts a maximum of 255 characters and is displayed in whatever manner handled by your environment.

PLVprs fills this gap in functionality by providing the following suite of string−wrapping programs:

wrap

Wraps a long string into a series of lines with a maximum specified length, each line of which is stored in consecutive rows in a PL/SQL table.

wrapped_string
Returns a long string wrapped into a series of lines separated by newline characters.

display_wrap
Displays a long string in paragraph−wrapped form at the specified length.

The **wrap** procedure supplies the core paragraph−wrapping functionality and is called both by **wrapped_string** and **display_wrap**. All three programs are described below.

10.1.4.1 wrap procedure

The header for the **wrap** procedure is:

```
PROCEDURE wrap
  (text_in IN VARCHAR2,
   line_length IN INTEGER,
   paragraph_out IN OUT PLVtab.vc2000_table,
   num_lines_out IN OUT INTEGER);
```

The first parameter is the text to be wrapped. The second parameter, **line_length**, is the length of the line into which the long text is to be wrapped. The **paragraph_out** argument is the PL/SQL table which will receive the wrapped text (each row, starting from 1, contains a single line of text). The **num_lines_out** argument contains the number of lines of text that have been placed in **paragraph_out**.

Examples of **PLVprs.wrap** are shown in the following sections.

10.1.4.2 display_wrap procedure

The **display_wrap** procedure comes in handy when you only want to display the end result −− the wrapped string. If you have no need to store that string in a PL/SQL table or a single string variable for further manipulation, call **display_wrap**. Then you will not have to declare temporary data structures to hold the wrapped text until it is displayed.

```
PROCEDURE display_wrap
  (text_in IN VARCHAR2,
   line_length IN INTEGER := 80,
   prefix_in IN VARCHAR2 := NULL);
```

The first parameter, **text_in**, is the string to be wrapped and then displayed. The second argument, **line_length**, is the length of the line within which the string is wrapped. The third argument, **prefix_in**, allows you to specify an optional prefix to display before each line of the wrapped string.

Here is an example of **display_wrap**:

```
SQL> exec PLVprs.display_wrap (RPAD ('a string',120,' to remember'),30);
a string to remember to
remember to remember to
remember to remember to
remember to remember to
remember to remember to
```

The implementation of **display_wrap** is short and sweet. It calls the **wrap** procedure to fill up a locally declared PL/SQL table with the wrapped lines of text. It then calls **PLVtab.display** to display the contents of that table.

The **p** package of PL/Vision makes use of the **display_wrap** procedure to automatically wrap long lines of text. As a result, when you call **p.l**, you don't have to worry about the DBMS_OUTPUT restriction of 255 characters displayed in a string. Instead, it will automatically check the length of the string and wrap the output as shown below:

```
ELSIF LENGTH (line_in) > 80
THEN
   PLVprs.display_wrap (line_in, 75, NULL);
```

Now why did I include a third argument of NULL in my call to **`display_wrap`**? That (lack of a) value is, after all, the default. I could simply have called **`display_wrap`** as follows:

```
PLVprs.display_wrap (line_in, 75);
```

and received the same results.

Usually I am a strong advocate of typing only the absolute minimum necessary to get the job done. Have I violated my own guidelines? Not really. I have, in fact, typed exactly what I needed to get the job done. The question you should now ask is: What is my "job" or requirement? I want to display a wrapped string without any prefix. The way I do that is to pass NULL for the prefix; hence, the inclusion of three arguments.

Notice that I did not state my requirement as follows: "I want to display a wrapped string with the default prefix." If that were my desire, I *should* pass just two arguments. That way, if the default ever changes, my call to **`display_wrap`** will automatically adapt to the new value. But in the **`p.l`** package I would not want the output from **`PLVprs.display_wrap`** to change when the default value changes. I really do want a NULL value, regardless of the actual default value for the **`display_wrap`** procedure. In **`p.l`**'s use of **`display_wrap`**, in other words, the fact that the desired prefix (none) is the same as the default is nothing more than a coincidence.

This distinction between needing the default value and happening to match the default value is an important one. If you simply have a coincidence, do not rely on the default value. Instead, pass in the value, even if it is currently the default and not, strictly speaking, needed. Your code will be less likely to break as the underlying layers of code you rely on change.

10.1.4.3 wrapped_string function

The last string−wrapping program is the **`wrapped_string`** function. This function returns a string with the wrapped lines of text concatenated together, with newline characters after every wrapped line. The specification for **`wrapped_string`** is as follows:

```
FUNCTION wrapped_string
  (text_in IN VARCHAR2,
   line_length IN INTEGER := 80,
   prefix_in IN VARCHAR2 := NULL)
RETURN VARCHAR2;
```

The first argument, **`text_in`**, is the string to be wrapped and then displayed. The second argument, **`line_length`**, is the length of the line within which the string is wrapped. The third argument, **`prefix_in`**, allows you to specify an optional prefix to display before *each* line of the wrapped string.

This function is useful in situations (such as that currently found in Oracle Developer/2000 Version 1) where you cannot reference and use PL/SQL tables. The **`wrapped_string`** returns the text in a string which can then be immediately displayed in the right format (because of the embedded newline characters) or passed to another program which works with the wrapped text.

Special Notes on PLVprs

Here are some factors to consider when working with PLVprs:

-

`PLVprs` does not currently recognize many nonprinting characters as delimiters (only newline and tab are in the predefined delimiter lists). If you have, for example, a backspace character in your string, this will not be treated as a delimiter. You can, of course, define your own delimiter list and then pass that set of characters to PLVprs's program.

- When you wrap a string, all carriage returns are replaced with single characters. Your "hard−coded" line breaks will be ignored in the wrap process.

PREVIOUS | HOME | NEXT
9.6 Implementing load_ from_dbms | BOOK INDEX | 10.2 PLVtkn: Managing PL/SQL Tokens

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

10.2 PLVtkn: Managing PL/SQL Tokens

The **PLVtkn** (PL/Vision ToKeN) package determines whether an identifier is a PL/SQL keyword. It does this by using a token table containing these keywords.

Every PL/SQL program is filled with identifiers. Identifiers are named PL/SQL language elements and include variable names, program names, and reserved words. Reserved words play a very different role in our programs than do the application−specific identifiers. I recommend strongly in *Oracle PL/SQL Programming* that you reflect these different roles in your program by using the UPPER−lower method: all reserved words are typed in UPPER case and all application−specific identifiers are typed in lower case. I even go so far, in PL/Vision, as to provide you with a package (PLVcase) which will automatically convert your programs to the UPPER−lower method.

10.2.1 Keeping Track of PL/SQL Keywords

Well, if **PLVcase** is going to uppercase only keywords, it has to know which identifiers in a PL/SQL program are the reserved words. This information is maintained in the **PLV_token** table, which has the following structure:

```
Name                            Null?    Type
------------------------------- -------- ----
TOKEN                           NOT NULL VARCHAR2(100)
TOKEN_TYPE                               VARCHAR2(10)
```

where the **token** column is the identifier and **token_type** indicates the type.

The different token types in the **PLV_token** table are stored in the **PLV_token_type** table, which has this structure:

```
Name                            Null?    Type
------------------------------- -------- ----
TOKEN_TYPE                               VARCHAR2(10)
NAME                            NOT NULL VARCHAR2(100)
```

The contents of the **PLV_token_type** are explained in the following table:

Token Type	Name	Description
B	BUILT−IN	Builtin functions and procedures of the PL/SQL language, including packaged builtins.
D	DATATYPE	Different datatypes of the PL/SQL language, such as INTEGER and VARCHAR2.
DD	DATA−DICTIONARY	Views and tables from the Oracle Server data dictionary, such as ALL_SOURCE and DUAL.
E	EXCEPTION	Predefined system exceptions such as ZERO_DIVIDE.

OF	ORACLE−FORMS	Reserved words from the Oracle Forms product set
S	SYMBOL	Symbols like + and =.
SQL	SQL	Elements of the SQL language. In many cases, SQL tokens are used in PL/SQL and also in Oracle Developer/2000. These are still listed as SQL tokens.
X	SYNTAX	Syntax elements of the PL/SQL language, such as AND or LIKE.

There is a row in **`PLV_token`** for each reserved word in PL/SQL. You can change the contents of this table if you want. You might, for example, want to add keywords for the Oracle Developer/2000 builtins or the Oracle Web Agent PL/SQL packages. You can even add your own application−specific identifiers to the table. As long as the token type you assign is not any of those listed above, PL/Vision will not misinterpret your entries.

There are currently 1,235 rows in the **`PLV_token`** table, broken down by token type as follows:

Token Type	Count
BUILT−IN	198
DATATYPE	22
DATA−DICTIONARY	168
EXCEPTION	15
ORACLE−FORMS	623
SYMBOL	32
SQL	94
SYNTAX	83

From the PL/SQL side of things, the PLVtkn package provides an interface to the **`PLV_token`** table. This package is used by PLVcase to determine the case of an individual token according to the UPPER−lower method.

As you will soon see, PLVtkn is not a particularly large or complicated package. Its purpose in life is to consolidate all of the logic having to do with individual PL/SQL tokens, particularly regarding keywords. By hiding the implementation details (the name and structure of the **`PLV_token`** table, the particular values used to denote a symbol or syntax element or builtin function), PLVtkn makes it easier for developers to apply this information in their own programs.

10.2.2 Determining Token Type

PLVtkn provides a set of functions you can use to determine a string's token type in PL/SQL. The headers for these functions are shown below:

```
FUNCTION is_keyword
   (token_in IN VARCHAR2, type_in IN VARCHAR2 := c_any) RETURN BOOLEAN;

FUNCTION is_syntax (token_in IN VARCHAR2) RETURN BOOLEAN;
FUNCTION is_builtin (token_in IN VARCHAR2) RETURN BOOLEAN;
FUNCTION is_symbol (token_in IN VARCHAR2) RETURN BOOLEAN;
FUNCTION is_datatype (token_in IN VARCHAR2) RETURN BOOLEAN;
FUNCTION is_exception (token_in IN VARCHAR2) RETURN BOOLEAN;
```

All of the functions except for **`is_keyword`** take a single string argument and return TRUE if the string is that type of token. The following examples illustrate the way the PLVtkn functions interpret various strings:

```
SQL> exec p.l(PLVtkn.is_builtin('to_char'));
TRUE
SQL> exec p.l(PLVtkn.is_builtin('loop'));
FALSE
SQL> exec p.l(PLVtkn.is_syntax('loop'));
TRUE
SQL> exec p.l(PLVtkn.is_syntax('='));
FALSE
SQL> exec p.l(PLVtkn.is_symbol('='));
TRUE
```

10.2.2.1 Generic keyword checking

The **is_keyword** function is a more general−purpose function. It returns TRUE if the token is a keyword of the type specified by the second argument. The default value for this second parameter is **PLVprs.c_any**, which means that **is_keyword** will return TRUE if the specified token is any kind of keyword.

PLVcase uses the **is_keyword** to determine whether the token should be upper− or lowercase. When applying the UPPER−lower method, it doesn't matter if the token is a builtin function or a syntax element, such as the END statement. All such keywords must be uppercase. Here is the code from the **PLVcase.token** procedure which performs the actual conversion:

```
IF PLVtkn.is_keyword (v_token)
THEN
  v_token := UPPER (v_token);
ELSE
  v_token := LOWER (v_token);
END IF;
```

To keep code volume in PLVtkn to an absolute minimum and eliminate redundancy, I implement all of the "specialized" **is** functions (**is_builtin**, **is_syntax**, etc.) with a call to **is_keyword**, as shown below:

```
FUNCTION is_symbol (token_in IN VARCHAR2)
   RETURN BOOLEAN
IS
BEGIN
   RETURN (is_keyword (token_in, c_symbol));
END;
```

10.2.3 Retrieving Information About a Token

You will use the **get_keyword** procedure to retrieve from the **PLV_token** table all information stored about a particular token. The header of this procedure is:

```
PROCEDURE get_keyword (token_in IN VARCHAR2, kw OUT kw_rectype);
```

You provide the token or string and **get_keyword** returns a PL/SQL record, which is a translated version of the row in the table. The translation generally involves converting string constants to Boolean values. For example, one of the record's fields is named **is_keyword**. The expression assigned to this Boolean field is:

```
kw.is_keyword :=
   kw_rec.token_type IN
      (c_syntax, c_builtin, c_symbol,
       c_sql, c_datatype, c_datadict, c_exception);
```

where **kw_rec** is the cursor−based record into which the **PLV_token** row is fetched.

The anonymous block below shows how to use **get_keyword**. It accepts a string from the user of this script (**plvtkn.tst**), retrieves the information about that string (as a token), and displays some of the data.

```
DECLARE
   my_kw PLVtkn.kw_rectype;
BEGIN
   PLVtkn.get_keyword ('&1', my_kw);
   p.l (my_kw.token_type);
   p.l (my_kw.is_keyword);
END;
/
```

The lines below show this script being executed for the THEN keyword.

```
SQL> @plvtkn.tst then
X
TRUE
```

Special Notes on PLVtkn

PLVtkn recognizes keywords defined in the Oracle Forms tool, but does not recognize reserved words in Oracle Reports, Oracle Graphics, or Oracle Developer/2000 that are not listed specifically as reserved words in Oracle Forms.

PREVIOUS | HOME | NEXT
10.1 PLVprs: Useful String Parsing Extensions | BOOK INDEX | 10.3 PLVprsps: Parsing PL/SQL Strings

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

10.3 PLVprsps: Parsing PL/SQL Strings

The PLVprsps (PL/Vision PaRSe) PL/SQL package builds upon all the other string parsing and analyzing packages to provide easy−to−use, high−level programs to parse PL/SQL strings and programs. The parsed output is passed back into a PL/SQL table you provide in your calls to PLVprs modules. You can then work with the contents of the PL/SQL table as you see fit.

This parsing process separates PL/SQL code into individual atomics, which can then be used for any of the following purposes:

- Analyze the contents of a PL/SQL program. What programs are defined in the package? Are any variables not being used?
- Reformat or pretty−print the PL/SQL program. Once the atomics are separated, you can put them back together however you want and end up with the same program (as long as you preserve the *order* of the atomics).

The PLVprsps package offers several different levels of parsing programs (parse a string, parse a line, parse a program). With PLVprsps, you can also specify precisely which type of language elements you want to return in your parse.

The following sections show how to use the different elements of PLVprsps.

10.3.1 Selecting Token Types for Parsing

One of the users of PLVprsps is the PLVcat package (described in Chapter 18, *PLVcase and PLVcat: Converting and Analyzing PL/SQL Code*), which catalogues the contents and usage of PL/SQL program units. You could ask, for example, to generate a list of those builtins which are used by a particular program. Or you could request to see only those nonkeyword references, which would give you the list of all application−specific identifiers in your code.

The way PLVcat is able to offer this flexibility is by offering you the ability in PLVprsps to request that the output from a call to the parsing programs (`plsql_string` or `module`) return only certain kinds of tokens. The different types currently recognized by PLVprsps are:

- Any keywords
- Builtin functions, procedures, and packages
- Application−specific identifiers (non−keyword identifiers)
-

All tokens

For each of these token types, PLVprsps offers toggles so that you direct the package to keep only the tokens in which you are interested. These are the "keep" and "nokeep" programs. The headers for these programs are:

```
PROCEDURE keep_all;
PROCEDURE keep_kw;
PROCEDURE keep_nonkw;
PROCEDURE keep_bi;

PROCEDURE nokeep_all;
PROCEDURE nokeep_kw;
PROCEDURE nokeep_nonkw;
PROCEDURE nokeep_bi;
```

So if I wanted to keep builtins and non−keywords when I perform my parse, I would issue these two calls:

```
PLVprsps.keep_nonkw;
PLVprsps.keep_bi;
```

All keywords which are not builtins would, therefore, be discarded in the parse. You would not see such atomics as IF and =.

If, on the other hand, I want to obtain all non−keywords, but reject all keywords in the parse, I would call these two programs:

```
PLVprsps.keep_nonkw;
PLVprsps.nokeep_kw;
```

10.3.2 Parsing PL/SQL Code

PLVprsps offers two programs for PL/SQL source code parsing: **`plsql_string`** and **`module`**. The **`plsql_string`** procedure parses the string passed to it. The **`module`** procedure parses all the lines of code for a specified program −− by calling the **`plsql_string`** program for each line in that program. Both of these programs are explained below.

10.3.2.1 plsql_string procedure

The header for **`plsql_string`** is:

```
PROCEDURE plsql_string
   (line_in IN VARCHAR2,
    tokens_out IN OUT PLVtab.vc2000_table,
    num_tokens_out IN OUT INTEGER,
    in_multiline_comment_out IN OUT BOOLEAN);
```

The **`line_in`** argument is the line of code to be parsed. The **`tokens_out`** PL/SQL table holds the distinct tokens found in the line. The **`num_tokens_out`** argument indicates the number of tokens found and in the PL/SQL table. The **`in_multiline_comment_out`** argument returns TRUE if the line of code has initiated or is part of a multiline comment block. You should initialize this IN OUT argument to FALSE to make sure that the parsing is performed as expected.

Comments are not considered atomics for the purposes of parsing. The comment text is parsed but never written to the PL/SQL table. For this reason, all of the following strings will be parsed into precisely the same set of tokens:

```
v_err := 'ORA−' || TO_CHAR (SQLERRM);
v_err := 'ORA−' || /* Concatenate! */ TO_CHAR (SQLERRM);
/* ASSIGN VALUE */ v_err := 'ORA−' || TO_CHAR (SQLERRM);
```

```
*/ v_err := 'ORA-' || TO_CHAR (SQLERRM);
v_err := 'ORA-' || TO_CHAR (SQLERRM); -- end of line
```

In all of these cases, the last argument of **`plsql_string`** will be returned as FALSE. The same set of tokens will be returned with the following string as well, but in this case the last argument of **`plsql_string`** will be returned as TRUE since a multiline comment block has been started:

```
v_err := 'ORA-' || TO_CHAR (SQLERRM); /* big comment coming:
```

10.3.2.2 Script to test plsql_string

The anonymous block shown below (and found in file **`PLVprsps.tst`**) illustrates how to set up variables and then call the **`plsql_string`** program. It also uses **`PLVtab.display`** to easily show the contents of my PL/SQL table of tokens.

```
DECLARE
   full_string VARCHAR2(100)
       := 'v_err := ''ORA-'' || TO_CHAR (SQLERRM)';
   strings PLVtab.vc2000_table;
   num INTEGER := 1;
   incmnt BOOLEAN := FALSE;
BEGIN
   p.l (full_string);
   PLVprsps.plsql_string (full_string, strings, num, incmnt);
   PLVtab.display (strings, num);
   p.l (incmnt);
END;
/
```

Here are the results from an execution of the test script:

```
SQL> start PLVprsps.tst
v_err := 'ORA-' || TO_CHAR (SQLERRM)
Contents of Table
v_err
:=
'ORA-'
||
TO_CHAR
(
SQLERRM
)
FALSE
```

10.3.2.3 module procedure

Use **`PLVprsps.module`** to parse an entire module or PL/SQL program unit. The **`module`** procedure is overloaded in the following two versions:

```
PROCEDURE module
   (module_in IN VARCHAR2 := NULL,
    tokens_out IN OUT PLVtab.vc2000_table,
    num_tokens_out IN OUT INTEGER);

PROCEDURE module
   (tokens_out IN OUT PLVtab.vc2000_table,
    num_tokens_out IN OUT INTEGER);
```

In the first version, you provide the name of the module that you wish to parse and the data structures that will be filled with the parsed tokens: a PL/SQL table and a count of the rows filled. In the second version, you simply supply the PL/SQL table and the variable to hold the number of tokens. This version of **`module`** assumes that you have already set the current object with a call to **`PLVobj.setcurr`**.

The three−argument version of **`module`** simply calls the two−argument version as shown below:

```
PROCEDURE module
   (module_in IN VARCHAR2 := NULL,
    tokens_out IN OUT PLVtab.vc2000_table,
    num_tokens_out IN OUT INTEGER)
IS
BEGIN
   PLVobj.setcurr (module_in);
   module (tokens_out, num_tokens_out);
END;
```

10.3.2.4 Implementing a module parser

The two−argument version of **`PLVprsps.module`** that assumes that the object has already been set is not much more complex than the one you see above. The reason that it is so straightforward is that it relies on the **`plsql_string`** program to parse each line of code. And it gets those lines of code by using the PLVio package. The implementation of **`module`** is shown below:

```
PROCEDURE module
   (tokens_out IN OUT PLVtab.vc2000_table,
    num_tokens_out IN OUT INTEGER)
IS
   srcline PLVio.line_type;
   in_multiline_comment BOOLEAN := FALSE;
BEGIN
   init_table (tokens_out, num_tokens_out);
   PLVio.asrc;
   LOOP
      PLVio.get_line (srcline);
      EXIT WHEN srcline.eof;
      plsql_string
         (srcline.text, tokens_out, num_tokens_out,
          in_multiline_comment);
   END LOOP;
END;
```

In the declaration section of the procedure, I declare a record to hold the line of code extracted with a call to the **`PLVio.get_line`** procedure. I also declare a Boolean variable, which is required for a call to **`plsql_string`**.

In the body of the procedure I first initialize the table that will hold the parsed tokens. Then I request reading of the source code from the ALL_SOURCE data dictionary view. Since I am going to read all of the lines of code in the specified program (assumed to be the current object in PLVobj), my call to **`PLVio.asrc`** does not have any arguments.

Now all I have to do is loop through the lines of code using the **`PLVio.get_line`** procedure. If I have not reached "end of file," I parse that string. Notice that I do not have to manually add the new parsed tokens to my table. The **`plsql_string`** program automatically places the new tokens in the rows after the current value of the **`num_tokens_out`** variable (that is why the third argument of **`plsql_string`** is an IN OUT parameter).

It is amazing how easy it can be to implement complex new functionality when you build upon preexisting elements.

10.3.3 Initializing a Table of Tokens

PLVprsps provides a program to initialize the PL/SQL table and row counter you will use to store parsed tokens. It is called by the **`module`** program; its header is shown below:

```
PROCEDURE init_table
   (tokens_out IN OUT PLVtab.vc2000_table,
    num_tokens_out IN OUT INTEGER);
```

At this time the **`init_table`** procedure does nothing more than assign an empty table to the PL/SQL table argument passed to it and set the row counter to 0.

Why bother building a program like **`init_table`**? There are two good reasons:

- By hiding this logic behind a programmatic interface, I make it easier to enhance this initialization process if the need arises in the future. If I rely on users to prepare the PL/SQL table to receive the parsed tokens, it will be very difficult for them to upgrade their own code to meet the new requirements of my package.

- If the user can call **`init_table`**, she does not have to worry about the specific steps necessary to prepare the PL/SQL table. The individual steps are abstracted into a named action and taken care of by PLVprsps itself.

The **`init_table`** procedure is a gesture of respect for the users of PLVprsps. You don't have time to worry about insignificant details. Furthermore, you are much more likely to use PLVprsps if I offer this kind of feature.

10.3.4 Using PLVprsps

To give you a sense of how PLVprsps breaks apart PL/SQL code, let's see what it does with the following package body:

```
PACKAGE BODY testcase
IS
   PROCEDURE save (string_in IN VARCHAR2)
   IS
      n INTEGER := DBMS_SQL.OPEN_CURSOR;
   BEGIN
      UPDATE PLV_output SET program = string_in;
      IF SQL%ROWCOUNT = 0
      THEN
         INSERT INTO PLV_output VALUES (string_in);
      END IF;
      PLVcmt.perform_commit;
   END;
END testcase;
```

This program has calls to builtins, as well as application−specific identifiers. Now take a look at the following script (found in **`modprs.sql`**). It takes two SQL*Plus arguments: the name of the program to be parsed and the type of tokens to retain and then display:

```
DECLARE
   strings PLVtab.vc2000_table;
   num INTEGER := 1;
BEGIN
   PLVprsps.nokeep_all;
   PLVprsps.keep_&2;
   PLVprsps.module ('&1', strings, num);
   PLVtab.display (strings, num);
END;
/
```

Let's take a look at the output generated by different calls to this script. The first call to **`modprs`** requests the parse to display only builtins. The second call shows all those identifiers which are not keywords in the PL/SQL language.

```
SQL> start modprs b:testcase bi
Contents of Table
DBMS_SQL.OPEN_CURSOR
UPDATE
ROWCOUNT
INSERT

SQL> start modprs b:testcase nonkw
Contents of Table
testcase
save
string_in
n
PLV_output
program
string_in
PLV_output
string_in
PLVcmt.perform_commit
testcase
```

Special Notes on PLVprsps

The current version always reads source code from the ALL_SOURCE data dictionary view. You cannot, in other words, parse PL/SQL code from operating system files (in PL/Vision Lite).

PREVIOUS | HOME | NEXT
10.2 PLVtkn: Managing PL/SQL Tokens | BOOK INDEX | 11. PLVobj: A Packaged Interface to ALL_OBJECTS

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

11. PLVobj: A Packaged Interface to ALL_OBJECTS

Contents:

The PLVobj (PL/Vision OBJect) package provides a programmatic interface to the PL/SQL objects stored in the ALL_OBJECTS data dictionary view. It is used throughout PL/Vision in two ways:

- To parse and manage a "current object," which is composed of the schema, name, and type of the object. The PLVobj package handles the complexity of parsing various versions of the current object specification. It also uses NAME_RESOLVE to locate the object you specify in the data dictionary.
- To easily fetch objects from the ALL_OBJECTS view. With the programmatic interface between you and the ALL_OBJECTS view, you never have to explicitly open, fetch from, or close a cursor against this view in order to retrieve object information. Instead, you call PL/SQL programs which do the job for you.

PLVobj offers some excellent lessons in how to use packages to:

- Hide implementational and data structure details from developers who don't want or need to deal with that level of detail.
- Use the persistent characteristic of packaged variables to implement a current object that can be used in many different programs and circumstances.
- Provide a comprehensive procedural interface to a cursor. This includes the **`loopexec`** program, which simulates a cursor FOR loop against the cursor.

The PLVobj package is not a flashy piece of software. It isn't anything end users or even developer users will ever really see. It is, however, a very useful low−level building−block component for developers who work with this data dictionary view and who may want to build similar interfaces to other predefined views.

11.1 Why PLVobj?

PL/Vision contains a number of utilities which analyze and manipulate the contents of data dictionary views containing PL/SQL code source text. These utilities convert the case of a PL/SQL program, analyze which external programs and package elements a program references, display stored source code, show compiler errors, etc. In each of these cases I needed to take the same or similar actions again and again:

-

Accept a string from the user that specifies the program unit he wants the package to work with. Convert it to the owner−name−type information I need to use when working with the data dictionary views.

- Fetch rows from one or more data dictionary views based on the program unit specified.

I would like to be able to say that as I began writing my first source−related utility I instantly recognized the need to create a package like PLVobj. The truth is that my first read of the situation was that it was very easy to define a cursor against USER_OBJECTS and get what I needed for my package. So I just started hacking away. I built the first version of my program and got it working. And then I started on my next utility. Suddenly I was confronted with having to write the same (or very similar) kind of code again. I was troubled by the redundancy. Still, it was pretty simple stuff, so I went ahead with the duplication of code. I got that second utility to work as well. Then I sent the packages to one of my devoted beta testers. He installed them in a networked environment under a common user and told his developers to try them out.

Neither utility worked. At all. It didn't take too long to figure out why. In my own, intimate development and testing environment, everything existed in the same Oracle account. In the beta environment the utilities were installed in a single account and then shared by all. My naive reliance on the USER_OBJECTS data dictionary view doomed the utilities. I needed instead to use the ALL_OBJECTS view. This meant that I also needed to provide a schema or owner to the cursor. Suddenly I had to perform less−than−trivial enhancements to two different programs.

At this point, I came to my senses. I needed to consolidate all of this logic, all code relating to the objects data dictionary view, into a single location −− a package. I could not afford, in terms of productivity and code quality, to have code redundancy. As you begin to use new data structures or develop a new technique the *first* time, it is sometimes difficult to justify cleaving off the code to its own repository or package. When you get to needing it the second time, however, there should be no excuses. Avoid with fanatical determination any redundancies in your application code.

And so PLVobj was born. Of course, the version I share with you is very different from the first, second, third, and fourth versions of the package. Believe me, it has changed a lot over a four−month period. I seem to come across new complexities every week. (For example, a module name is not always in upper case; you can create program units whose names have lowercase letters if you enclose the name in double quotes.)

The PLVobj package offers functionality in several areas:

- Set and view the "current object" maintained inside the package.
- Access a cursor into the ALL_OBJECTS view, providing a full range of cursor−based functionality through PL/SQL procedures and functions.
- Trace the actions of the package.

The elements available in PLVobj are described in the following sections. Before diving into the programs, however, let's review the ALL_OBJECTS view.

PREVIOUS HOME NEXT
10.3 PLVprsps: Parsing PL/SQL Strings BOOK INDEX 11.2 ALL_OBJECTS View

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8*i* Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

11.2 ALL_OBJECTS View

The main objective of PLVobj is to provide a programmatic interface to the ALL_OBJECTS view. This data dictionary view has the following columns:

```
Name                Null?    Type
------------------- -------- ----
OWNER               NOT NULL VARCHAR2(30)
OBJECT_NAME         NOT NULL VARCHAR2(30)
OBJECT_ID           NOT NULL NUMBER
OBJECT_TYPE                  VARCHAR2(12)
CREATED             NOT NULL DATE
LAST_DDL_TIME       NOT NULL DATE
TIMESTAMP                    VARCHAR2(75)
STATUS                       VARCHAR2(7)
```

It contains a row for every stored code object to which you have execute access (the USER_OBJECTS view contains a row for each stored code object you created). The OWNER is the schema that owns the program unit.

object_name

The name of the object.

object_type

The type of the object, which is one of the following: package, package body, procedure, function, or synonym.

object_id

An internal pointer used to quickly obtain related information about the object in other data dictionary views.

last_ddl_time

Stores the date and timestamp when this object was last compiled into the database.

Status column

Either VALID or INVALID. This column is set by the Oracle Server as it maintains dependencies and performs compiles.

Without a package like PLVobj, every time you wanted to read from this view, you would need to write a SELECT statement in SQL*Plus or, in PL/SQL, define a cursor, and then do the "cursor thing." You would need to understand the complexities of the ALL_OBJECTS view (for example, all names are uppercased unless you created the object with double quotes around the name). You would also need to know how to parse a program name into its full set of components: owner, name, and type.

If several developers in your organization need to do the same thing, you soon have a situation where the same kind of query and similar code is "hard−coded" across your application or utilities. Lots of hours are wasted and the resources required for maintenance of the application multiply.

A better solution is to write the cursor once −− in a package, of course −− and then let all developers reference that cursor. They each do their own opens, fetches, and closes, but the SQL (just about the most volatile part of one's application) is shared. An even better solution, however, is to hide the cursor in the body of the package and build a complete programmatic interface to the cursor. With this approach, you build procedures and functions that perform the cursor operations; a user of the package never has to call native PL/SQL cursor operations. This is the approach I have taken with PLVobj and described in the following sections.

11.2.1 Cursor Into ALL_OBJECTS

At the heart of **PLVobj** (in the package body) is a cursor against the ALL_OBJECTS view. The cursor is defined as follows:

```
CURSOR obj_cur
IS
   SELECT owner, object_name, object_type, status
     FROM ALL_OBJECTS
    WHERE object_name LIKE v_currname
      AND object_type LIKE v_currtype
      AND owner LIKE v_currschema
    ORDER BY owner,
       DECODE (object_type,
          'PACKAGE', 1,
          'PACKAGE BODY', 2,
          'PROCEDURE', 3,
          'FUNCTION', 4,
          'TRIGGER', 5,
          6),
       object_name;
```

Notice that the cursor contains references to three package variables: **v_currschema**, **v_currname**, and **v_currtype**. These three variables make up the current object of **PLVobj** and are discussed in more detail later in this chapter.

Notice that I use the LIKE operator so that you can retrieve multiple objects from a single schema, or even multiple schemas. Furthermore, since my PL/SQL development is PL/SQL and package−centric, I use a DECODE statement to bring those up first in the ordering system.

Again, since this cursor is defined in the package body, users of **PLVobj** cannot directly OPEN, fetch from, or CLOSE the cursor. All of these actions must be taken through procedures which are described in the next section.

PREVIOUS | HOME | NEXT
11.1 Why PLVobj? | BOOK INDEX | 11.3 Setting the Current Object

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

11.3 Setting the Current Object

PLVobj provides a number of different programs to set and change the current object of **PLVobj**. The current object of **PLVobj** is defined by three private package variables:

v_currschema
: The owner of the object(s).

v_currname
: The name of the object(s). The name can be wildcarded, by including % in the name specified.

v_currtype
: The type of the object(s). The type can be wildcarded as well, again by including % in the type specified.

Since the above elements are private variables, a user of **PLVobj** will never see or reference these variables directly. Instead, I provide the following set of procedures and functions to maintain these variables (I call this layer of code which surrounds variables "get and set" routines): **setcurr**, **set_name**, **set_type**, and **set_schema**.

The **setcurr** program calls the other set programs. Its header is:

```
PROCEDURE setcurr
   (name_in IN VARCHAR2, type_in IN VARCHAR2 := NULL);
```

The first argument is the module name, the second, optional argument is the module type. While the first argument is called a "name" argument, you can actually in this single argument supply the name, type, and schema. **PLVobj** makes it as easy as possible to supply this information. All of the following formats for the **name_in** argument are acceptable:

name
: An unqualified identifier. In this case, PLVobj determines the type of the object from the data dictionary. This type will be unambiguous if name refers to a procedure or function. If a package, then the type could be either PACKAGE or PACKAGE BODY. PLVobj defaults to PACKAGE.

schema.name
: The name of the object is qualified by the schema name. You will need to specify the schema name if you want to convert the case of a program owned by another user.

type:name
: Both the type and the name are specified, separated by a colon. Valid type strings for each type of program unit are shown below.

type:schema.name

In this case, the user has specified all three elements of a program unit: the type of the program, the owner, and the name.

In fact, the second argument is optional because **PLVobj** allows you to concatenate the type onto the name argument.

The following table shows different ways of specifying programs and the resulting **PLVobj** current object values. In this table, **pkg** is the name of a package, **func** the name of a function, and **proc** the name of a procedure. The **setcurr** program is executed from the PLV account.

Call to setcurr	Schema	Program Type	Program Name
`PLVobj.setcurr ('proc');`	PLV	PROCEDURE	`proc`
`PLVobj.setcurr ('func');`	PLV	FUNCTION	`func`
`PLVobj.setcurr ('pkg');`	PLV	PACKAGE	`pkg`
`PLVobj.setcurr ('b:pkg');`	PLV	PACKAGE BODY	`pkg`
`PLVobj.setcurr ('body:pkg');`	PLV	PACKAGE BODY	`pkg`
`PLVobj.setcurr ('s:SCOTT.empmaint');`	SCOTT	PACKAGE	`empmaint`
`PLVobj.setcurr ('%:plv%');`	PLV	ALL TYPES	`Like PLV%`
`PLVobj.setcurr ('s:scott.% ');`	SCOTT	PACKAGE	`All present`

The above table assumes, by the way, that the PLV account has execute authority on SCOTT's **empmaint** package.

Notice that when I specify a function or procedure, I do not have to provide the type at all. There can only be one object of a given name in a schema, and the **object_type** is therefore unambiguously set to PROCEDURE. On the other hand, when I am working with packages, the situation is more ambiguous. A package can have up to two objects: the specification and the body. So if you provide a package name but do not supply a type, PLVobj will set the current type to PACKAGE. If you want to set the current object to a package body, you must supply a valid object type or object type abbreviation.

Here are the valid options for object type:

Program Type	Valid Entries for Type in Call to setcurr
Package Specification	**S PS SPEC** or **SPECIFICATION**
Package Body	**B PB BODY** or **PACKAGE BODY**
Procedure	**P PROC** or **PROCEDURE**
Function	**F FUNC** or **FUNCTION**

So I can set the current object to the PACKAGE BODY of the **testcase** package with any of the following calls to **setcurr**:

```
PLVobj.setcurr ('b:testcase');
PLVobj.setcurr ('pb:testcase');
PLVobj.setcurr ('body:testcase');
PLVobj.setcurr ('package body:testcase');
```

> ***NOTE:*** The **setcurr** program relies on DBMS_UTILITY.NAME_RESOLVE to uncover all the components of a non−wildcarded object entry. This builtin only returns non−NULL values for PL/SQL stored code elements. Consequently, you cannot use **setcurr** to set the current object to non−PL/SQL elements such as tables and indexes. Instead, you will need to call the individual set programs explored in the next section.

You can use the **PLVobj.setcurr** program to convert your entry and set the current object accordingly. In some cases, however, you may want simply to change the current object type. Or you may want to take advantage of the parsing and conversion algorithms of PLVobj without actually changing the current object. To provide this flexibility, PLVobj offers a number of additional programs which are explained below.

11.3.1 Setting Individual Elements of Current Object

You can change the current schema, name, or type independently with the set programs. You can also retrieve the current object values with corresponding functions. The "get and set" programs are shown below:

```
PROCEDURE set_schema (schema_in IN VARCHAR2 := USER);
FUNCTION currschema RETURN VARCHAR2;
PROCEDURE set_type (type_in IN VARCHAR2);
FUNCTION currtype RETURN VARCHAR2;

PROCEDURE set_name (name_in IN VARCHAR2);
FUNCTION currname RETURN VARCHAR2;
```

I use these programs in the **PLVvu.err** procedure, which displays compile errors for the specified program. The main body of **err** is shown below:

```
PLVobj.setcurr (name_in);
IF PLVobj.currtype = PLVobj.c_package AND
   INSTR (name_in, ':') = 0
THEN
   /* Show errors for package spec, then body. */
   show_errors;
   PLVobj.set_type (PLVobj.c_package_body);
   show_errors (TRUE);
ELSE
   show_errors (TRUE);
END IF;
```

Translation: I call the **setcurr** procedure to set the current object. Then I call **currtype** to see if the program type is PACKAGE. If it is, I need to check to see if the developer has requested to see errors for just the package specification or both specification and body. If both were requested (there is no colon in the name, therefore no type specified), then I show errors for the specification, explicitly set the type to PACKAGE BODY −− overriding the existing value −− and then show errors for the body.

11.3.2 Converting the Program Name

All the logic to convert a string into the separate components of schema, name, and type are handled by the **convobj** program, whose header is shown below:

```
PROCEDURE convobj
   (name_inout IN OUT VARCHAR2,
    type_inout IN OUT VARCHAR2,
    schema_inout IN OUT VARCHAR2);
```

All three arguments of **convobj** are IN OUT parameters, so that you can provide a value and also have a value sent back for each of the components of an object.

The logic inside **convobj** is complex. If the name or type passed in to **convobj** contains a wildcard, those wildcarded strings are returned (in their separate components) by the procedure. If, on the other hand, no wildcards are present, **convobj** relies on the DBMS_UTILITY.NAME_RESOLVE builtin procedure to automatically resolve the program name into its individual components.

NAME_RESOLVE resolves the specified object name into the owner or schema, first name, second name (if in a package), program type, and database link if any. The program type is one of the following numbers:

5
(synonym)

7
(procedure)

8
(function)

9
(package)

NAME_RESOLVE is so useful because it automatically determines the appropriate schema of the named element you pass in. Notice that NAME_RESOLVE does not distinguish between a package body and its specification (they both have the same name, so that would be something of a challenge). The **`nameres.sql`** script in the *plvision\use* subdirectory provides an easy way to call and see the results from DBMS_UTILITY.NAME_RESOLVE.

You can call **`convobj`** when you want to convert a name to its different components without changing the current object in the PLVobj package. I do this, for example, in the **`setcase.sql`** script, which provides an easy−to−use frontend to PLVcase for case conversion of your code. I call **`PLVobj.convobj`** even before I call any PLVcase programs because I want to display the program you have just requested for conversion. This code is shown below:

```
PLVobj.convobj (modname, modtype, modschema);
modstring := modtype || ' ' || modname;
p.l ('========================');
p.l ('PL/Vision Case Conversion');
p.l ('========================');
p.l ('Converting ' || modstring || '..');
```

The PLVobj package also offers a **`convert_type`** procedure, which encapsulates the logic by which it converts any number of different abbreviations and strings to a valid object type for the ALL_OBJECTS view. The header for **`convert_type`** is as follows:

```
PROCEDURE convert_type (type_inout IN OUT VARCHAR2);
```

You pass it a string and it changes that screen to a valid type.

11.3.3 Displaying the Current Object

PLVobj provides the **`showcurr`** procedure to display the current object. Its header is:

```
PROCEDURE showcurr (show_header_in IN BOOLEAN := TRUE);
```

You can display the current object with a header (the default) or without, in which case you will simply see the schema, name, and type (note that this full name is constructed with a call to the **`fullname`** function, also available for your use). Some examples follow:

```
SQL> exec PLVobj.set_name ('PLVctlg');
SQL> exec PLVobj.set_type ('table);
SQL> exec PLVobj.showcurr; −− Displays a header
Schema.Name.Type
−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
PLV.PLVCTLG.TABLE
```

```
SQL> exec PLVobj.setcurr ('PLVio');
SQL> exec PLVobj.showcurr (FALSE); -- Suppresses header
PLV.PLVIO.PACKAGE
```

The **`showobj1.sql`** SQL*Plus script uses **`showcurr`** to display all the objects specified by your input. This script is described below in the section called "Accessing ALL_OBJECTS."

11.3.4 Saving and Restoring the Current Object

PLVobj provides programs to both save the current object and restore it from the last save. The headers for these programs are shown below:

```
PROCEDURE savecurr;
PROCEDURE restcurr;
```

You will want to use these programs if you are using PLVobj (in particular, the current object of PLVobj) more than once in a given program execution. Suppose, for example, that you have nested loops. In the outer loop, you call **`PLVobj.setcurr`** to scan through a set of program units. Inside the inner loop, you need to use **`PLVobj.setcurr`** to change the focus of activity to another object. When you are done with the inner loop execution, however, you will want to set the current object back to the outer loop values.

PL/Vision runs into this scenario because of the extensive work manipulating PL/SQL code objects.

PREVIOUS | HOME | NEXT
11.2 ALL_OBJECTS View | BOOK INDEX | 11.4 Accessing ALL_OBJECTS

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

11.4 Accessing ALL_OBJECTS

Once you have set the current object in PLVobj (with either a call to **`setcurr`** or calls to the individual set programs), you can open, fetch from, and close the PLVobj cursor.

11.4.1 Opening and Closing the PLVobj Cursor

To open the cursor, you call the **`open_objects`** procedure, defined as follows:

```
PROCEDURE open_objects;
```

This procedure first checks to see if the cursor is already open and, if not, takes that action. The implementation of **`open_objects`** is shown below:

```
PROCEDURE open_objects IS
BEGIN
   IF obj_cur%ISOPEN
   THEN
      NULL;
   ELSE
      OPEN obj_cur;
   END IF;
END;
```

When you are done fetching from the cursor, you may close it with the following procedure:

```
PROCEDURE close_objects;
```

whose implementation makes sure that the cursor is actually open before attempting to close the cursor:

```
PROCEDURE close_objects IS
BEGIN
   IF obj_cur%ISOPEN
   THEN
      CLOSE obj_cur;
   END IF;
END;
```

11.4.2 Fetching from the PLVobj Cursor

Once the cursor is open, you will usually want to fetch rows from the result set. You do this with the **`fetch_object`** procedure, which is overloaded as follows:

```
PROCEDURE fetch_object;
PROCEDURE fetch_object (name_out OUT VARCHAR2, type_out OUT VARCHAR2);
```

If you call **`fetch_objects`** without providing any OUT arguments, the name and type will be passed directly into the current object variables, **`v_currname`** and **`v_currtype`**.

If, on the other hand, you provide two return values in the call to **`fetch_object`**, the current object will remain unchanged and you will be able to do what you want with the fetched values. The call to **`fetch_object`** without arguments is, therefore, equivalent to:

```
PLVobj.fetch_object (v_name, v_type);
PLVobj.setcurr (v_name, v_type);
```

11.4.3 Checking for Last Record

To determine when you have fetched all of the records from the cursor, use the **`more_objects`** function, whose header is:

```
   FUNCTION more_objects RETURN BOOLEAN;
```

This function returns TRUE when the **`obj_cur`** is open and when **`obj_cur%FOUND`** returns TRUE. In all other cases, the function returns FALSE (including when the PLVobj cursor is not even open).

11.4.4 Showing Objects with PLVobj

To see how all of these different cursor−oriented programs can be utilized, consider the following script (stored in **`showobj1.sql`**).

```
DECLARE
   first_one BOOLEAN := TRUE;
BEGIN
   PLVobj.setcurr ('&1');
   PLVobj.open_objects;
   LOOP
      PLVobj.fetch_object;
      EXIT WHEN NOT PLVobj.more_objects;
      PLVobj.showcurr (first_one);
      first_one := FALSE;
   END LOOP;
   PLVobj.close_objects;
END;
/
```

It sets the current object to the value passed in at the SQL*Plus command line. It then opens and fetches from the PLVobj cursor, exiting when **`more_objects`** returns FALSE. Finally, it closes the PLVobj cursor. This cursor close action is truly required. The PLVobj cursor is not declared in the scope of the anonymous block; instead, it is defined in the package body. After you open it, it will remain open for the duration of your session, unless you close it explicitly.

In the following example of a call to **`showobj1.sql`**, I ask to see all the package specifications in my account whose names start with "PLVC". I see that I have four packages.

```
SQL> start showobj1 s:PLVc%
Schema.Name.Type
PLV.PLVCASE.PACKAGE
PLV.PLVCAT.PACKAGE
PLV.PLVCHR.PACKAGE
PLV.PLVCMT.PACKAGE
```

If you are not working in SQL*Plus, you can easily convert the **`showobj1.sql`** script into a procedure as follows:

```
CREATE OR REPLACE PROCEDURE showobj (obj_in IN VARCHAR2)
IS
   first_one BOOLEAN := TRUE;
```

```
BEGIN
   PLVobj.setcurr (obj_in);
   PLVobj.open_objects;
   LOOP
      PLVobj.fetch_object;
      EXIT WHEN NOT PLVobj.more_objects;
      PLVobj.showcurr (first_one);
      first_one := FALSE;
   END LOOP;
   PLVobj.close_objects;
END;
/
```

PREVIOUS 11.3 Setting the Current Object | HOME BOOK INDEX | NEXT 11.5 Binding Objects to a Dynamic Cursor

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

11.5 Binding Objects to a Dynamic Cursor

PLVobj provides the **bindobj** procedure to make it easier for you to utilize the PLVobj current object inside dynamic SQL. This program can be used when you have placed bind variables in your dynamically constructed cursor that correspond to one or more of the elements of the current object.

The header for **bindobj** is as follows:

```
PROCEDURE bindobj
   (cur_in IN INTEGER,
    name_col_in IN VARCHAR2 := 'name',
    type_col_in IN VARCHAR2 := 'type',
    schema_col_in IN VARCHAR2 := 'owner');
```

The first, and only required, argument is the handle to the DBMS_SQL cursor handle. The other parameters provide the strings which are the placeholders in the string that was parsed for that cursor handle. The default values for these placeholders correspond to the names of the columns in the ALL_SOURCE data dictionary view.

11.5.1 Specifying Which Binds Occur

The **bindobj** procedure will only call BIND_VARIABLE for those placeholders for which a non−NULL column name is provided. For example, in the following call to **bindobj**, BIND_VARIABLE will only be executed for the name placeholder.

```
PLVobj.bindobj (cur_handle, 'objname', NULL, NULL);
```

Notice that since the default values for these column names are not NULL, you must explicitly pass a NULL value in to **bindobj** in order to turn off a binding for that placeholder (if you do not, DBMS_SQL will raise an exception). If you only want to turn off one of the trailing bind operations (such as for the schema), while leaving the earlier column names with their defaults, you can use named notation to specify an override for just that column as shown below:

```
PLVobj.bindobj (cur_handle, schema_col_in => NULL);
```

11.5.2 Using bindobj

The **bindobj** procedure comes in handy when you are using PLVobj to manage a current object, but you are not using PLVobj to query records from the ALL_OBJECTS view. You might, as does PL/Vision, want to read information from another data dictionary view that also contains object−related information, such as USER_SOURCE or USER_ERRORS.

I'll take you through a simple example of how to use **bindobj**. The script below (found in **inline.sql**) uses PLVobj and PLVdyn to display the line numbers of the stored source code which contains the specified string. With this script you answer such questions as: "How many (and which) lines of code in the PLVio package use the SUBSTR function?" Here, in fact, is the answer to that question:

```
SQL> start inline b:PLVprs SUBSTR
Lines with SUBSTR in PLV.PLVPRS.PACKAGE BODY
54
63
76
141
144
219
242
282
303
306
312
315
377
```

And here is the **`inline.sql`** script:

```
SET VERIFY OFF
DECLARE
   v_sql VARCHAR2(2000) :=
     'SELECT line FROM user_source ' ||
     ' WHERE name = :name ' ||
     '   AND type = :type ' ||
     '   AND INSTR (text, ''&2'') > 0' ||
     ' ORDER BY line';
   v_line INTEGER;
   cur INTEGER;
BEGIN
   PLVobj.setcurr ('&1');

   cur := PLVdyn.open_and_parse (v_sql);
   DBMS_SQL.DEFINE_COLUMN (cur, 1, v_line);
   PLVobj.bindobj (cur, schema_col_in => NULL);
   PLVdyn.execute (cur);

   p.l ('Lines with &2 in ' || PLVobj.fullname);
   LOOP
      EXIT WHEN DBMS_SQL.FETCH_ROWS (cur) = 0;
      DBMS_SQL.COLUMN_VALUE (cur, 1, v_line);
      p.l (v_line);
   END LOOP;
END;
/
```

In the **`inline.sql`** script, I use **`PLVobj.setcurr`** to set the current object (passed in as the first argument to the script). I then perform several steps of dynamic SQL to open and parse the cursor and then define the single column for the SELECT statement. Before I can execute the cursor, I need to provide bind values for the **`:name`** and **`:type`** placeholders.

Since I have called **`PLVobj.setcurr`**, I can take advantage of the current object by calling the **`bindobj`** procedure. It automatically binds the name and type of the current object to my locally defined dynamic SQL statement. Since I am working with USER_SOURCE, I specify in my call to **`bindobj`** that I do not have any schema placeholder to be bound.

Following the bind, I execute and then loop through all the rows in the result set, displaying the line number.

11.5.3 Using bindobj in PL/Vision

The PLVio package contains a private procedure that makes use of **`PLVobj.bindobj`**. The **`prepsrc`** procedure prepares the source when it is a database table. This preparation phase involves calling the necessary dynamic SQL programs to define and execute a cursor against the table. Here is a simplified version of **`prepsrc`**:

```
PROCEDURE prepsrc (cur_in IN OUT INTEGER)
IS
   v_namecol PLV.plsql_identifier%TYPE := srcrep.name_col;
   v_typecol PLV.plsql_identifier%TYPE := srcrep.type_col;
   v_schemacol PLV.plsql_identifier%TYPE := srcrep.schema_col;
BEGIN
   cur_in := PLVdyn.open_and_parse (srcselect);

  /* Check to see if placeholders need to be bound. */

   IF INSTR (srcselect, ':' || v_namecol) = 0
   THEN
      v_namecol := NULL;
   END IF;

   IF INSTR (srcselect, ':' || v_typecol) = 0
   THEN
       v_typecol := NULL;
    END IF;

   IF INSTR (srcselect, ':' || v_schemacol) = 0
   THEN
       v_schemacol := NULL;
    END IF;

   PLVobj.bindobj (cur_in, v_namecol, v_typecol, v_schemacol);

   PLVdyn.execute (cur_in);

END prepsrc;
```

Translation: Use the PLVdyn (PL/Vision DYNamic SQL) package to open and parse a select statement which has already been constructed (and is returned by the call to the function **`srcselect`**). Since the user of PLVio can modify the contents of the SELECT statement, I then use IF statements to check to see whether the standard name and type placeholders are in the dynamic SQL string. I use the INSTR builtin combined with the default column names to see if placeholders for name, type, or schema appear in the SELECT statement. If not, I set the corresponding column names to NULL.

Next, I call the **`bindobj`** procedure to bind this cursor for the current object (**`PLVmod.currschema`**, **`PLVmod.currname`**, and **`PLVmod.currtype`**), but only for those placeholders that are present. At the end of **`prepsrc`**, I execute the cursor using PLVdyn.

PREVIOUS | HOME | NEXT
11.4 Accessing ALL_OBJECTS | BOOK INDEX | 11.6 Populating a PL/SQL Table with Object Names

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

11.6 Populating a PL/SQL Table with Object Names

PLVobj provides a procedure to transfer the names of all objects identified by user input from the view into a PL/SQL table. This **vu2pstab** procedure's header is as follows:

```
PROCEDURE vu2pstab
   (module_in IN VARCHAR2,
    table_out OUT PLVtab.vc2000_table,
    num_objects_inout IN OUT INTEGER);
```

The first argument, **module_in**, is the module specification. This can be a single module or, with wildcarding characters, a set of objects. The second argument, **table_out**, is the PL/SQL table that will hold the names of all identified objects. The final argument, **num_objects_inout**, contains the number of rows populated in the PL/SQL table (starting from row 1).

Use the **vu2pstab** procedure when you want to create a list of the objects which you can then use as the basis for one or more passes through the list to perform actions against the objects. This can be particularly important when you want to make use of different elements of PL/Vision which rely on PLVobj and a current object for processing. If these packages are nested, the outer loop that uses PLVobj can be affected or overridden by the inner usage.

The script **showobj1.sql** shown in a previous section used a simple loop to retrieve and display each of the objects specified by the SQL*Plus argument. That loop can be replaced by a call to **vu2pstab** and a call to **PLVtab.display** to show the contents of the table. This version of "show objects" (stored in the file **showobj2.sql**) is shown below:

```
DECLARE
   objects PLVtab.vc2000_table;
   numobjs INTEGER;
BEGIN
   PLVobj.vu2pstab ('&1', objects, numobjs);
   PLVtab.display (objects, numobjs);
END;
/
```

This is far less code than was required by the first version; the open, fetch, and close steps of the cursor manipulation are hidden behind the **vu2pstab** program. In this way, **PLVobj.vu2pstab** offers some of the flavor and code savings of a cursor FOR loop. The **loopexec** procedure covered in the next section, on the other hand, offers an even closer resemblance to the cursor FOR loop and is a very entertaining application of dynamic PL/SQL code execution.

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

11.7 A Programmatic Cursor FOR Loop

The **loopexec** procedure provides the procedural equivalent of a cursor FOR loop. Its header is shown below:

```
PROCEDURE loopexec
   (module_in IN VARCHAR2,
    exec_in IN VARCHAR2 := c_show_object,
    placeholder_in IN VARCHAR2 := c_leph,
    name_format_in IN VARCHAR2 := c_modspec);
```

The **loopexec** procedure executes the line of code found in the **exec_in** argument for all of the modules specified by the **module_in** argument. If the **module_in** string does not have any wildcard characters, then it will apply the **exec_in** command to the single program only.

The default value for the executable statement is **c_show_object**, a constant defined in the package specification as follows:

```
c_show_object CONSTANT VARCHAR2(100) := 'p.l (:rowobj)';
```

where **rowobj** is the placeholder for the object identified by the current row fetched from the PLVobj cursor. The default action is, therefore, to display the name of the current object.

The **placeholder_in** argument tells **loopexec** which string will serve as a placeholder in the execution string (this placeholder is similar to the dynamic SQL bind variable placeholder). The default is defined in the PLVobj constant, **c_leph**, as follows:

```
c_leph CONSTANT VARCHAR2(10) := 'rowobj';
```

You can, however, override this value with your own string (an example of this process is shown in the next section).[1]

[1] For curious readers, the **leph** stands for "LoopExec PlaceHolder."

The **name_format_in** argument specifies the form that the current object string should take when constructed by PLVobj. The options for the format are:

Format Constant	Format of Object String
c_modspec	TYPE:SCHEMA.NAME
c_modname	SCHEMA.NAME

The **c_modspec** format is useful when the current object is to be passed to another program that supports the PLVobj format for specifying objects (particularly in PL/Vision utilities). The **c_modname** option structures the name so that it is a valid program unit name in PL/SQL.

11.7.1 Some Simple Applications of loopexec

The default statement executed by **PLVobj.loopexec** requests that **loopexec** display the name of the current object. So if I want to display all of the objects located with the string **s:PLVc%**, I would simply enter:

```
SQL> exec PLVobj.loopexec ('s:PLVc%');
PACKAGE:PLV.PLVCASE
PACKAGE:PLV.PLVCAT
PACKAGE:PLV.PLVCHR
PACKAGE:PLV.PLVCMT
```

and would discover that I have four package specifications whose names start with PLVC. To obtain this information, I did not have to write a loop against the ALL_OBJECTS cursor. Neither did I have to call the various **PLVobj** cursor management programs like **fetch_object**. Instead, I simply told **PLVobj**: For every object identified, execute this code. It is, in other words, a programmatically defined cursor FOR loop! See what I mean about PL/SQL being fun?

Now suppose that I want to do something besides display the objects. Let's see how to use **loopexec** to generate a set of DESCRIBE commands for use in SQL*Plus to show the call interface to stored code. The format for this command in SQL*Plus is:

```
SQL> desc prog
```

where **prog** is the name of a PL/SQL program, either a function or a procedure (standalone or packaged). So if my PL/SQL program is going to generate these commands, I would have to execute something like this:

```
p.l ('DESC ' || current_object);
```

where **current_object** is a variable containing the current object string. In this scenario, however, I am working with dynamic PL/SQL; **loopexec** does not know in advance which statement I want to execute. So I need to convert this command into a string that will be evaluated *into* the proper PL/SQL command. In particular, I must double all single quotes and give **loopexec** a way to find my reference to the current object. I do this through the use of a placeholder string.

This approach is shown in the following script (stored in file **gendesc.sql**):

```
BEGIN
   PLVobj.loopexec
      ('&1', 'p.l (''DESC '' || :XX)', 'XX', PLVobj.c_modname);
END;
/
```

In this call to **loopexec**, I provide a value for every argument! The first value is actually a SQL*Plus substitution parameter containing the specification of the program unit(s) for which I want to generate a DESC command. The second argument is the dynamic PL/SQL version of the call to **p.l**, which outputs a DESC command. Notice the double quotes around DESC and the hard−coding of a concatenation of the **XX** placeholder. The third argument (**XX**) tells **loopexec** to replace any occurrence of **:XX** in the command with the current object. Finally, the fourth argument requests that the current object string be returned as a valid PL/SQL object name (the DESC command doesn't know about the *type:schema.name* syntax of PL/Vision).

I execute **gendesc** below to create DESCRIBE commands for all procedures in the PLV schema.

```
SQL>  @gendesc p:%
DESC PLV.CREATE_INDEX
DESC PLV.MODVALS
DESC PLV.MORE
```

```
        DESC PLV.PLVHELP
        DESC PLV.PLVSTOP
        DESC PLV.SHOWEMPS
        DESC PLV.SHOWERR
        DESC PLV.SHOWUSER
```

I can then cut and paste these commands into a file (or use the SPOOL command) and execute them.

You might still be looking at the arguments I passed to **PLVobj.loopexec** and wondering what the heck that all means and, more importantly, how you could ever figure out how to use **loopexec** properly. So let's now turn our attention to the task of constructing an execution string for the **loopexec** procedure.

11.7.2 Constructing the Execution String

The **loopexec** procedure uses dynamic PL/SQL (via the PL/Vision **PLVdyn** package and, as a result, the builtin DBMS_SQL package) to execute the string you pass to it. For this to work properly, you must build a string which evaluates to a valid PL/SQL command. This task can become very complicated when you need to include single quote marks and especially when you want your executed code to operate on the current object (which is, after all, the main reason you would use **loopexec**). To work with the current object fetched from the PLVobj cursor, **loopexec** needs to bind the current object into the dynamic PL/SQL string.

To understand how to deal with these issues, let's start by looking more closely at the default action. This code string is contained in the packaged constant, **c_show_object**, which is the following string:

```
        c_show_object CONSTANT VARCHAR2(100) := 'p.l (:rowobj)';
```

In PLVobj terminology, the string **:rowobj** is the placeholder for the current object. This is the default placeholder string and is defined in a package−level constant shown below:

```
        c_leph CONSTANT VARCHAR2(10) := 'rowobj';
```

The **p.l** procedure, as you should be well aware by now, displays output to the screen. So this command says: "Display the current object." When **loopexec** prepares to execute this simple command, it replaces all occurrences of the string **:rowobj** with the variable containing the current object string (in the form *type:schema.name*). It then passes this string to the **open_and_parse** function of PLVdyn and immediately executes the PL/SQL program contained in the string.

When you construct your own strings to pass to **loopexec**, you can use the default placeholder string or you can specify your own string. You saw in the last section how I used my own placeholder, **XX**, to direct **loopexec** to perform the right substitution. Now let's look at how **PLVcase** uses **loopexec** to convert the case of multiple programs to demonstrate use of the default placeholder. The full body of the **PLVcase.modules** procedure is shown below:

```
        PLVobj.loopexec
           (module_spec_in,
            'PLVcase.module(' || PLVobj.c_leph || ', PLVcase.c_usecor,
               FALSE)');
```

As you can see, it consists of a single line: a call to the **loopexec** program. This call contains only two arguments, so the default values will be used for the last two arguments (the placeholder string and the string format). The line of code executed by **loopexec** is a call to **PLVcase.module** program, which converts the case of a single PL/SQL program. Suppose that I am converting the **employee_maint** package. I would then want this string executed by **loopexec**:

```
        PLVcase.module ('employee_maint', PLVcase.c_usecor, FALSE);
```

Since I am passing in a variable containing the package name, however, my call to **`PLVcase.module`** would look more like this:

```
PLVcase.module (v_currobj, PLVcase.c_usecor, FALSE);
```

Now, it is theoretically possible for me to find out the specific string used by PLVobj for its placeholder replacement (you have already seen it: **`:rowobj`**). This is an example, however, of dangerous knowledge. In this situation, what I know could hurt me. What if I hard−code the **`rowobj`** string into my calls to **`loopexec`** and then somewhere down the line, PLVobj is changed and a new string is used? Yikes! Lots of broken code.

11.7.3 Using the Predefined Placeholder

A better approach is to reference the placeholder string by a named constant, rather than a literal value. This constant is provided by the **`PLVobj.c_leph`** constant. In this approach, when I call **`PLVcase.module`**, I would concatenate this constant into my command string wherever the current object variable appeared in the last example:

```
'PLVcase.module(' || PLVobj.c_leph || ', PLVcase.c_usecor, FALSE)'
```

When passed to **`loopexec`**, this string will be executed for every object retrieved from the cursor. And for each of those objects, the placeholder string will be replaced by the object name and the dynamic PL/SQL code then executed for that object.

The PLVcat package also calls the **`loopexec`** procedure in its **`modules`** program to catalogue multiple programs. In this case, when I pass the current object to the **`PLVcat.module`** I only want to pass the SCHEMA.NAME portion of the current object. Consequently, I request the alternative name format:

```
PLVobj.loopexec
   ('s:' || module_in,
    'PLVcat.module(' || PLVobj.c_leph || ')',
    name_format_in => PLVobj.c_modname);
```

If you do not want to bother with making reference to the PLVobj constant for the placeholder value, you can specify another of your own design. For example, I could recode my call to **`loopexec`** in PLVcat to this:

```
PLVobj.loopexec
   ('s:' || module_in,
    'PLVcat.module(:XX)',
    'XX',
    PLVobj.c_modname);
```

> ***NOTE:*** When you execute dynamic PL/SQL as practiced by PLVobj, you can only reference global data structures and programs. You can, in other words, only reference elements defined in a package specification to which you have access. A PL/SQL block that is executed dynamically is not a nested block in the current program; it is treated as a standalone block. That is why when **`PLVcase.modules`** calls **`loopexec`** in this section's example, it includes the package name PLVcase in its reference to module and the constant, **`c_usecor`**.

Now I have hard−coded the **`XX`** placeholder string into my execution string, but I also inform **`loopexec`** that **`XX`** is the new placeholder string. I have, therefore, established internal consistency. Even if the value of the **`c_leph`** constant changes, my code will not be affected.

As you can see, **`PLVobj.loopexec`** offers a tremendous amount of flexibility and potential. That's what happens when you leverage dynamic PL/SQL code execution and you take the time to build an interface through which users can tweak a utility's behavior. Take some time to play around with **`loopexec`** and all its parameter variations. You will benefit not only in your ability to take advantage of PLVobj, but also in your efforts with DBMS_SQL. It will be an investment of time richly rewarded.

11.7.4 Applying loopexec in PL/Vision

I found that in a number of PL/Vision packages I wanted to execute a certain piece of functionality (convert the case, build a catalogue, etc.) for more than one object at a time. For example, I might want to catalogue all the external references in all packages with names like **PLV%**. The first time I did this, for PLVcase, I made use of my carefully constructed cursor−related programs to come up with a loop like this:

```
PROCEDURE modules
  (module_spec_in IN VARCHAR2 := NULL)
IS
   objects PLVtab.vc2000_table;
   numobj INTEGER := 0;
BEGIN
   PLVobj.setcurr (module_spec_in);

   PLVobj.open_objects;

   LOOP
      PLVobj.fetch_object;
      EXIT WHEN NOT PLVobj.more_objects;
      numobj := numobj + 1;
      objects (numobj) :=
         PLVobj.currtype || ':' ||
         PLVobj.currschema || '.' ||
         PLVobj.currname;
   END LOOP;

   PLVobj.close_objects;

   FOR objind IN 1 .. numobj
   LOOP
      module (objects (objind), c_usecor, FALSE);
   END LOOP;

   save_program;
END;
```

In this procedure, I loop through the cursor, loading up a PL/SQL table with the selected objects. Then I use a cursor FOR loop to convert the case of each program found in that table. I was proud of the way I was able to quickly apply my different, high−level elements of PL/Vision to come up with rich functionality. But then I got to the PLVcat package and I wanted to do the same thing there as well. Suddenly my elegant set of loops seemed like an awful lot of code to repeat. So what did I do? I built the **vu2pstab** procedure of PLVobj and was able to shrink down the **PLVcase.modules** program to nothing more than:

```
PROCEDURE modules
  (module_spec_in IN VARCHAR2 := NULL)
IS
   objects PLVtab.vc2000_table;
   numobj INTEGER := 0;
BEGIN
   PLVobj.vu2pstab (module_spec_in, objects, numobj);

   FOR objind IN 1 .. numobj
   LOOP
      module (objects (objind), c_usecor, FALSE);
   END LOOP;
   save_program;
END;
```

And *that* was a very satisfying reduction of code and simultaneous abstraction of the process. Yet I still found myself repeating that FOR loop again and again, the only difference being the line of code that was executed inside the loop. This repetition brought me to another realization: Maybe I could use the PLVdyn package to dynamically construct and execute the body of the FOR loop. If that were true, then I could hide all of these details behind a single procedure call.

With the **`PLVobj.loopexec`** procedure, such a consolidation is possible. The final implementation of **`PLVcat.modules`** does in fact consist of a single line of code as shown below:

```
PROCEDURE modules (module_in IN VARCHAR2) IS
BEGIN
   PLVobj.loopexec
      ('s:' || notype (module_in),
       'PLVcat.module(' || PLVobj.c_leph || ')',
       name_format_in => PLVobj.c_modname);
END;
```

It uses the default placeholder string in the call to **`PLVcat.module`**. Since that placeholder argument does not need to be specified, it uses named notation to make sure that the module name format is used.

It was possible in this way to shrink a 15−line program body down to just one line. In the process, I switched from a procedural mode of coding to a declarative style. By calling **`loopexec`**, I simply describe the action I want and let the underlying engine work out the details.

PREVIOUS | HOME | NEXT
11.6 Populating a PL/SQL Table with Object Names | BOOK INDEX | 11.8 Tracing PLVobj Activity

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

11.8 Tracing PLVobj Activity

You can get feedback on activity in the PLVobj package by requesting that the trace be displayed. PLVobj offers a standard on−off toggle with the following three programs:

```
PROCEDURE display;
PROCEDURE nodisplay;
FUNCTION displaying RETURN BOOLEAN;
```

The default setting for PLVobj is no display.

In the following SQL*Plus session, I turn on the trace for **PLVobj** and then execute a script to see all the lines in PLVio that contain the keyword SUBSTR. The **inline.sql** program first calls **PLVobj.setcurr** and later calls **PLVobj.bindobj**. The first three lines after the call to **inline.sql** show that I called **convobj**, then set the current values, and finally performed a bind. The reason that the "convert" trace appeared is that **setcurr** calls **convobj**.

```
SQL>  exec PLVobj.display
SQL>  @inline b:PLVio SUBSTR
convert: Schema.Name.Type = "PLV.."
set: Schema.Name.Type = "PLV.PLVIO.PACKAGE BODY"
bind: Schema.Name.Type = "PLV.PLVIO.PACKAGE BODY"
Lines with SUBSTR in PLV.PLVIO.PACKAGE BODY
330
332
512
```

12. PLVio: Reading and Writing PL/SQL Source Code

Contents:

The PLVio (PL/Vision Input/Output) package consolidates all the logic required to read from and write to repositories for PL/SQL source code. The PLVio package supports operating system files (as of Release 2.3), database tables, strings, and PL/SQL tables as both input and output sources. It hides all of the complexities of this I/O from users of the PLVio package. By relying on PLVio, other PL/Vision packages, such as PLVhlp and PLVcase, can pass on support for these varying repositories with minimal effort and maximum flexibility.

PLVio is a large package with many different elements. Generally, PLVio activity breaks down into the following areas:

- Set and manage the source repository, including the WHERE clause of the SQL statement used to define the source if it is a database table. Among other things, you can initialize and close the source.
- Set and manage the target repository. You can display, close, and clear the target.
- Read from the source and write to the target. You can even transfer the contents of the source repository to the target in a single step with the **`src2trg`** procedure.
- Save and restore repository settings. This is useful when you are using PLVio simultaneously in different contexts.
- Trace PLVio activity with a package window.

You can also use PLVio to manipulate text that is not PL/SQL source. Indeed, you can use PLVio to more easily write text out to operating system files (an interface to PLVfile and, underneath that, the builtin UTL_FILE package).

12.1 Why PLVio?

When you use PLVio, you can step away from the details of reading to and writing from specific types of text repositories. You build programs and utilities which get lines from the source and/or put lines to the target. But your code is not necessarily tied down to any specific type of repository. Consequently, your code can work with any of these different repositories without undergoing radical change.

The PLVhlp package's use of PLVio shows the power of abstracting the concept of source and repository for PL/SQL code. PLVhlp offers an architecture for displaying online help text for your own PL/SQL programs. The default mode for PLVhlp is to display the help text to standard output or your screen. Thus, I can call PLVhlp (through a frontend procedure in the PLV package) as shown below to get high−level information about PL/Vision itself:

```
SQL> exec PLV.help
Help for PLV
Overview of PL/Vision

PL/Vision is a development accelerator for PL/SQL programmers.
It is made up of a set of packages which you can use as
plug−and−play components in your own applications. Here is a
quick overview of some of the available packages:

PLVdyn − performs dynamic SQL and PL/SQL operations.
PLVexc − High−level exception handling capabilities.
PLVlog − Generic log mechanism.
PLVvu − View stored code and compile errors.
```

What if you are not developing applications in SQL*Plus and you cannot rely on DBMS_OUTPUT.PUT_LINE to display the help text? Instead of abandoning online help, you can instead redirect the output of the PLVhlp package to a PL/SQL table, for example, as shown below:

```
PLVio.settrg (PLV.pstab);
```

After this call, any attempts to display the help text will send that information to a PL/SQL table (**PLVio.target_table**, to be precise). You can then build your own program to read from this PL/SQL table and display the results in your development environment. You could even display this information (to reassure yourself, perhaps, that it really is there) with this command:

```
PLVio.disptrg;
```

Notice that I did not have to make any changes to the PLVhlp package to achieve this redirection; nor does any of the code you have built to extract the help text from your own program units need to be modified.

PL/Vision packages make extensive use of the PLVio package. To use it yourself, you will first set your source and target. Then you must use **get_line** to read from the source and **put_line** to write to the target. To make this package as flexible as possible, PLVio offers lots of additional programs, constants, and data structures. Don't be intimidated by the number and variety. Dip into the package as you need it, experiment with its nuances, and look at the examples of its usage in PL/Vision.

The following sections show how to use each of the different elements of the PLVio package.

PREVIOUS | HOME | NEXT
11.8 Tracing PLVobj Activity | BOOK INDEX | 12.2 Code Repositories Supported by PLVio

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

12.2 Code Repositories Supported by PLVio

The various programs in the PLVio package that manage the source and target repositories and that read and write lines of text are independent of the particular sources and targets. When you call **`put_line`**, for instance, you do not write code that says "write this line to a file." You simply "say" with your code: "Write this line to the target." You define the target independently of the actual read and write commands. This separation of logical and physical aspects of PL/SQL code I/O makes it easy to support a wide range of repositories −− and to add to that range as PL/SQL's capabilities expand.

When I first built PLVio, I was working with Release 2.1 of the PL/SQL language. I was able, therefore, to write PLVio to read to and write from database tables, but I could not read and write operating system files. That feature was not available until Release 2.3. I was still able to build the package and put it to use throughout PL/Vision. When Release 2.3 became available, I enhanced PLVio to support this new repository option and, with the simple act of a recompile *only* of the PLVio package body, my existing utilities could now manipulate PL/SQL source code in operating system files!

The PLVio package supports the following types of repositories:

- PL/SQL string
- Database table
- PL/SQL table
- Operating system file (server side)
- Standard output (the screen, usually)

You set the source by calling **`PLVio.setsrc`**: you set the target by calling **`PLVio.settrg`**. Both are described in later sections.

Before diving into all the different programs, here are some details about how these different repositories are handled in PLVio, both as source and target.

12.2.1 String Source or Target

When you specify a string as source, you pass that string of text in to PLVio when you call the **`PLVio.setsrc`** procedure. At that time, your string will be assigned to the **`text_in`** field of the **`string_repos`**. The **`string_repos`** record is defined as an instance of the following record TYPE:

```
TYPE string_repostype IS RECORD
     (text_in PLV.max_varchar2%TYPE,
      start_pos INTEGER := 1,
      text_len INTEGER := NULL,
      text_out PLV.max_varchar2%TYPE := NULL);
```

PLVio defines a line within this string source as all text up to the next newline character in the string (equivalent to CHR(10) and available as a named constant of **`PLVchr.newline_char`**). The maximum size of a line in PLVio is 2000 bytes, so you will need to break up a large program into multiple strings separated by a newline if you want to specify a string as a source. The maximize size of the entire text is 32,767 −− the maximum length of a PL/SQL variable length string (represented in the above record definition by the **`PLV.max_varchar%TYPE`** anchored declaration).

You can view the current contents of the source or target strings by calling the **`PLVio.srcstg`** or **`PLVio.trgstg`** functions, respectively.

12.2.2 Database Source or Target

You can use a database table as a source or target for PL/SQL code. In either case, you can use the default table (which is the USER_SOURCE data dictionary view for source and the **`PLV_source`** table for target) or you can specify your own table. Since PLVio uses PLVdyn to execute dynamic SQL, you can provide the name of the table and its columns. Regardless of their names, however, the columns of a database repository for PLVio must have at least four columns structured as shown in the record TYPE for a repository below:

```
TYPE repos_rectype IS
RECORD
   (name VARCHAR2(60),
    type VARCHAR2(10) := c_notset,
    name_col VARCHAR2(60) := 'name',
    type_col VARCHAR2(60) := 'type',
    line#_col VARCHAR2(60) := 'line',
    text_col VARCHAR2(60) := 'text',
    select_sql VARCHAR2(2000),
    insert_sql VARCHAR2(2000),
    where_clause VARCHAR2(1000) := NULL,
    starting_at VARCHAR2(1000) := NULL);
```

In other words, you will need a name string column, a type string column, a line number column, and a text string column. These columns can be named whatever you want and you can have other columns in addition to these four, but these columns must be available and specified to PLVio.

Given these requirements, the table shown in the left−hand column below is valid for use in PLVio, while the table in the right−hand column cannot be used, since it lacks a line number column:

Valid for PLVio Source	Not Usable for PLVio Source

`CREATE TABLE temp_source`
`   (progname VARCHAR2(100),`
`    progtype VARCHAR2(30),`
`    linenum INTEGER,`
`    linetext VARCHAR2(120));`
	`CREATE TABLE temp_source`
	`   (objname VARCHAR2(100),`
	`    objtype VARCHAR2(30),`
	`    objline VARCHAR2(120));`

As you can see, the record TYPE for a PLVio repository also stores other database−related information, such as the dynamically constructed SELECT and INSERT strings and the optional WHERE clause.

You need to have SELECT privileges only on the source database table. You will need INSERT and DELETE authority on the target database table. You may not, therefore, specify the USER_SOURCE data dictionary view as the target for output from PLVio.

When you specify a database table as the source repository, you will also make use of the PLVobj package to indicate the schema, program name, and program type you are interested in. Examples of this dependency are shown in Section 12.3, "Managing the Source Repository".

12.2.3 PL/SQL Table Target

If you want to avoid the SQL layer, you can use a PL/SQL table defined inside PLVio as the target for PL/SQL source code. PLVio does not currently support PL/SQL tables as sources for reading PL/SQL code. The PL/SQL table is defined in the PLVio specification as follows:

```
target_table PLVtab.vc2000_table;
target_row BINARY_INTEGER;
```

Since the **`target_table`** is in the specification, a user of PLVio can directly access and change the contents of **`target_table`**. It is up to you to only use this table in ways that are appropriate to PLVio and/or your specific coding objectives.

The **`target_row`** variable will tell you how many lines of code are defined in the PL/SQL table. The row number is treated as the line number for the source code. Once you have populated the table, you can display its contents or pass the table as an argument to another program to process the data in that table.

12.2.4 File Source or Target

You can request that **`PLVio.put_line`** write its text to an operating system file. In this case, **`PLVio.put_line`** calls the **`PLVfile.put_line`** program. This procedure in turn calls the appropriate elements of the builtin UTL_FILE package to interact with the operating system file. For more information on the requirements and restrictions when working with UTL_FILE, see Chapter 13, *PLVfile: Reading and Writing Operating System Files*.

12.2.5 Standard Output Target

You can request that output from calls to **`PLVio.put_line`** be directed to standard output or the screen. When you do this, **`PLVio.put_line`** is, in effect, calling the DBMS_OUTPUT.PUT_LINE program to display output (although it does happen through the **`p`** package). This is the way that **`PLVgen`** generates its PL/SQL source code, for example.

So if you ever execute a PLVgen program to generate code and you don't see anything, check your PLVio target type (with a call to the **`PLVio.trgtype`** function). You might be writing your code to a file or PL/SQL table or database table!

PREVIOUS	HOME	NEXT
12.1 Why PLVio?	BOOK INDEX	12.3 Managing the Source Repository

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

12.3 Managing the Source Repository

To read information from a PLVio repository, you must do the following:

1. Define the source using the **setsrc** program.
2. Fine−tune your access to the source repository. You can modify the WHERE clause of your SELECT from a database table source with a number of procedures.
3. Initialize the source using the **initsrc** program.

You are then ready to use the **get_line** procedure to retrieve lines of text from the source repository.

This section explains the **setsrc** and **initsrc** procedures. The next section explores how to modify the WHERE clause of the source SELECT for database tables.

12.3.1 Setting the Source

The **setsrc** procedure assigns the type of source repository and defines the structure of the source (if a database table). The header for **setsrc** is as follows:

```
PROCEDURE setsrc
   (srctype_in IN VARCHAR2,
    name_in IN VARCHAR2 := 'user_source',
    name_col_in IN VARCHAR2 := 'name',
    srctype_col_in IN VARCHAR2 := 'type',
    line#_col_in IN VARCHAR2 := 'line',
    text_col_in IN VARCHAR2 := 'text',
    schema_col_in IN VARCHAR2 := NULL);
```

The first argument, **srctype_in**, is the type of repository. The second argument, **name_in**, is the name of the repository. Its content varies according to the type of repository and will be explained below. The third through seventh arguments provide the names of the columns required for a database table source. The default values match the structure of the default table (the USER_SOURCE data dictionary view). Notice, therefore, that the schema column name is NULL. This column would be used only if you specified a table/view like ALL_SOURCE, which contains the source code for all programs to which you have access, regardless of schema.

The **setsrc** procedure transfers the arguments to the **srcrep** record, which is defined using the **repos_rectype** shown earlier. If you are using a string source, then the **string_repos** record is updated. If you are using a database source, then the SELECT statement that will be used to query from that table is constructed as follows (this is a simplified version of the actual SELECT, but gives you an idea of its structure):

```
srcrep.select_sql :=
   'SELECT ' || source_text_col_in || ', ' ||
                source_line#_col_in ||
   '  FROM ' || name_in || ' ' ||
   ' WHERE ' || source_name_col_in || ' = :name ' ||
   '   AND ' || srctype_col_in || ' = :type' ||
   ' ORDER BY ' || source_line#_col_in;
```

If you are using a string source, the **`name_in`** argument contains the string which holds the text for the program you want to read. All other arguments are ignored. This string is then assigned into the string repository record as shown below:

```
IF string_source
THEN
   string_repos.text_in := name_in;
   string_repos.start_pos := 1;
   string_repos.text_len := LENGTH (name_in);
   string_repos.text_out := NULL;
```

Notice that the other fields of the **`string_repos`** record are also initialized.

If you are using a file source, then the **`name_in`** argument is the name of the file. The **`source_name_col_in`** argument should contain the *type:name* specification for the object you are reading. So if you are reading the package body of PLVvu from the file **`PLVvu.spb`**, you would call **`setsrc`** as follows:

```
PLVio.setsrc (PLV.file, 'PLVvu.spb', 'b:PLVvu');
```

You must supply this third argument if you are writing (have set the target to) a database table.

If you are using a file source, all other arguments (after the first three) are ignored.

12.3.2 Initializing the Source

Once you have set the source repository, you can either move directly to initializing that repository, or you can, in the case of a database table source, modify the WHERE clause of the SELECT constructed by **`setsrc`**. In most cases, you will simply call **`initsrc`**, so I will discuss that procedure below. The next section discusses how to modify the source repository WHERE clause.

The header for the **`initsrc`** procedure is overloaded as follows:

```
PROCEDURE initsrc
   (starting_at_in IN INTEGER,
    ending_at_in IN INTEGER,
    where_in IN VARCHAR2 := NULL);

PROCEDURE initsrc
   (starting_at_in IN VARCHAR2 := NULL,
    ending_at_in IN VARCHAR2 := NULL,
    where_in IN VARCHAR2 := NULL);
```

The "integer version" of **`initsrc`** accepts up to three arguments, as follows:

`starting_at_in`
: The line number at which the query should start.

ending_at_in

The line number at which the query should end.

where_in
An optional WHERE clause to add to the existing clause of the source's SELECT statement.

If the source is a database table, specifying start and/or end line numbers results in additional elements in the WHERE clause of the SELECT statement. The **where_in** string is also appended to the SELECT's WHERE clause, if provided. For any other code sources, these three arguments are currently ignored. In other words, if you work with non−database table sources, you will always read the full set of lines of text in those sources. It is easy to see how **initsrc** should be enhanced to support these arguments; it's just a matter of time and resources. I encourage you to try adding this functionality yourself.

The "string version" of **initsrc** allows you to specify starting and ending strings for the source repository. In this case (and only when the source is a database table), the **get_line** procedure will only read those lines that come after the first occurrence of the starting string and before the first occurrence of the ending string.

12.3.3 Using setsrc and initsrc

Here are some examples of calls to **setsrc** and **initsrc**.

1. Set the source to a string and then initialize the source.

```
PLVio.setsrc (PLVio.c_string, long_code_string);
PLVIO.initsrc;
```

2. Set the current object in PLVobj to the body of the PLVvu package. Set the source to the ALL_SOURCE data dictionary view and restrict access to only the first five lines of the code for the PLVvu package body.

```
PLVobj.setcurr ('b:PLVvu');
PLVio.setsrc (PLV.dbtab, 'all_source', schema_col_in => 'owner');
PLVio.initsrc (1, 5);
```

3. Set the source to a table named **temp_source** with a set of completely different column names. Request that only those rows for the procedure **calc_totals** containing the string RAISE be read. Notice the use of named notation in my call to **initsrc**. Which of the two versions of **initsrc** is executed?

```
PLVobj.setcurr ('p:calc_totals');
PLVio.setsrc
   (PLV.dbtab, 'temp_source', 'progname', 'progtype', 'line#', 'line');
PLVio.initsrc (where_in => 'INSTR (line, ''RAISE'') > 0);
```

Answer: the string version of **initsrc** is executed, since there are default values for the string arguments. The integer version requires that the start and end numbers be provided.

12.3.4 High−Level Source Management Programs

Recognizing the most common sources of PL/SQL code, PLVio offers two specialized programs to both set and initialize the source, **usrc** and **asrc**. The **usrc** procedure sets the source repository to the USER_SOURCE data dictionary view. The **asrc** procedure sets the source repository to the ALL_SOURCE data dictionary view. Both **usrc** and **asrc** are overloaded with the same arguments as **initsrc**: the "starting at" string or line number, the "ending at" string or line number, and the optional WHERE clause. The headers for these programs are shown below:

```
PROCEDURE usrc
   (starting_at_in IN VARCHAR2 := NULL,
    ending_at_in IN VARCHAR2 := NULL,
    where_in IN VARCHAR2 := NULL);

PROCEDURE usrc
   (starting_at_in IN INTEGER,
    ending_at_in IN INTEGER,
    where_in IN VARCHAR2 := NULL);

PROCEDURE asrc
   (starting_at_in IN VARCHAR2 := NULL,
    ending_at_in IN VARCHAR2 := NULL,
    where_in IN VARCHAR2 := NULL);

PROCEDURE asrc
   (starting_at_in IN INTEGER,
    ending_at_in IN INTEGER,
    where_in IN VARCHAR2 := NULL);
```

With **asrc**, for example, I could replace these three lines of code:

```
PLVobj.setcurr ('b:PLVvu');
PLVio.setsrc (PLV.dbtab, 'all_source', schema_col_in => 'owner');
PLVio.initsrc (1, 5);
```

with the following two lines:

```
PLVobj.setcurr ('b:PLVvu');
PLVio.asrc (1, 5);
```

You shouldn't have to deal with all those details when it is the kind of source setting you will be performing again and again.

PREVIOUS | HOME | NEXT

12.2 Code Repositories Supported by PLVio | BOOK INDEX | 12.4 The Source WHERE Clause

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

12.4 The Source WHERE Clause

PLVio provides a set of programs used within PLVio and also available to you to modify the contents of the WHERE clause of the SELECT statement for a database table source. These programs must be called after the call to **`setsrc`** and before the call to **`initsrc`**.

The default WHERE clause for the database source is:

```
WHERE name = PLVobj.currname
  AND type = PLVobj.currtype
```

This WHERE clause reflects the relationship between the current object of PLVobj and the default PLVio source database table, **`user_source`**. It is stored directly in the **`srcrep.select_sql`** field and is set in the call to **`setsrc`**. Additional WHERE clause information is stored in the **`where_clause`** field of the same **`srcrep`** record (see Section 12.2.2, "Database Source or Target" earlier in this chapter).

You can modify this WHERE clause in two ways: replace it completely or add additional elements to that clause. The **`set_srcselect`** will do either of these actions. The **`set_line_limit`** applies additional elements to the WHERE clause. **`rem_srcselect`** and **`rem_line_limit`** remove elements from the WHERE clause. The **`srcselect`** function displays the current SELECT statement.

Each of these programs is explained below.

12.4.1 Viewing the Current Source SELECT

First, use the **`srcselect`** function to retrieve the current structure of the SELECT statement for the source repository. In the following example, I use **`p.l`** to display the current SELECT.

```
SQL> exec p.l(PLVio.srcselect);
SELECT text, line  FROM user_source WHERE instr (text, 'RAISE') > 0 AND
name = 'PLVEXC' ORDER BY line
```

This string is an example of a SELECT in which the WHERE clause was substituted completely by a call to **`set_srcwhere`**. The following session in SQL*Plus sets the source to the ALL_SOURCE view. The **`srcselect`** function returns the default (and more normal) kind of SELECT built and executed by PLVio.

```
SQL> exec PLVio.asrc
SQL> exec p.l(PLVio.srcselect);
SELECT text, line  FROM all_source WHERE name = :name AND type = :type
 AND owner = :owner ORDER BY line
```

12.4.2 Changing the WHERE Clause

To modify directly the WHERE clause of the SELECT statement, you will call the **`set_srcwhere`** procedure, whose header is:

```
PROCEDURE set_srcwhere (where_in IN VARCHAR2);
```

This procedure modifies the WHERE clause according to the following rules:

1.
   If the string starts with AND, then the string is simply concatenated to the current WHERE clause.

2.
   If the string starts with WHERE, then the entire current WHERE clause is replaced with the string provided by the user.

3.
   In all other cases, the core part of the WHERE clause (containing the bind variables for **`PLVobj.currname`** and **`PLVobj.currtype`**) is preserved, but any other additional elements are replaced by the specified string.

A few examples will demonstrate this procedure's impact. In each case, I initialize the SELECT statement with a call to **`PLVio.asrc`** so that the **`select_stg`** contains this information:

```
SELECT text, line
  FROM all_source
 WHERE name = :name
   AND type = :type
   AND owner = :owner
 ORDER BY line
```

Let's see what happens when I use **`set_srcselect`** to change the WHERE clause:

1.
   Add a clause to request that only lines 1 through 5 are read from ALL_SOURCE:

   ```
   PLVio.set_srcselect ('AND line BETWEEN 1 AND 5');
   ```

   The **`srcselect`** now looks like this:

   ```
   SELECT text, line
     FROM all_source
    WHERE name = :name
      AND type = :type
      AND owner = :owner
      AND line BETWEEN 1 AND 5
    ORDER BY line
   ```

2.
   Add the same clause as in Example 1 and then *replace* it with an element that limits rows retrieved to those that start with the keyword IF.

   ```
   PLVio.set_srcselect ('AND line BETWEEN 1 AND 5');
   PLVio.set_srcselect ('LTRIM (text) LIKE ''IF%''');
   ```

   The **`srcselect`** now looks like this:

   ```
   SELECT text, line

     FROM all_source
    WHERE name = :name
      AND type = :type
      AND owner = :owner
      AND LTRIM (text) LIKE 'IF%'
    ORDER BY line
   ```

3.

The following script displays all the lines currently stored in the USER_SOURCE data dictionary view that contain the keyword RAISE.

```
DECLARE
   line PLVio.line_type;
   numlines NUMBER;
BEGIN
   PLVio.setsrc (PLV.dbtab);
   PLVio.set_srcwhere
      ('WHERE instr (text, ''RAISE'') > 0');
   PLVio.initsrc;
   LOOP
      PLVio.get_line (line, numlines);
      exit when line.eof;
      p.l (line.text);
   END LOOP;
END;
/
```

Notice that the string I pass to **`set_srcwhere`** begins with the WHERE keyword. This signals to PLVio that the entire WHERE clause is to be discarded and replaced with the argument string so, in this case, **`srcselect`** would display this string:

```
SELECT text, line
  FROM all_source
 WHERE instr (text, 'RAISE') > 0
 ORDER BY line
```

12.4.3 Setting a Line Limit

The final program you can use to change the WHERE clause is the **`set_line_limit`** procedure. The header of **`set_line_limit`** is:

```
PROCEDURE set_line_limit
   (line_in IN INTEGER, loc_type_in IN VARCHAR2 := c_first);
```

The first argument, **`line_in`**, is the line number involved in the restriction. The **`loc_type_in`** argument dictates how the line number is used to narrow down the rows retrieved. There are four possible location types; the impact of each of these is explained in the table below.

Constant	Action
`c_first`	Retrieve lines >= specified line number
`c_last`	Retrieve lines <= specified line number
`c_before`	Retrieve lines > specified line number
`c_after`	Retrieve lines < specified line number

Here are some examples of the impact of **`set_line_limit`**:

1. Request that only lines greater than 100 be retrieved:

   ```
   PLVio.set_line_limit (100, PLVio.c_after);
   ```

   which adds the following element to the WHERE clause:

   ```
   /*LL100*/ AND line > 100 /*LL100*/
   ```

The comments which bracket the AND statement are included so that the entire element can be identified and removed as needed.

2. Request that only lines less than or equal to 27 be retrieved:

    ```
    PLVio.set_line_limit (27, PLVio.c_last);
    ```

    This call adds the following element to the WHERE clause:

    ```
    /*LL100*/ AND line <= 27 /*LL100*/
    ```

    The **set_line_limit** procedure is used by **initsrc** to process the "starting at" and "ending at" arguments. The string version of **initsrc** also makes use of the **line_with** function to convert a "starting at" string into the appropriate line number, which is then passed to the integer version of **initsrc**, which then calls **set_line_limit**. Review that code for more pointers about how to use both of these line−restricter programs.

12.4.4 Cleaning Up the WHERE Clause

You can also *remove* elements from the WHERE clause using the **rem_srcwhere** and **rem_line_limit** procedures. The **rem_srcwhere** program sets the **srcrep.where_clause** string to NULL, which means that the entire SELECT statement will be determined by the contents of the **srcrep.select_sql** field. The **rem_srcwhere** procedure takes no arguments so you would call it simply as follows:

```
PLVio.rem_srcwhere;
```

It is important to remember that **rem_srcwhere** only NULLs out the **srcrep.where_clause**. If you have previously called **set_srcwhere** with a string that started with WHERE, then the text of the **srcrep.select_sql** field itself is modified. This change is not corrected in any way by a call to **rem_srcwhere**. Instead, in this situation you will have to re−execute **setsrc** (and consequently, **initsrc**) to get back to the default SELECT statement.

The **rem_line_limit** will remove an element from the WHERE clause that was added by a call to **set_line_limit**. The header of this procedure is:

```
PROCEDURE rem_line_limit (line_in IN INTEGER);
```

You specify the same line number of the line limit passed to **set_line_limit**, and the appropriate chunk of text is extracted from the **srcrep.where_clause** string.

Suppose I called **set_line_limit** to ask that I only retrieve rows where the line number is greater than 10:

```
PLVio.set_line_limit (10, PLVio.c_after);
```

Then the following call to **rem_line_limit** will take out this restricting factor:

```
PLVio.rem_line_limit (10);
```

PREVIOUS | HOME | NEXT
12.3 Managing the Source Repository | BOOK INDEX | 12.5 Managing the Target Repository

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8*i* Features | Oracle Built−in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built−ins Pocket Reference

12.5 Managing the Target Repository

After reading about how to manage the source repository, you will probably be relieved to find that there is much less complexity involved in working with the target repository. Since the target is on the receiving end, there is no need to fiddle with the WHERE clause. You simply set and initialize the target, and then you are ready to write into that repository.

The **settrg** procedure defines the type of repository and its structure. The header for **settrg** is:

```
PROCEDURE settrg
   (trgtype_in IN VARCHAR2,
    name_in IN VARCHAR2 := 'PLV_source',
    target_name_col_in IN VARCHAR2 := 'name',
    trgtype_col_in IN VARCHAR2 := 'type',
    target_line#_col_in IN VARCHAR2 := 'line',
    target_text_col_in IN VARCHAR2 := 'text');
```

The first argument, **trgtype_in**, is the type of repository. The second argument, **name_in**, is the name of the repository. Its content varies according to the type of repository and is explained below. The third through sixth arguments provide the names of the columns required for a database table target. The default values match the structure of the default table (**PLV_source**).

The **settrg** procedure transfers the arguments to the **trgrep** record, which is defined using the **repos_rectype** shown earlier. If you are using a database target, then the INSERT statement that will be used to write lines to that table is constructed as follows:

```
trgrep.insert_sql :=
   'INSERT INTO ' || trgrep.name ||
      '( ' ||
      target_name_col_in || ', ' ||
      trgtype_col_in || ', ' ||
      target_line#_col_in || ', ' ||
      target_text_col_in || ') ' ||
   'VALUES (:name, :type, :line, :text)';
```

Notice that in the INSERT statement the name of the table and its columns are not hard−coded. You can establish those names in your call to **PLVio.settrg**. For example, if the target table is structured like this:

```
CREATE TABLE temp_target
   (progname VARCHAR2(100),
    progtype VARCHAR2(30),
    linenum INTEGER,
    linetext VARCHAR2(120));
```

then you would need to call **settrg** as follows:

```
PLVio.settrg
    (PLV.dbtab, 'temp_target',
     'progname', 'progtype',
     'linenum', 'linetext');
```

If you are using a string target, all arguments after the first are ignored. Lines of text are written to the **`string_repos.text_out`** field with newline characters inserted between lines.

If you are using a file target, then the **`name_in`** argument is the name of the file. All other arguments are ignored. Lines of text are written to the file in the order of execution.

> ***NOTE:*** Only the Write mode (replacing current contents of file) is supported. You cannot use PLVio to *append* to the end of a file.

12.5.1 Initializing the Target

Since there are really no intermediate actions between setting and initializing the target repository, the first line of **`settrg`** executes **`inittrg`**, so there is no need for you to do so explicitly.

If the target is a PL/SQL table, the PLVio **`target_table`** is emptied. If the target is a string, **`string_repos.text_out`** is set to NULL.

Currently, if the target is a database table or file, no initialization actions are taken. You will, therefore, need to perform the appropriate management on these data structures before using the **`PLVio.put_line`** procedure.

PREVIOUS | HOME | NEXT

12.4 The Source WHERE Clause | BOOK INDEX | 12.6 Reading From the Source

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8*i* Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

12.6 Reading From the Source

The **`PLVio.get_line`** procedure is the core program for reading from the source. This header for this program is:

```
PROCEDURE get_line
   (line_inout IN OUT line_type,
    curr_line#_in IN INTEGER := NULL);
```

The first argument, **`line_inout`**, is a record of type **`line_type`** (defined in the PLVio specification). The second argument, **`curr_line#`**, provides a current line number; if that number is not NULL, it will be used to increment the **`line#`** value found in the **`line_inout`** record.

The record contains all the information about a line necessary either for PLVio activity or other actions on a line of text. The definition of the record TYPE is:

```
TYPE line_type IS RECORD
   (text VARCHAR2(2000) := NULL,
    len INTEGER := NULL,
    pos INTEGER := 1,
    line INTEGER := 0, /* line # in original */
    line# INTEGER := 0, /* line # for new */
    is_blank BOOLEAN := FALSE,
    eof BOOLEAN := FALSE);
```

The following table explains the different fields of a **`line_type`** record:

`text`	The line of text.
`len`	The length of the line of text.
`pos`	The current position of a scan through this line.
`line`	The line number associated with this text in the source.
`line#`	The line number associated with this text in the target.
`is_blank`	TRUE if the text RTRIMS to NULL.
`eof`	TRUE if no line was placed into the record.

12.6.1 Main Steps of get_line

The **`get_line`** procedure has two main steps:

1. *Read a line from the source repository.* If reading from a database table, **`get_line`** uses the DBMS_SQL builtin package to fetch the next row and read the text and line number from the retrieved data. If reading from a file, **`get_line`** calls the **`PLVfile.get_line`** procedure. If reading from a string, **`get_line`** finds the next newline character and uses SUBSTR to extract the

desired characters.

2. *Massage the data retrieved from the repository so that the values of all record fields are set properly.* Assuming that data was found (the **eof** field is not set to TRUE), then the following actions are taken: replace newline characters with single spaces, replace tab characters with three spaces, increment the line number (using the second argument in the call to **get_line**, if provided), set the **pos** field to 1, set **is_blank** to TRUE if the string is composed solely of blanks, and compute the length of the line of text.

When these two steps are completed, the newly populated record is returned to the calling program.

12.6.2 Using get_line

To give you an idea of how you can put **get_line** to use, consider the SQL*Plus script shown below. Stored in file **inline2.sql**, this program displays all the lines of code in a given program that contain the specified string.

```
DECLARE
   line PLVio.line_type;
BEGIN
   PLVobj.setcurr ('&1');
   PLVio.asrc (where_in => 'INSTR (text, ''&2'') > 0');
   LOOP
      PLVio.get_line (line);
      EXIT WHEN line.eof;
      p.l (line.text);
   END LOOP;
   PLVio.closesrc;
END;
/
```

I call **PLVobj.setcurr** to set the current object to the requested program. I then point the source repository to ALL_SOURCE and add an element to the WHERE clause that will find only those lines in which the INSTR on the second argument returns a nonzero location. Now I am all set to loop through the rows identified by this WHERE clause. I exit when the **eof** field is set to TRUE; otherwise, I display the line and then call **get_line** again. Finally, I close the source when I am done, freeing up the memory used to read through ALL_SOURCE.

Here is an example of output from the **inline2** program:

```
SQL>  start inline2 b:PLVio SUBSTR
         (SUBSTR (srcrep.select_sql, 1, loc-1) ||
          SUBSTR (srcrep.select_sql, loc));
             SUBSTR (srcrep.where_clause, 1, loc-1) ||
             SUBSTR (srcrep.where_clause, loc2+cmnt_len-1);
             SUBSTR
          SUBSTR
      RETURN SUBSTR (line_in.text, pos_in);
```

You might compare the implementation of this functionality in **inline2.sql** with the approach taken in the **inline.sql** script. What are the differences between the two implementations? Which would you prefer to use and maintain?

PREVIOUS HOME NEXT
12.5 Managing the Target Repository BOOK INDEX 12.7 Writing to the Target

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8*i* Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

12.7 Writing to the Target

The **put_line** procedure of PLVio writes text to the target repository. Its header is overloaded as follows:

```
PROCEDURE put_line
   (string_in IN VARCHAR2, line#_in IN INTEGER := NULL);

PROCEDURE put_line (line_in IN line_type);
```

The first, "string" version of **put_line** simply bundles the text and line number into a record of type **line_type** and then calls the second, "record" version of **put_line** as shown below:

```
PROCEDURE put_line
   (string_in IN VARCHAR2, line#_in IN INTEGER := NULL)
IS
   v_line line_type;
BEGIN
   v_line.text := string_in;
   v_line.line# := line#_in;
   put_line (v_line);
END;
```

Why do I bother with these two versions? To make the package as easy as possible to use. In many situations, you will simply want to take a string and an optional line number and throw it out into the target. You aren't dealing with the more complex aspects of PLVio and therefore have no need for a line record. In this situation, calling the "record version" of **put_line** becomes a hassle. By writing a few extra lines of code into the package itself, I relieve my users of the burden of declaring a throw−away data structure −− the **line_type** record.

If you plan to build reusable code that will really and truly be reused, you will need to make this kind of extra effort.

The **put_line** procedure (which from this point on refers to the record version) hides all of the complexity about the current target. You simply tell PLVio that you want to put a line in the target; it worries about the specific mechanics required for the current type of repository, as discussed below.

12.7.1 Putting to Different Repositories

If writing to a file, **put_line** calls its comrade−in−code, PLV**file.put_line**. All details are pushed down to this building block package. This lower−level layer of code helps PLVio avoid being bogged down in writing to an operating system file.

If writing to a string, **put_line** concatenates the new text onto the existing **string_repos.text_out** value, making sure to append the specified line to the current value of **string_repos.text_out** with an intervening newline character:

```
ELSIF string_target
THEN
   IF string_repos.text_out IS NULL
   THEN
      string_repos.text_out := line_in.text;
   ELSE
      string_repos.text_out :=
         string_repos.text_out ||
         PLVchr.newline_char ||
         line_in.text;
   END IF;
```

The use of the newline character allows you to dump the contents of this string into a readable format either for display purposes or for spooling to a file.

When writing to a PL/SQL table, **put_line** assigns the value to the appropriate row in the table. If standard output is the target, **p.l** is used to display the text.

Finally, if the target is a database table, **put_line** makes use of dynamic SQL (using the builtin DBMS_SQL package and the PL/Vision PLVdyn and PLVobj packages) to insert a row in the table and then (at the interval specified by PLVcmt) possibly perform a commit as well. The program name and type that are written to the database table (see **settrg** for information on specifying the names of the columns holding this information) are taken from the PLVobj current object.

> ***NOTE:*** When you are writing to an operating system file, you must execute the **PLVio.closetrg** command to close the file before you can see any of the new information you have written to the file.

12.7.2 Batch Transfer of Source to Target

PLVio provides a procedure named **src2trg**; with a single line of code, this procedure copies the specified contents of the source repository to the target. The header for **src2trg** is:

```
PROCEDURE src2trg (close_in IN BOOLEAN := TRUE);
```

If you pass a value of TRUE in your call to **src2tg**, then **src2trg** will also close the target when it is done performing the transfer. You will almost certainly want to do this when you are writing to a file.

The **src2trg** procedure executes a loop to read through the source with **get_lines** and write to the target with **put_lines**, as the body of the procedure makes clear:

```
PROCEDURE src2trg
IS
   line line_type;
BEGIN
   LOOP
      get_line (line);
      EXIT WHEN line.eof;
      put_line (line);
   END LOOP;

   IF close_in
   THEN
      closetrg;
   ENDIF;
END;
```

I wrote this procedure for the **PLVhlp.show** procedure. This program needs to read all the help text from the source and transfer it to a PL/SQL table. From that point on, the **more** program displays a page's worth of text from the PL/SQL table using the **PLVio.disptrg** procedure (described in the next section).

12.7.3 Displaying the Target Repository

The **disptrg** procedure displays the contents of the target repository. Its header is:

```
PROCEDURE disptrg
   (header_in IN VARCHAR2 := NULL,
    start_in IN INTEGER := 1,
    end_in IN INTEGER := target_row);
```

The first argument, **header_in**, is an optional header to describe the output. The second and third arguments, also optional, restrict the lines to be displayed. These arguments are currently used only when the target type is PL/SQL table. For all other target types, all the code found in the target will be displayed.

Let's look at an example of using **disptrg**. In the following script (stored in file **dumpemp.sql**), I employ PLVio to write information to a table and then display it. This script simply transfers employee names from **emp** to the target PL/SQL table and then displays the contents of the target. Notice that I do not use PLVio for reading from any kind of source repository. I am only using the target side of PLVio. No one says you have to employ both source and target repositories of PLVio.

```
BEGIN
   PLVio.settrg (PLV.pstab);
   FOR emp_rec IN
       (SELECT ename FROM emp WHERE deptno = &1)
   LOOP
      PLVio.put_line (emp_rec.ename);
   END LOOP;
   PLVio.disptrg;
END;
/
```

Here is an example of the execution of **dumpemp.sql**:

```
SQL> start dumpemp 20
Contents of Table
JONES
FORD
SMITH
SCOTT
ADAMS
```

I could remove the first line of the script, which sets the target, and the rest of the script would work just fine. With this simple action, the script would work with whatever target has been selected outside of the script. The call to **put_line** would add a line to the target. And the final call to **PLVio.disptrg** would display the current target's contents. The utility and applicability of **disptrg** are kept distinct from the particular target.

PREVIOUS | HOME | NEXT
12.6 Reading From the Source | BOOK INDEX | 12.8 Saving and Restoring Settings

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

PREVIOUS

Chapter 12
PLVio: Reading and
Writing PL/SQL Source
Code

NEXT

12.8 Saving and Restoring Settings

The PLVio package is used by many PL/Vision packages and now you can use it as well. Code reusability is a wonderful thing, but it can get awfully dicey if one program's use of PLVio steps on another program's reliance on settings and data from PLVio. To help avoid such conflicts, PLVio can automatically save and restore its settings for the source and target repositories.

You get to decide when and if saves and restores should take place. To turn on saves/restores for the source repository, call the **`savesrc`** procedure (the default is to perform save/restores automatically). You can disable save/restore on source by calling the **`nosavesrc`** procedure. Finally, you can determine the current status of save/restore on source by displaying the value returned by the **`saving_src`** function. The headers for these programs are shown below:

```
PROCEDURE savesrc;
PROCEDURE nosavesrc;
FUNCTION saving_src RETURN BOOLEAN;
```

To turn on saves/restores for the target repository, call the **`savetrg`** procedure (the default is to perform save/restores automatically). You can disable save/restore on the target by calling the **`nosavetrg`** procedure. Finally, you can determine the current status of save/restore on target by displaying the value returned by the **`saving_trg`** function. The headers for these programs are shown below:

```
PROCEDURE savetrg;
PROCEDURE nosavetrg;
FUNCTION saving_trg RETURN BOOLEAN;
```

The save/restore facility does not maintain a stack of settings. If you save on top of a previous save, the former save settings are wiped out.

> ***NOTE:*** The values associated with the current source and target are saved, but the current position in a cursor or file, for example, cannot be maintained with a simple call to **`savesrc`** or **`savetrg`**. If you switch source or target in the middle of getting or putting data, you will not be able to rely on a restore to put you right back where you were.

PREVIOUS | HOME | NEXT
12.7 Writing to the Target | BOOK INDEX | 12.9 Cleaning Up Source and Target

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

12.9 Cleaning Up Source and Target

PLVio provides several programs so that you can clean up after yourself when using the package. These programs are described below.

12.9.1 Closing the Source

When you are done reading from the source repository, you should close it. The header for the **`closesrc`** procedure is:

```
PROCEDURE closesrc;
```

If the source is a database table, **`closesrc`** closes the dynamic SQL cursor. If the source is a file, the procedure closes the file. For a string or PL/SQL table source, no action is taken.

It is extremely important that you close your source; otherwise, a cursor or file will remain open for the duration of your session. This could lead to errors or unnecessary memory utilization.

The **`closesrc`** program will also automatically restore the PLVio settings if they were saved (i.e., if **`PLVio.saving_src`** returns TRUE).

12.9.2 Closing the Target

When you are done writing to the target repository, you should close it. The header for the **`closetrg`** procedure is:

```
PROCEDURE closetrg;
```

If the target is a database table, then **`closetrg`** calls **`PLVcmt.perform_commit`** to save your writes to the target (you can disable the commit with a call to **`PLVcmt.turn_off`**). If the source is a file, the procedure closes the file. For a string or PL/SQL table source, no action is taken.

When your target is a database table or a file, it is extremely important that you close your target. If you skip this step for a file, for example, that file might remain open for the duration of your session. You could also have outstanding transactions (the inserts to the target table) which are wiped out by a subsequent and perhaps unrelated rollback. This could lead to errors or unnecessary memory utilization.

The **`closetrg`** program will also automatically restore the PLVio settings if they were saved.

12.9.3 Clearing the Target

Before you write to a target repository, you may want to make sure that it is empty. The **`clrtrg`** procedure performs this action; its header is shown below:

```
PROCEDURE clrtrg
   (program_name_in IN VARCHAR2 := NULL,
    program_type_in IN VARCHAR2 := NULL);
```

The two arguments provide the name and type of program to be removed from the target source repository. These arguments are used only when the target is a database table. If the supplied values are NULL (the default), then the table identified in the call to **settrg** will be truncated using **PLVdyn**.

If you do provide a name and/or type, **clrtrg** uses those values to construct a WHERE clause so that only the specified program and type will be removed from the database table.

Remember that the default target database table is structured to hold the source code for one or more programs (it looks just like USER_SOURCE).

Suppose that I have called **settrg** as follows:

```
PLVio.settrg (PLV.pstab, 'new_source');
```

This means that I will be writing my text out to a table with this structure:

```
SQL> desc new_source
 Name       Null?    Type
 ---------- -------- --------------
 NAME       NOT NULL VARCHAR2(30)
 TYPE                VARCHAR2(12)
 LINE       NOT NULL NUMBER
 TEXT                VARCHAR2(2000)
```

This first call to **clrtrg**, then, will remove all records from the **new_source** table:

```
PLVio.clrtrg;
```

This next call to **clrtrg** will remove all package bodies stored in the table:

```
PLVio.clrtrg (program_type_in => 'PACKAGE BODY');
```

And this last call to **clrtrg** will remove the code for the **calc_totals** procedure:

```
PLVio.clrtrg ('calc_totals', 'procedure');
```

Currently, **clrtrg** only operates on database table targets.

Special Notes on PLVio

Here are some factors to consider when working with PLVio:

- The PLVio package comes in two flavors, depending on the version of the database you are using. The **PLVio.sps** and **PLVio.spb** files contain the PLVio package compatible with PL/SQL Release 2.2 and earlier. The **PLVio23.spb** file makes use of the UTL_FILE builtin package and can only be used with PL/SQL Release 2.3 (it is called in the **plvins23.sql** installation script).

- When the target is a database table, the **put_line** program will issue a commit by calling **PLVcmt.increment_and_commit**, as specified by the **PLVcmt. commit_after** procedure. If you do not want any commits to occur from within **PLVio**, call the **commit_after** program as follows:

```
PLVcmt.commit_after (0);
```

This command will turn off incremental commits, but it will still allow a **`PLVcmt.perform_commit`** to save changes. To turn off committing entirely, execute this command:

```
PLVcmt.turn_off
```

PREVIOUS | HOME | NEXT
12.8 Saving and Restoring Settings | BOOK INDEX | 13. PLVfile: Reading and Writing Operating System Files

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

13. PLVfile: Reading and Writing Operating System Files

Contents:

The PLVfile (PL/Vision FILE package) provides a layer of code around the builtin UTL_FILE package (which is available only with Release 2.3 of PL/SQL and beyond). UTL_FILE allows you to read from and write to operating system files on the same machine in which the database instance is running. The ability to read and write operating system files has been a long−standing request ("desperate plea" would, perhaps, be a better description) of PL/SQL developers.

The PLVfile package provides a number of high−level programs, such as **`fcopy`** to copy files, and **`infile`**, a file−oriented version of INSTR, to make it easier for PL/SQL developers to take advantage of this very useful builtin package.

This chapter show how to use each of the different elements of the PLVfile package.

13.1 A Review of UTL_FILE

Before you dive in to using either UTL_FILE or the PLVfile package, however, you should review the following information about UTL_FILE. *Chapter 15* of *Oracle PL/SQL Programming* offers more detail about these topics and the programs of the UTL_FILE package. The following sections offer some information about UTL_FILE that you need to know in order to use PLVfile properly.

13.1.1 Enabling File Access in the Oracle Server

To use the UTL_FILE package, you must add a line to the initialization file or *init.ora* for your database instance that indicates the directories in which you can read and write operating system files. This precaution is taken by Oracle so that you do not inadvertently corrupt important files like the database log files.

The entry in the *init.ora* file can have one of two formats:

```
utl_file_dir='*'
or
utl_file_dir='dir1,dir2...dirn'
```

where dir1 through dir*n* are individual, specific directory listings. If you use the first format, you are telling the Oracle database that developers can use UTL_FILE to write to any directory.

13.1.2 File Handles

Before you can do anything with a file, you have to open it (this process is explained below). At this point, UTL_FILE returns a handle or pointer to that file. You will then use this handle in all future manipulations of the file. A file handle has a special datatype of UTL_FILE.FILE_TYPE. FILE_TYPE is actually a PL/SQL record whose fields contain all the information about the file needed by UTL_FILE. (Currently, the record consists of a single column, named "id".)

You will reference the file handle, but not any of the individual fields of the handle. A handle is declared as follows:

```
DECLARE
   file_handle UTL_FILE.FILE_TYPE;
BEGIN
```

You could display the file handle which is generated by a call to UTL_FILE.FOPEN or the corresponding **`PLVfile.fopen`** functions as follows:

```
DECLARE
   file_handle UTL_FILE.FILE_TYPE;
BEGIN
   file_handle := PLVfile.fopen ('login.sql', PLVfile.c_read);
   p.l (file_handle.id);
END;
/
```

The **`p.l`** procedure is also overloaded in the PL/SQL 2.3 version so you can pass it the file handle directly and it will display the id field, as shown here:

```
p.l (file_handle);
```

Many PLVfile programs give you the option of providing either the file name or the file handle. In some cases, such as when you read from a file, you must use the file handle. In other situations, you can choose your method of specifying the file you want.

13.1.3 File Location, Name, and Mode

When you open a file with the UTL_FILE.FOPEN function, you must provide three arguments, as shown in the header below:

```
FUNCTION FOPEN
   (location_in IN VARCHAR2, file_name_in IN VARCHAR2,
    file_mode_in IN VARCHAR2)
RETURN UTL_FILE.FILE_TYPE;
```

The first argument is the location of the file (the directory); the second is the name of the file (name and extension); and the third is the file mode: "R" for read−only, "W" for write−only, and "A" for append.

While UTL_FILE needs all of this information, you should not necessarily have to provide it all every time you want to perform a file−related action. To make it easier for developers to work with files, PLVfile offers several options for opening and referencing files. You can provide separate locations and names in the UTL_FILE format. You can also provide a single string that contains both the location and name and let PLVfile parse that string into its separate components.

See Section 13.2, "Specifying the File in PLVfile" for more information on the approach taken by PL/Vision.

13.1.4 Handling UTL_FILE Errors

The UTL_FILE package provides a set of package−based exceptions and also makes use of two, more generic exceptions to inform you of problems it encounters. These exceptions are shown in Table 13.1.

It is great that the UTL_FILE package offers some predefined exceptions. By providing specific names for different exception conditions, I can trap for and handle those conditions. The downside of this approach is that I need to include explicit exception handlers by name, as shown below:

```
EXCEPTION
   WHEN UTL_FILE.INVALID_PATH
   THEN
      p.l ('Invalid path');
```

If I try to use a WHEN OTHERS clause instead (as you can see, there are many UTL_FILE−specific exceptions), the SQLCODE function simply and uniformly returns the number 1 −− indicating a user−defined exception. I cannot, in other words, determine which of the UTL_FILE exceptions occurred.

Table 13.1: Exceptions Related to the UTL_FILE Package

Exception Name	Description
NO_DATA_FOUND	The GET_LINE procedure tried to read past the end of the file. Remember that this same exception is also raised by implicit cursors and references to PL/SQL tables.
UTL_FILE.INTERNAL_ERROR	An internal error occurred. The requested operation was not completed.
UTL_FILE.INVALID_FILE_HANDLE	The specified file handle does not identify a valid, open file. This exception may be raised by calls to FCLOSE and FFLUSH.
UTL_FILE.INVALID_MODE	The mode supplied to FOPEN is not valid. Valid modes are: \`a', \`r', or \`w' (upper or lower case is acceptable).
UTL_FILE.INVALID_OPERATION	In FOPEN, this exception is raised when the file cannot be opened as requested. To open a file in read or append mode, the file must exist already. To open in write mode, the file must be writeable/ createable. In GET_LINE, FFLUSH, NEW_LINE, PUT, PUTF, and PUT_LINE, this exception is raised when you try to perform an operation which is incompatible with the mode under which the file was opened. For example, you tried to write to a read−only file.
UTL_FILE.INVALID_PATH	The path name supplied in a call to FOPEN is not valid. This error occurs when the location is not accessible or the path name is improperly constructed.
UTL_FILE.READ_ERROR	An operating system−specific error occurred when you tried to read from the file. For example, there might be a disk error.
UTL_FILE.WRITE_ERROR	An operating system−specific error occurred when you tried to write to the file. For example, the disk might be full.
VALUE_ERROR	The text read by GET_LINE is too long to fit in the specified buffer.

To help you deal with this situation, PLVfile offers the **exc_section** procedure, which predefines all these handlers (see Section 13.9, "Handling File Errors with PLVfile").

PREVIOUS | HOME | NEXT
12.9 Cleaning Up Source and Target | BOOK INDEX | 13.2 Specifying the File in PLVfile

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

13.2 Specifying the File in PLVfile

Now that you are aware of the way that UTL_FILE works, let's look at how PLVfile makes it easier to use the builtin package.

First of all, rather than insist that you separate out the file location from the file name to open and manipulate files, PLVfile provides a set of programs to make it easier to specify files. These programs are discussed below.

13.2.1 Setting the Operating System Delimiter

Each operating system has a delimiter that it uses to separate out directories and subdirectories, as well as separating directories from file names. Since PLVfile allows you to specify a file name as a single string (directory and file name combined), it needs to know about the operating system delimiter.

Use the **`set_delim`** to set the operating system delimiter. Its header is:

```
PROCEDURE set_delim (delim_in IN VARCHAR2);
```

You can find out the current operating system delimiter by calling the **`delim`** function:

```
FUNCTION delim RETURN VARCHAR2;
```

The PLVfile package offers two predefined delimiters for UNIX and DOS as shown:

```
c_unixdelim CONSTANT VARCHAR2(1) := '/';
c_dosdelim CONSTANT VARCHAR2(1) := '\';
```

The default, initial setting for the OS delimiter is the UNIX delimiter: "/".

13.2.2 Setting the Default Directory

PLVfile maintains a current directory so that you do not have to continually specify a directory if you are always working in the same area on disk. To set the current directory, call the **`set_dir`** procedure. To determine the current setting for the directory, call the **`dir`** function. The headers for these programs are:

```
PROCEDURE set_dir (dir_in IN VARCHAR2);
FUNCTION dir RETURN VARCHAR2;
```

The following call to **`set_dir`** sets the default directory to a path in DOS:

```
SQL> exec PLVfile.set_dir ('c:\orawin\oe_app');
```

> ***NOTE:*** If you do not call **`PLVfile.set_dir`** before passing in file names for reading and writing, there is a very good chance that your efforts to use PLVfile will be very frustrating. You will get errors that are difficult to understand, since you know your file exists. One way to minimize the frustration is to place a call to **`PLVfile.set_dir`** in your **`login.sql`** script.

Notice that I do not include a terminating backslash in the string. That "final" delimiter is needed when attaching the directory to the file name, but is neither needed nor legitimate for specifying a directory. In fact, if you include a final delimiter, PLVfile will strip it from the string, as shown below:

```
PROCEDURE set_dir (dir_in IN VARCHAR2)
IS
BEGIN
   v_dir := RTRIM (dir_in, v_delim);
END;
```

13.2.3 Parsing the File Name

PLVfile allows you to provide the file name as a single string. When you do this, PLVfile calls **`parse_name`** to parse the string into its separate components. The header for **`parse_name`** is:

```
PROCEDURE parse_name
   (file_in IN VARCHAR2, loc_out IN OUT VARCHAR2,
    name_out IN OUT VARCHAR2);
```

where **`file_in`** is the full file specification (location, name, and extension). The **`loc_out`** argument receives just the directory, while the **`name_out`** argument receives the name and extension. It relies on the operating system delimiter you assigned with a call to **`set_dir`** in order to find the start of the file name.

If the string you pass to **`parse_name`** does not have a directory prefixed on the file name, PLVfile will return the default directory as the location.

The following table shows how **`parse_name`** parses and returns values:

parse_name	Default Directory	File Location Returned	File Name Returned
/usr/app/names.lis	NULL	*/usr/app*	*names.lis*
/usr/app/names.lis	*/oracle/prod/defdir*	*/usr/app*	*names.lis*
names.lis	NULL	NULL	*names.lis*
names.lis	*/oracle/prod/defdir*	*/oracle/prod/defdir*	*names.lis*

This procedure is used extensively inside PLVfile (see Section 13.4, "Opening and Closing Files" for an example of how **`parse_name`** is used to overload several different versions of **`fopen`**). You can, however, also call **`parse_name`** directly in your own application. Just make sure that you have set the OS delimiter before you use **`parse_name`**.

PREVIOUS | HOME | NEXT
13.1 A Review of UTL_FILE | BOOK INDEX | 13.3 Creating and Checking Existence of Files

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

13.3 Creating and Checking Existence of Files

PLVfile provides a program (four overloaded versions, actually) to create a file. The headers for **fcreate** are the following:

```
FUNCTION fcreate
   (loc_in IN VARCHAR2, file_in IN VARCHAR2, line_in IN VARCHAR2)
RETURN UTL_FILE.FILE_TYPE;

FUNCTION fcreate (file_in IN VARCHAR2, line_in IN VARCHAR2 := NULL)
RETURN UTL_FILE.FILE_TYPE;

PROCEDURE fcreate
   (loc_in IN VARCHAR2, file_in IN VARCHAR2, line_in IN VARCHAR2);

PROCEDURE fcreate (file_in IN VARCHAR2, line_in IN VARCHAR2 := NULL);
```

In versions of **fcreate** with three arguments, you provide the location, name, and single line to be deposited in the file. Notice that all three values are required. In versions of **fcreate** with two arguments, you provide the file specification (location and name combined, or just the name, in which case the default directory will be applied).

Notice that the overloading is not only among different parameter lists, but even different program types. I will explain this approach to overloading in PLVfile in this section; you will see it repeatedly throughout the package.

The overloading of **fcreate** achieves two objectives:

1. It allows the developer to either obtain the handle to the newly created file (the function versions) or ignore that file handle entirely (the procedure versions). You'll want to retrieve the handle if you plan to perform other actions on that file. If you only want to create the file and then move on to other business, it will be easier and more intuitive to use the procedure versions.

2. It allows the developer to provide the location and name separately (UTL_FILE style) or specify a single, combined string. The three−argument version requires all entries. The two−argument version allows you provide just a name; if you do not specify a line, it places the following default substitution line in the file:

```
v_subst_line VARCHAR2(200) :=
   'I make my disk light blink, therefore I am.';
```

When you call **fcreate**, it "initializes" a file to the line you provide (or the default line value) and then it closes the file if you have called the procedure version of **fcreate**. On the other hand, if you have called the **fcreate** function, PLVfile returns the handle to the file and then keeps the file open.

13.3.1 Checking a File's Existence

Perhaps you only want to create a file if it already exists. PLVfile offers the **`fexists`** function to provide you with this information. The headers for this overloaded function are:

```
      FUNCTION fexists (loc_in IN VARCHAR2, file_in IN VARCHAR2)
      RETURN BOOLEAN;

      FUNCTION fexists (file_in IN VARCHAR2)
      RETURN BOOLEAN;
```

You can provide separate locations and file names, or simply pass in the single string with combined information. The function returns TRUE if the file can be opened for read−only access successfully. If the file is already open, this function will return FALSE −− so use **`fexists`** with care.

PREVIOUS | HOME | NEXT
13.2 Specifying the File in PLVfile | BOOK INDEX | 13.4 Opening and Closing Files

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

13.4 Opening and Closing Files

PLVfile offers its own **fopen** and **fclose** programs to open and close operating system files (UTL_FILE has programs of the same names). The **fopen** module is overloaded as shown below:

```
PROCEDURE fopen
   (loc_in IN VARCHAR2, file_in IN VARCHAR2, mode_in IN VARCHAR2);

PROCEDURE fopen
   (file_in IN VARCHAR2, mode_in IN VARCHAR2 := c_all);

FUNCTION fopen
   (file_in IN VARCHAR2, mode_in IN VARCHAR2 := c_all)
RETURN UTL_FILE.FILE_TYPE;
```

Use the function version of **fopen** when you want to retrieve and use the file handle. Otherwise, you can simply call either of the procedure versions to open the file and not worry about declaring a file handle.

The **fclose** and **fclose_all** procedures may be used to close one or all files. Their headers are:

```
PROCEDURE fclose (file_in IN UTL_FILE.FILE_TYPE);

PROCEDURE fclose_all;
```

Notice that there is no overloading of the **fclose** program. Until a reader or the author enhances PLVfile to maintain a PL/SQL table of opened file names and their handles, there is no way to close a file by name. The **fclose_all** procedure is a passthrough to UTL_FILE.FCLOSE_ALL, which shuts down any files that have been opened with the UTL_FILE.FOPEN program.

You may find it useful to include a call to **fclose** or, more likely, **fclose_all**, in the exception sections of programs that work with PLVfile. That way if your program fails during file manipulation, it will not leave a file open. If the file is left open, that could cause another, different error, the next time you try to run it.

13.5 Reading From a File

PLVfile offers several different ways to read information from an operating system file. The **get_line** procedure gets the next line from the file. The **line** function returns the *n*th line from a file. The overloaded **infile** functions returns the line in which a string is found. These programs are explored below.

13.5.1 Reading the Next Line

Use the **get_line** procedure to read the next line from a file. The header for **get_line** is:

```
PROCEDURE get_line
   (file_in IN UTL_FILE.FILE_TYPE, line_out OUT VARCHAR2,
    eof_out OUT BOOLEAN);
```

You must provide a file handle (**file_in**); you cannot get the next line from a file by name. This means that you must already have opened the file using one of the **fopen** functions. The second argument of **get_line** (**line_out**) receives the string which is found on the next line. The **eof_out** argument is a flag which is set to TRUE if you have read past the end of the file.

When **eof_out** returns TRUE, **line_out** is set to NULL. You should not, however, test the value of **line_out** to determine if you are at the end of the file. The **line_out** argument could be set to NULL if the next line in a file is blank.

The following script (stored in the file **dispfile.sql**) uses **get_line** to read all the lines from a file and then display those lines.

```
DECLARE
   fileid UTL_FILE.FILETYPE;
   line PLVfile.max_line%TYPE;
   eof BOOLEAN;
BEGIN
   fileid := PLVfile.fopen ('&1');
   LOOP
      PLVfile.get_line (fileid, line, eof);
      EXIT WHEN eof;
      p.l (line);
   END LOOP;
   PLVfile.fclose (fileid);
END;
/
```

I use the **max_line** variable of PLVfile to declare the line datatype. This gives me a way to avoid having to hard−code the length of a line. Then I open the file (provided through a SQL*Plus substitution parameter) in the simplest possible way: location and name combined, assuming read−only access. My simple loop reads the next line and exits when the end−of−file condition is reached. If I did retrieve a line, I display it. When done, I close the file.

13.5.2 Reading the nth Line

Use the **`line`** function to retrieve the specified line from a file. The header for **`line`** is:

```
FUNCTION line (file_in IN VARCHAR2, line_num_in IN INTEGER)
RETURN VARCHAR2;
```

Notice that in this function you supply a file name and not a file handle (in fact, you don't even have the option of providing the location and name separately). The second argument is the line number you want retrieved.

The **`line`** function opens (in read−only mode), scans, and closes your file. You do not have to −− and should not −− perform any of these steps. If the line number specified is 0 or is greater than the number of lines in the file, the function will return a NULL value.

This function is handy when the lines in your file have a predefined or predictable structure. For example, you might have an *.ini* or initialization file for a program in which the first line is the name of the program, the second line the date and time of last use, and the third line the user who last accessed account information. You could then use **`PLVfile.line`** to retrieve precisely the information you needed. The following call to the **`line`** function extracts just the date and time of last use. It assumes that you have also made use of the standard PL/Vision date mask when writing this information to the file.

```
v_lastuse := TO_DATE (PLVfile.line ('oe.ini', 2), PLV.datemask);
```

13.5.3 The INSTR of PLVFile

PLVfile provides a function which operates within a file in much the same way that the builtin INSTR function operates on a string. INSTR returns the position in which the *n* th occurrence of a substring is found. **`PLVfile.infile`** returns the line number in which the *n*th occurrence of a string occurs. The header of the **`infile`** function, again overloaded to allow specification of the file in two different ways, is shown below:

```
FUNCTION infile
   (loc_in IN VARCHAR2,
    file_in IN VARCHAR2,
    text_in IN VARCHAR2,
    nth_in IN INTEGER := 1,
    start_line_in IN INTEGER := 1,
    end_line_in IN INTEGER := NULL,
    ignore_case_in IN BOOLEAN := TRUE)
RETURN INTEGER;

FUNCTION infile
   (file_in IN VARCHAR2,
    text_in IN VARCHAR2,
    nth_in IN INTEGER := 1,
    start_line_in IN INTEGER := 1,
    end_line_in IN INTEGER := NULL,
    ignore_case_in IN BOOLEAN := TRUE)
RETURN INTEGER;
```

The arguments to the **`infile`** function are described below:

Parameter Name	Description

`file_in`
`loc_in, file_in`
	The name of the file to be opened. The function is overloaded to allow both the location name and combined name specification for the file. All other arguments are common among the two.

`text_in`	The chunk of text to be searched for in each line of the file.
`nth_in`	The number of times the text should be found in distinct lines in the file before the function returns the line number. Default is 1, which means the first match. This value must be at least 1.
`start_line_in`	The first line in the file from which the function should start its search. This value must be greater than 0.
`end_line_in`	The last line in the file to which the function should continue its search. If NULL (the default), then search through end of the file. This value must be greater than or equal to **`start_line_in`**.
`ignore_case_in`	Indicates whether the case of the file contents and **`text_in`** should be ignored when checking for its presence in the line.

The **`infile`** function opens (in read−only mode), scans, and closes your file. You do not have to −− and should not −− perform any of these steps.

The only required parameters are **`file_in`** and **`text_in`**. I can, as a result, call **`infile`** with this minimum number of arguments:

```
first_find := PLVfile.infile ('names.vp', 'Hanubi';
```

I can, however, also do so much more, as shown in the examples below.

1. Confirm that the role assigned to this user is SUPERVISOR.

   ```
   IF PLVfile.line ('config.usr', 'ROLE=SUPERVISOR') > 0
   THEN
      update_schedule;
   END IF;
   ```

2. Find the second occurrence of `DELETE' starting with the fifth line.

   ```
   v_line := PLVfile.line ('commands.dat', 'delete', 2, 5);
   ```

3. Verify that the third line contains a terminal type specification. I ask for an exact match on the case of the text in the file, since the setup file has a specific structure.

   ```
   v_line := PLVfile.line
      ('setup.cfg', 'termtype=', 1, 3, ignore_case_in => FALSE);
   ```

The **`infile`** function differs from INSTR in at least one way: it does not support negative values for the starting line number of the search. INSTR does recognize this kind of argument, causing it to scan backwards through the string. You cannot scan backwards through the contents of a file.

13.5.3.1 Building utilities around infile

Suppose I receive profit−and−loss statements electronically from each of my regional offices every month. The number of items in the statement can change, but the file must contain a monthly total line in the format:

```
month_total=NNNNNN
```

where *NNNNNN* is the dollar amount.

Before the availability of UTL_FILE, you would have had to use SQL*Loader to load the file into a "temporary" table and then query the contents of that table. With UTL_FILE and (more to the point of this chapter, PLVfile's functions) you can skip the temporary table and extract the information directly from the file.

The **`mth_total`** function shown below (stored in file *use\mthtotal.sf*) makes use of both **`PLVfile.line`** and **`PLVfile.infile`** to extract the monthly total for the specified region and month.

```
FUNCTION mth_total
   (region_in IN INTEGER,
    month_in IN VARCHAR2,
    key_in IN VARCHAR2 := 'month_total=') RETURN NUMBER
IS
   v_file VARCHAR2(100) :=
      'pnl' || TO_CHAR (region_in) || '.' || month_in;
   v_linenum INTEGER;
   v_line PLVfile.max_line%TYPE;

   retval NUMBER := NULL;
BEGIN
   v_linenum := PLVfile.line (v_file, key_in);
   If v_linenum IS NOT NULL
   THEN
      v_line := PLVfile.line (v_file, v_linenum);
      retval := SUBSTR (v_line, LENGTH (key_in) + 1);
   END IF;
   RETURN retval
EXCEPTION
   WHEN OTHERS THEN RETURN NULL;
END;
```

This function, first of all, assumes that the default directory for the profit−and−loss files has already been set. It then constructs the file name from the region number and month string (the month string is an extension of the form *MMYY*). The call to **`PLVfile.line`** locates the line containing the keyword (which is also passed in as an argument to increase the flexibility of the function). If the line number if not NULL, a match was found. So **`PLVfile.line`** is then called to return the text of that line. Finally, I use SUBSTR to extract only the numeric part of the line.

PREVIOUS | HOME | NEXT
13.4 Opening and Closing Files | BOOK INDEX | 13.6 Writing to a File

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

13.6 Writing to a File

When you open a file in write−only mode (**PLVfile.c_write**) or append mode (**PLVfile.c_append**), you usually then want to write or put a line into that file. To do this, you will call the **PLVfile.put_line** procedure, whose header is shown below:

```
PROCEDURE put_line
   (file_in IN UTL_FILE.FILE_TYPE, line_in IN VARCHAR2);
```

Notice that you must provide a file handle to put a line in a file. You must, therefore, have already opened the file using one of the function versions of **fopen**.

13.6.1 Appending a Line

The **append_line** procedure offers a quick way to add a line on to the end of a file. Its header is:

```
PROCEDURE append_line (file_in IN VARCHAR2, line_in IN VARCHAR2);
```

You provide the file and the line you want appended on the end of the file. PLVfile automatically opens the file in append mode, writes the line using **put_line**, and then closes the file.

13.7 Copying File Contents

PLVfile offers several different programs to copy the contents of a file. You can copy from one file to another, from a file to a PL/SQL table, and from a PL/SQL table to a file. These programs are explained below.

13.7.1 Copying File to File

You can copy the entire contents of one file to another file with a single program call via the **fcopy** procedure; its header is shown here:

```
PROCEDURE fcopy
   (ofile_in IN VARCHAR2,
    nfile_in IN VARCHAR2,
    start_in IN INTEGER := 1,
    end_in IN INTEGER := NULL);
```

The **ofile_in** string is the name of the "original" file. The **nfile_in** string is the name of the "new" file. You can also specify which lines from the original file you want to copy. The **start_in** value is the number of the first line to copy. The **end_in** argument is the last line of the file to copy. If the **end_in** argument is NULL, then all lines from the **start_in** line to the end of the file are copied.

13.7.2 Copying File to PL/SQL Table

You can copy the contents of a file directly into a PL/SQL table with the **file2pstab** program. The header for the procedure is:

```
PROCEDURE file2pstab
   (file_in IN VARCHAR2,
    table_inout IN OUT PLVtab.vc2000_table,
    rows_out OUT INTEGER);
```

You specify the name of the file and provide the procedure with a PL/SQL table declared using the PLVtab package. Each line of the file is then copied into consecutive rows in the PL/SQL table. The **rows_out** argument contains the total number of rows set in the table. If the file is empty, the **rows_out** argument will be set to 0.

Once you have moved the file contents to a PL/SQL table, you can access the information in any PL/SQL program.

13.7.3 Copying File to List

You can copy the contents of a file directly into a PL/Vision list, implemented using the PLVlst package. The header for the procedure is:

```
PROCEDURE file2list (file_in IN VARCHAR2, list_in IN VARCHAR);
```

where **file_in** is the file name and **list_in** is the name of the PL/Vision list. This list is implemented in a PL/SQL table.

If the list does not exist, it is initialized by **file2list**. If the list already exists, the lines from the file are appended to the end of the list.

The combination of **PLVfile.file2list** and the full set of list management programs in PLVlst offers you a powerful means of storing lists of information in operating system files and then integrating that information into PL/SQL−based applications.

13.7.4 Copying PL/SQL Table to File

You can perform a "bulk" write to a file with the **pstab2file** procedure. This program transfers the contents of a VARCHAR2 PL/SQL table into the specified file. The header for this procedure is:

```
PROCEDURE pstab2file
   (table_in IN PLVtab.vc2000_table,
    rows_in IN INTEGER,
    file_in IN VARCHAR2,
    mode_in IN VARCHAR2 := c_write);
```

where the first argument is **table_in**, the PL/SQL table (defined using the PLVtab package). The **rows_in** parameter provides the number of rows in the table (**pstab2file** assumes that the PL/SQL table is populated sequentially from row 1). The string **file_in** provides the file name. Finally, you can also provide the file operation mode in the **mode_in** argument. The default value for **mode_in** is **PLVfile.c_write**, which means that any existing roles in the specified file will be replaced by the contents of the PL/SQL table.

You can also request that the PL/SQL table be transferred in "append mode." In this case, all PL/SQL table data will be appended to the end of the file. This approach is shown below:

```
PLVfile.pstab2file
   (my_table, rows_in_tab, 'temp.sql', PLVfile.c_append);
```

If you try to execute **pstab2file** with the read−only mode, **PLVfile.c_read**, you will receive an error.

If **pstab2file** encounters any undefined rows between 1 and **rows_in** − 1, it will trap the NO_DATA_FOUND exception and continue past that error. It will, as a result, transfer as many rows as possible from the PL/SQL table.

The following script (stored in the file **dumpprog.sql**) shows how to use the **pstab2file** to generate a source code file for the specified program.

```
BEGIN
   /* Set the current object from user. */
   PLVobj.setcurr ('&1');

   /* Read from ALL_SOURCE, write to PL/SQL table. */
   PLVio.asrc;
   PLVio.settrg (PLV.pstab);

   /* Copy the program source to the PL/SQL table. */
   PLVio.src2trg;

   /* Write the contents of the PL/SQL table to a file. */
   PLVfile.pstab2file
      (PLVio.target_table, PLVio.target_row, '&2');
```

```
END;
/
```

Here is the command you would execute in SQL*Plus to copy the stored source code of the body of the PLVvu package to a file named **`PLVvu.spp`**:

```
SQL> @dumpprog b:PLVvu PLVvu.spp
```

PREVIOUS | HOME | NEXT
13.6 Writing to a File | BOOK INDEX | 13.8 Displaying File Contents

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

13.8 Displaying File Contents

Use the **`display`** procedures (overloaded for file name and file handle) to display the contents of an operating system file from within a PL/SQL program. The headers for these procedures are:

```
PROCEDURE display
   (file_in IN UTL_FILE.FILE_TYPE,
    header_in IN VARCHAR2 := NULL,
    start_in IN INTEGER := 1,
    end_in IN INTEGER := NULL);

PROCEDURE display
   (file_in IN VARCHAR2,
    header_in IN VARCHAR2 := NULL,
    start_in IN INTEGER := 1,
    end_in IN INTEGER := NULL);
```

where **`file_in`** is the file name or the handle, **`header_in`** is an optional header for the display, and the **`start_in`** and **`end_in`** arguments specify the rows which you wish to display. If the **`end_in`** argument is NULL (the default value), then all lines from the starting line number to the end of the file will be displayed.

13.9 Handling File Errors with PLVfile

Many different errors can occur when you are attempting to read and write files. A file may not have been opened when you try to read it, or you may have opened it in read−only status and tried to write to it, and so on. When the PL/SQL runtime engine detects a problem, it raises one of the system exceptions (NO_DATA_FOUND and VALUE_ERROR) or one of the package−specific exceptions (such as UTL_FILE.READ_ERROR). The package−specific exceptions are user−defined, which means that if you check the SQLCODE return value in a WHEN OTHERS exception section, you see the value 1. This information is not very useful for debugging purposes.

The current version of PLVfile does attempt to trap the appropriate UTL_FILE exceptions in some of its programs (see the implementations of **fcreate**, **fopen**, and **get_line**, among others). It also provides a sample program which has an exception section handling each of the package−specific exceptions.

This procedure, **exc_section**, is not intended to be executed as it is written. Instead, you should cut the exception section from this procedure and paste it into your own program. You will then be able to trap, identify, and respond to the specific errors raised when using the UTL_FILE package.

To show how to use this block of code, suppose that the exception section consists only of these lines:

```
EXCEPTION
   WHEN UTL_FILE.INVALID_PATH
   THEN
      PLVexc.recNstop ('Invalid path');
   WHEN OTHERS
   THEN
      PLVexc.recNstop (SQLCODE);
END;
```

In other words, if the INVALID_PATH exception is raised, the PLVexc program records the problem and then raises an exception to stop the process. For any other error, the SQLCODE is saved with the error information and program execution halts.

I then build the following procedure to read the first line from a file and display that line.

```
PROCEDURE checkitout (file_in IN VARCHAR2)
IS
   aline PLVfile.max_line%TYPE;
BEGIN
   aline := PLVfile.line (file_in, 1);
   p.l (aline);
END;
```

When I run the program, I keep getting unhandled exceptions. Since I do not check for any specific exceptions, the PL/SQL runtime engine simply informs me that a user−defined exception occurred. It is very difficult for me to figure out what is going on. So I paste in the exception section provided by PLVfile and then I have:

```
PROCEDURE checkitout (file_in IN VARCHAR2)
IS
   aline PLVfile.max_line%TYPE;
BEGIN
   aline := PLVfile.line (file_in, 1);
   p.l (aline);
EXCEPTION
   WHEN UTL_FILE.INVALID_PATH
   THEN
      PLVexc.recNstop ('Invalid path');
   WHEN OTHERS
   THEN
      PLVexc.recNstop (SQLCODE);
END;
```

Now when the program executes, the message "Invalid path" appears on my screen. I immediately realize that I have not called the **`PLVfile.set_dir`** to set my default directory and so my file open step is failing.

In the real world you will find yourself adding an exception section with eight different handlers so you can distinguish between all the different errors. This adds many lines to your programs, but it is worth it when you have to debug your file I/O activity. In addition, since it is predefined for you in PLVfile, you do not have to personally do a whole lot of work to get the benefit.

This standard exception section is defined in the package body within the **`exc_section`** procedure, but it is also included as a help text stub under the topic EXC_SECTION in the package specification. So you can at any time execute the following line to display the exception section:

```
SQL> exec PLVfile.help ('exceptions');
```

PREVIOUS HOME NEXT

13.8 Displaying File Contents BOOK INDEX 13.10 Tracing PLVfile Activity

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

13.10 Tracing PLVfile Activity

PLVfile offers a standard PL/Vision toggle to allow you to dynamically turn on or off a trace of PLVfile activity. The triumvirate of programs for this toggle is:

```
PROCEDURE show;
PROCEDURE noshow;
FUNCTION showing;
```

When you call **`PLVfile.show`**, the trace is turned on (the default is that the trace is not active). Currently, PLVfile offers a trace of the **`get_line`** and **`put_line`** actions only.

It can be very frustrating to work with UTL_FILE and, by extension, PLVfile. You might think that you have taken all the necessary steps to perform file I/O, but you keep getting "Invalid Operation" errors. The PLVfile trace may well be able to give you the information you need to identify your problem more easily.

Chapter 14

14. PLVtmr: Analyzing Program Performance

Contents:

In *Oracle PL/SQL Programming*, I explored the implementation of the **`sp_timer`** package, which provides an easy−to−use interface to the DBMS_UTILITY.GET_TIME builtin function. GET_TIME gives us a mechanism for calculating the elapsed time of PL/SQL code execution down to the hundredth of a second. Since the publication of that book, I have enhanced **`sp_timer`** and it has evolved into the PLVtmr (PL/Vision TiMeR) package. The following sections show how to use each of the different elements of PLVtmr. For more information about how this package was developed, see *Oracle PL/SQL Programming*.

Now you have the following options when analyzing PL/SQL performance:

- *Turn the timer on or off.* You can keep your timers embedded in your application. They will not do anything if you explicitly turn off PLVtmr.
- *Retrieve and display the elapsed time.* You have many options for the format of the elapsed time data. These options allow you to use PLVtmr in SQL*Plus, as well as Oracle Developer/2000 and any other environment which supports PL/SQL program execution.
- *Execute prebuilt performance comparison procedures.* These programs allow you to get a sense of the difference in performance of implicit and explicit cursors and of the overhead of a function call.

14.1 Toggling the Timer

PLVtmr supplies a toggle so that you can leave your timing program calls in your code, but only have the timings execute when desired.

To turn on the activity of PLVtmr, call the **`turn_on`** program; its header is:

```
PROCEDURE turn_on;
```

To turn off the activity of PLVtmr, call the **`turn_on`** program; its header is:

```
PROCEDURE turn_off;
```

When turned off, PLVtmr will not add any overhead to your code execution and will not display any timing information.

PREVIOUS | HOME | NEXT
IV. Developer Utility Packages | BOOK INDEX | 14.2 Capturing the Start Time

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

14.2 Capturing the Start Time

The **capture** procedure of PLVtmr allows you to capture the start time for a timing session. Its header is:

```
PROCEDURE capture (context_in IN VARCHAR2 := NULL);
```

The single argument supplies a context which is associated with this timing session. When you call **capture**, PLVtmr in turn calls DBMS_UTILITY.GET_TIME to "capture the moment."

The following call to **capture** starts the timing session and assigns it the name "Calculating Totals":

```
PLVtmr.capture ('Calculating Totals');
```

PREVIOUS
14.1 Toggling the Timer
HOME
BOOK INDEX
NEXT
14.3 Retrieving and Displaying the Elapsed Time

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

14.3 Retrieving and Displaying the Elapsed Time

PLVtmr offers several different programs to retrieve and display the elapsed time: **elapsed**, **elapsed_message**, and **show_elapsed**. To get a non−NULL result from these programs, you must first have called the **capture** procedure.

The **elapsed** function returns the number of hundredths of seconds since the last call to **capture**. Its header is:

```
FUNCTION elapsed RETURN NUMBER;
```

You will want to use **elapsed** when you wish to store the elapsed time, rather than display it, or when you wish to display it in a format not supported by the other programs of PLVtmr.

14.3.1 elapsed_message Function

The **elapsed_message** function returns the elapsed time displayed within a standard format that incorporates the iteration factor (see Section 14.4, "Using PLVtmr in Scripts"). The header for **elapsed_message** follows:

```
FUNCTION elapsed_message
   (prefix_in IN VARCHAR2 := NULL,
    adjust_in IN NUMBER := 0,
    reset_in IN BOOLEAN := TRUE,
    reset_context_in IN VARCHAR2 := NULL)
RETURN VARCHAR2;
```

The first argument, **prefix_in**, is another "context" string. In addition to the string provided in the call to **capture**, you can pass additional information to the PLVtmr package to display with the elapsed timing message. The value of the **adjust_in** argument allows you adjust or calibrate your timing with any overhead computations you may have made. The **reset_in** argument indicates whether or not the **capture** procedure should be called to "reset" the starting time of the next elapsed timing. The final argument, **reset_context_in**, provides a starting context for that reset call to **capture**.

Notice that default values are provided for every argument to **elapsed_message**. I can, therefore, execute it as simply as this:

```
v_howmuch := PLVtmr.elapsed_message;
```

The impact of this call to **elapsed_message** will be to return a message that contains the elapsed time without any prefix string, unadjusted for any overhead, with the start time that is maintained by PLVtmr reset by a call to the **PLVtmr.capture** procedure.

If you want to obtain the elapsed timing information without automatically calling **capture** to reset the start time, pass FALSE for **reset_in** as shown below:

```
v_howmuch := PLVtmr.elapsed_message (reset_in => FALSE);
```

This next call to **elapsed_message** supplies a full set of arguments:

```
v_howmuch := PLVtmr.elapsed_message
                (TO_CHAR (v_empid), v_compcalc, TRUE, 'Profit Share');
```

In this call, a prefix of the current employee ID number is provided to be placed in the message string. The timing is also adjusted by the amount of time it took to calculate the compensation (**v_compcalc**). I request that **capture** be called to start the clock ticking again from a new start time. Finally, I associate the string "Profit Share" with the new timing session.

For examples of the format of the output from **elapsed_message**, see the next session on the **show_elapsed** procedure.

14.3.2 show_elapsed Procedure

The **show_elapsed** procedure is the highest−level mechanism for displayed elapsed timings. It relies on the DBMS_OUTPUT.PUT_LINE procedure (through the **p** package) to display the information. If you want to use another mechanism for displaying the elapsed timing (such as MESSAGE in Oracle Forms), you will simply call **elapsed** or **elapsed_message** and display that information.

The header for **show_elapsed** is:

```
PROCEDURE show_elapsed
   (prefix_in IN VARCHAR2 := NULL,
    adjust_in IN NUMBER := 0,
    reset_in IN BOOLEAN := TRUE);
```

The first argument, **prefix_in**, is another "context" string. In addition to the string provided in the call to **capture**, you can pass additional information to the PLVtmr package to display with the elapsed timing message. The **adjust_in** value allows you adjust or calibrate your timing with any overhead computations you may have made. The **reset_in** argument indicates whether or not the **capture** procedure should be called to "reset" the starting time of the next elapsed timing.

As with **elapsed_message**, all arguments have defaults, so you can simply specify:

```
PLVtmr.show_elapsed;
```

to see the number of hundredths of seconds elapsed since the last call to **capture** without any adjustment. You will also reset the start time.

The following SQL*Plus−based calls to the PLVtmr programs will give you a sense of the format used to display the timing information:

```
SQL> exec PLVtmr.capture ('calc totals');
SQL> exec PLVtmr.show_elapsed;
Elapsed since calc totals: 5.28 seconds.
SQL>  exec PLVtmr.capture ('calc totals');
SQL>  exec PLVtmr.show_elapsed ('year 1995');
year 1995 Elapsed since calc totals: 9.45 seconds.
```

14.3.3 Setting the Timing Factor

In many cases (and as demonstrated in Section 14.4), you will want to calculate the performance of an element of code which executes very quickly. In order to gain a true sense of its performance, you will put this code inside a loop and execute it multiple times.

You can use the factoring programs of PLVtmr to tell PLVtmr the number of times you are executing your code. It will then factor that number into its presentation of elapsed time.

The factoring programs are as follows:

```
PROCEDURE set_factor (factor_in IN NUMBER);
FUNCTION factor RETURN NUMBER;
```

The **`set_factor`** procedure sets the factoring value. The **`factor`** function returns the current setting.

Here is an example of how the use of the factoring value affects the output from the PLVtmr package:

```
SQL> exec PLVtmr.set_factor(100);
SQL> exec PLVtmr.capture ('calc totals');
SQL> exec PLVtmr.show_elapsed ('year 1995');
year 1995 Elapsed since calc totals: 3.79 seconds. Factored: .0379 seconds.
```

PREVIOUS | HOME | NEXT
14.2 Capturing the Start Time | BOOK INDEX | 14.4 Using PLVtmr in Scripts

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8*i* Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

14.4 Using PLVtmr in Scripts

In most situations, you will not place calls to PLVtmr inside your production code. Instead, you will extract specific elements of your application which you wish to focus on and understand their performance implications. You will usually write a SQL*Plus script that executes your code one or more times. If you do place the code within a loop, you should use the **`set_factor`** procedure to let PLVtmr know that it is timing multiple iterations of the code.

The following anonymous block, for example, calculates how long it takes to calculate totals. It also computes an average execution time over the specified number of iterations (passed in as a SQL*Plus argument) by calling the **`set_factor`** procedure:

```
BEGIN
   PLVtmr.set_factor (&1);
   PLVtmr.capture;
   FOR rep IN 1 .. &1
   LOOP
      calc_totals;
   END LOOP;
   PLVtmr.show_elapsed ('calc_totals');
END;
/
```

The PLVgen package will generate a loop like the one you see above. In fact, that script *was* generated with the following call in SQL*Plus:

```
SQL> exec PLVgen.timer ('calc_totals');
```

14.4.1 Comparing Performance of Different Implementations

Another common operation with PLVtmr is to compare two or more implementations of the same business rule or function. One example of this approach is shown below. In this script, I see which of my implementations of an "is number" function is most efficient. The first version is based on the TO_NUMBER builtin, while the second uses the LTRIM function.

```
SET SERVEROUTPUT ON
SET VERIFY OFF
DECLARE
   stg VARCHAR2(66) := '&2';
   bool BOOLEAN;
BEGIN
   PLVtmr.capture;
   FOR i IN 1 .. &1
   LOOP
      bool := isnum.tonumber (stg);
      IF i = 1 THEN do.pl(bool); END IF;
   END LOOP;
   PLVtmr.show_elapsed ('tonumber');

   PLVtmr.capture;
   FOR i IN 1 .. &1
   LOOP
      bool := isnum.trim (stg);
      IF i = 1 THEN do.pl(bool); END IF;
   END LOOP;
   PLVtmr.show_elapsed ('trim');
END;
/
```

14.4.2 Calculating Overhead of an Action

You can also use PLVtmr to calculate the overhead associated with a given operation. One example of this approach is shown by the **`func`** procedure of PLVtmr:

```
PROCEDURE func
IS
   myval NUMBER;
   baseval NUMBER;
BEGIN
   PLVtmr.capture;
   FOR rep IN 1 .. v_repeats
   LOOP
      myval := 0;
   END LOOP;
   baseval := elapsed;
   PLVtmr.capture;
   FOR rep IN 1 .. v_repeats
   LOOP
      myval := numval;
   END LOOP;
   show_elapsed ('Function Overhead', baseval);
END;
```

In this procedure, I compute the overhead associated with calling a function (versus making a direct assignment). I execute two different loops the number of times specified by **`v_repeats`** (which is set by the **`set_repeats`** procedure). In the first loop I obtain a baseline in which an assignment is executed. Rather than display that value, I simply assign to a local variable, **`baseval`**. I then execute a loop in which the function is called in place of the assignment. When this loop is completed, I display the elapsed time, passing the **`baseval`** variable as the amount by which the total elapsed time should be adjusted.

Here is an example of an execution of the **`func`** procedure three times, each based on 10,000 iterations. It shows that you can expect to incur upwards of 1/2 of one−thousandth of a second to make a function call (this was on a Pentium 90 Mhz laptop).

```
SQL> exec PLVtmr.set_repeats(10000);
SQL> exec PLVtmr.func
Function Overhead Elapsed: 5.33 seconds. Factored: .00053 seconds.
SQL> exec PLVtmr.func
Function Overhead Elapsed: 4.73 seconds. Factored: .00047 seconds.
SQL> exec PLVtmr.func
Function Overhead Elapsed: 4.56 seconds. Factored: .00046 seconds.
```

In fact, PLVtmr offers a number of programs to perform these kinds of comparisons: **`calibrate`**, **`currsucc`**, **`currfail`**, and, of course, **`func`**. In all cases, when you use PLVtmr to analyze performance, you should execute your test multiple times to make sure that your results stabilize around a consistent answer.

PREVIOUS | HOME | NEXT
14.3 Retrieving and Displaying the Elapsed Time | BOOK INDEX | 15. PLVvu: Viewing Source Code and Compile Errors

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8*i* Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

Chapter 15

15. PLVvu: Viewing Source Code and Compile Errors

Contents:

The PLVvu (PL/Vision View) package offers a set of programs which allow you to view both stored source code and compile errors. It provides an alternative to the SQL*Plus SHOW ERRORS command which offers you significantly more information about your compile problem. It allows you to quickly scan PL/SQL source code stored in the data dictionary.

This chapter shows you how to use the PLVvu programs and also explores the steps involved in constructing a utility like PLVvu. Before diving into the programs provided by PLVvu, however, let's review the situation most PL/SQL developers around the world face on a daily basis as they try to compile their code.

15.1 Compiling PL/SQL Code in SQL*Plus

Suppose that you work really, really hard at building this very complicated PL/SQL program. It's a big one, but you feel as if you've got a handle on it. You "create or replace" it in SQL*Plus and here is what you see:

```
SQL> start bigone.sql
Warning: Procedure created with compilation errors.
```

You groan. At the same time, you realize that you weren't very likely to get it all right the first time. Well, it's time to find out what the error is. Fortunately, Oracle Corporation provides a utility to view the compile errors: the SHOW ERRORS command.

```
SQL> show errors
LINE/COL ERROR
-------- --------------------------------------------------------------
624/10   PLS-00103: Encountered the symbol "IF" when expecting one of
         the following:
         * & = - + ; < / > in mod not rem an exponent (**)
         <> or != or ~= >= <= <> and or like between is null etc.
         ; was inserted before "IF" to continue.
```

Wow. Underwhelmed or what? Let's see...so, this error of some kind was found on the tenth character of line 624. Line 624, eh? I open up the file containing my fantastic new program and go down to line 624. Here is what I find:

```
622  FOR data_rec IN data_cur
623  LOOP
624     restructure (data_rec.key_val);
635  END LOOP;
```

Not an IF in sight. My sense of elation deflates. Getting this program to compile is going to be more difficult that I had thought. What is going on and why am I so depressed?

The most critical problem is that SHOW ERRORS does not actually show the line of code upon which the error was found. And even if it did show you that line, it might not necessarily reveal the error, since the error might actually occur on a *different* line, as you saw above.

15.1.1 Compiling Stored Code

When you compile a PL/SQL program in SQL*Plus from a file, the following actions occur:

1. SQL*Plus strips out all blank lines (!) and passes them on to the SQL layer ("create or replace" is a DDL statement).

2. The PL/SQL program is compiled. The source code in the file is loaded into the data dictionary in the SYS.SOURCE$, which has the following structure:

   ```
          Name           Null?    Type
          -------------- -------- ----
          OBJ#           NOT NULL NUMBER
          LINE           NOT NULL NUMBER
          SOURCE                  VARCHAR2(2000)
   ```

   When the compile is complete, the SYS.OBJECT$ table is updated with the date and time of the compile and the status (VALID or INVALID).

3. If there are compile errors, then that information is written to the SYS.ERROR$ table, which has the following structure:

   ```
   Name           Null?    Type
   -------------- -------- ----
   OBJ#           NOT NULL NUMBER
   SEQUENCE       NOT NULL NUMBER
   LINE           NOT NULL NUMBER
   POSITION       NOT NULL NUMBER
   TEXTLENGTH     NOT NULL NUMBER
   TEXT           NOT NULL VARCHAR2(2000)
   ```

The LINE column shows the line on which the error was found. The POSITION column contains the character offset to the token on which the error was found. Sadly, that line number reflects the "stripped" version of my program. So it doesn't correlate back to the source code in the file.

15.1.2 What SHOW ERRORS Does

The SHOW ERRORS command simply dumps the contents of SYS.ERROR$ (known, by the way, to mere mortals as the USER_ERRORS view) for the most recently compiled module. You can also display lines from USER_ERRORS for a specific program by specifying the type and name of the program, as shown:

```
SQL> show errors procedure greetings
```

This comes in handy when you have compiled (or tried to compile) multiple modules from a single script file. I am really glad that Oracle Corporation provides SHOW ERRORS, but I sure wish it were more useful. Even getting the line number on which the error occurs is not all that helpful. Sure, I can check my source code (usually in an operating system file). Yet my file line numbers will probably not match the stored code line numbers since SQL*Plus removes blank lines at compile time. I can write a query against USER_SOURCE to see my stored code, but what would be really great is if the SHOW ERRORS command at least showed the source code with which PL/SQL had its problem.

Wishful thinking does not, however, help a developer very much. I could wait until Oracle Corporation gets around to enhancing SHOW ERRORS, or maybe I could do something about it myself right now. I have learned over the years[1] two important lessons:

[1] This dates from 1991 when I built my own debugger for SQL*Forms, XRay Vision, in SQL*Forms itself.

1. Don't wait for Oracle Corporation to provide the finishing touches on products that improve developer productivity and general quality of life. Those enhancement requests are usually way down on the list of priorities.

2. I can usually build some kind of utility that goes a long way towards addressing a deficiency in the Oracle tools. It's not the same as Oracle really doing it right and it's not as polished or "shrink−wrapped" as a real third−party vendor solution, but it can still have a noticeable impact on my productivity.

The next section offers an alternative to SHOW ERRORS that handles many of the problems of this builtin command. This package should come in very handy, and it should also serve as a lesson (maybe even an inspiration) to all of my readers out there: don't whine, design! If you've got a complaint and you've got a need, take development into your own hands and build yourself a solution.

PREVIOUS | HOME | NEXT

14.4 Using PLVtmr in Scripts | BOOK INDEX | 15.2 Displaying Compile Errors

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

15.2 Displaying Compile Errors

The **err** procedure of the PLVvu package offers a very useful alternative to the SQL*Plus SHOW ERRORS command. The header for this procedure is:

```
PROCEDURE err
   (name_in IN VARCHAR2 := NULL, overlap_in IN INTEGER := overlap);
```

You provide the name of the program unit for which you want errors displayed and the **err** procedure not only displays all errors found in the USER_ERRORS view, but it also shows you exactly which lines of code are causing the problem. The second argument specifies the numbers of lines of code to display around the line with the compile error. The default value is the value set by the **set_overlap** procedure (described in a later section).

If you do not supply a program name, **PLVvu.err** will show you the compile errors for the most recently−compiled program unit. It determines this information by searching for the object in ALL_OBJECTS whose **last_ddl_time** equals the MAX (**last_ddl_time**).

The format for specifying a program unit is explained fully in Chapter 11, *PLVobj: A Packaged Interface to ALL_OBJECTS*. Briefly, you can supply only the name, the *type:name* (as in "b:PLVio" for the body of the PLVio package), or even the *type:schema.name* (as in "s:scott.showemps" to see the specification of the **showemps** package owned by SCOTT).

The **err** procedure tries to be smart about displaying the surrounding lines of code. Suppose, for example, that you have errors on two consecutive lines (318 and 319) and you have specified 10 lines of overlap. You would not want to see lines 309 through 318 as well as 319 through 328, twice, would you? The logic required to handle this complexity is covered in Section 15.4.2, "Implementing the SHOW ERRORS Alternative"

PL/Vision also provides a script named **sherr.sql** so that you do not have to type the full execute command for the **PLVvu.err** procedure at the SQL*Plus prompt. The following two requests to show errors are, therefore, equivalent:

```
SQL> exec PLVvu.err
SQL> @sherr
```

If you want to pass the name of a particular program to **PLVvu.err**, you will not be able to use the **sherr.sql** script.

15.2.1 Impact of PLVvu.err

Consider the SQL*Plus session shown below. First, we have the output from SHOW ERRORS. It reveals that compile errors were found on lines 333 and 349. As usual, it is very difficult to determine from these error messages what is actually wrong with my program and how I should go about finding the source of the problems. It is hard to even tell what the problem is because SHOW ERRORS does not display the line of code in which the error was found.

```
SQL> show errors
Errors for PACKAGE BODY PLVGEN:
LINE/COL ERROR
-------- --------------------------------------------------------------
333/53   PLS-00103: Encountered the symbol ";" when expecting one of
         the following:
         . ( ) , * @ % & | = - + < / > in mod not rem => ..
         an exponent (**) <> or != or ~= >= <= <> and or like etc.
         ) was inserted before ";" to continue.
349/4    PLS-00103: Encountered the symbol "BEGIN" when expecting one
         of the following:
         begin end function package pragma procedure form
         Replacing "BEGIN" with "begin".
```

Now let's check out the PLVvu alternative. In the following SQL*Plus session, instead of typing SHOW ERRORS, I call the **PLVvu.err** procedure. Since I do not provide a program name, it automatically locates the last compiled object, finds some errors, and displays the information.

```
SQL> exec PLVvu.err
-----------------------------------------------------------------------
PL/Vision Error Listing for PACKAGE BODY PLVGEN
-----------------------------------------------------------------------
Line#  Source
-----------------------------------------------------------------------
  331          put_line;
  332       END IF;
  333       PLVio.put_line (indent_stg (plus_in) || stg_in;
ERR                                                        *
    PLS-00103: Encountered the symbol ";" when expecting one of
    the following:  . ( ) , * @ % & | = - + < / > in mod not rem
     => ..     an exponent (**) <> or != or ~= >= <= <> and or
    like etc.  ) was inserted before ";" to continue.
  334       IF blanks_in IN (c_both, c_after)
  335       THEN
-----------------------------------------------------------------------
  347        plus_in IN INTEGER := 0,
  348        blanks_in IN VARCHAR2 := c_none);
  349    IS
ERR       *
    PLS-00103: Encountered the symbol "BEGIN" when expecting one
    of the following:  begin end function package pragma
    procedure form Replacing "BEGIN" with "begin".
  350    BEGIN
  351       IF using_cmnt
-----------------------------------------------------------------------
```

Well, that certainly looks different! The same two error messages show up (immediately after the word ERR on the left margin of the output). But in addition to those error messages, the **PLVvu.err** procedure displays the line of source code with which PL/SQL had a problem, as well as two lines before and after that problematic line of code.

With this added data, it is very easy for me to see what went wrong:

1. I left off the right parenthesis in my call to PLV**io.put_line**.

2. I included a semicolon at the end of line 348 −− this usually happens when I copy the header for a program out of the package specification and then paste it into the body and forget to remove the semicolon.

Notice that in both these cases the lines of code that were in error were not the lines of code indicated by the compiler. Hey, no compiler's perfect, right? In any case, I am able to immediately return to my source code and make the corrections. I have found that the **PLVvu.err** procedure can save me a solid 30 seconds each time I run into a compile error −− and believe me, I run into lots of compile errors with my code. This utility alone saves me an incredible amount of development time.

15.2.2 Setting the Code Overlap

You can specify the number of lines to display around the line in error by using the **set_overlap** procedure:

```
PROCEDURE set_overlap (size_in IN INTEGER := c_overlap);
```

The default value for the single size argument is provided by the package constant, **c_overlap**, which has a value of 5. So if you never call **set_overlap**, PLVvu will display five lines of code before and after a line with a compile error.

The following line of code changes the overlap to only two lines:

```
PLVvu.set_overlap (2);
```

And this call to **set_overlap** changes the overlap back to the default, since no value is provided.

```
PLVvu.set_overlap;
```

You can obtain the current value of the overlap by calling the **overlap** function:

```
FUNCTION overlap RETURN INTEGER;
```

If you don't like the default value of five lines, you might include a call to **PLVvu.set_overlap** in your **login.sql** to make sure that it is always set the way you like it in SQL*Plus.

PREVIOUS | HOME | NEXT
15.1 Compiling PL/SQL Code in SQL*Plus | BOOK INDEX | 15.3 Displaying Source Code

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

15.3 Displaying Source Code

PLVvu provides two procedures to display source code: the **code** and **code_after** programs. The **code** program displays all the lines of code for the specified program unit found between the start and end lines specified. The **code_after** program displays the specified number of lines found after the *n* th occurrence of a particular string. They are both explained below.

15.3.1 Displaying Code by Line Number

The header for the **code** program is:

```
PROCEDURE code
   (name_in IN VARCHAR2 := NULL,
    start_in IN INTEGER := 1,
    end_in IN INTEGER := NULL,
    header_in IN VARCHAR2 := 'Code for');
```

The first argument is the name of the program unit. If you do not supply a program, **PLVvu.code** will use the object last compiled into the database.

The format for specifying a program unit is explained in Chapter 11. Briefly, you can supply only the name, the *type:name* (as in "b:PLVio" for the body of the PLVio package), or even the *type:schema.name* (as in "s:scott.showemps" to see the specification of the **showemps** package owned by SCOTT).

The second and third arguments provide the range of line numbers of the code to be displayed. The default is all lines, with the start value of 1 and the end value NULL. The final argument provides a prefix for the output's header.

If you want to see all the lines of source code for a program unit, simply pass the program name and leave all the other arguments as the default. This approach is shown below:

```
SQL> exec PLVvu.code('s:testcase');
 --------------------------------------------------------------------
 Code for PACKAGE TESTCASE
 --------------------------------------------------------------------
 Line#  Source
 --------------------------------------------------------------------
     1 package testcase
     2 is
     3    procedure save (string_in in varchar2);
     4 end testcase;
```

The next call to **PLVvu.code** requests that it display lines 85 through 95 of the body of the PLVvu package.

```
SQL> exec PLVvu.code ('b:PLVvu', 85, 95, 'Contents of');
 ----------------------------------------------------------------------
 Contents of PACKAGE BODY PLVVU
 ----------------------------------------------------------------------
 Line#  Source
 ----------------------------------------------------------------------
    85     THEN
    86        p.l ('ERR' || LPAD ('*', err_rec.position+4));
    87        PLVprs.display_wrap
    88           (PLVchr.stripped (err_rec.text, PLVchr.newline_char),
    89            60, '    ');
    90     END IF;
    91     CLOSE err_cur;
    92  END;
    93  /*--------------- Public Modules -----------------*/
    94  PROCEDURE set_overlap
    95     (size_in IN INTEGER := c_overlap)
```

The **code.sql** SQL*Plus script allows you to skip some of the typing (and all of those irritating single quotes) when you use **PLVvu.code**. The last execution of **PLVvu.code**, for example, could be shortened to:

```
SQL> @code b:PLVvu 85 95
```

15.3.2 Displaying Code by Keyword

The code procedure is very useful and saves you the effort of putting together a quick SQL*Plus script to view lines of source code. However, scanning source code by line number ranges is not the only way you might want to locate and view your code. Another common method is to search for a keyword and then display the lines of code before, after, or around that keyword.

The **code_after** procedure displays the specified lines of code appearing after the *n*th occurrence of a keyword you provide. The header for **code_after** is:

```
PROCEDURE code_after
   (name_in IN VARCHAR2 := NULL,
    start_with_in IN VARCHAR2,
    num_lines_in IN INTEGER := overlap,
    nth_in IN INTEGER := 1)
```

The first argument is the name of the program unit. If you do not supply a program, **PLVvu.code** will use the object last compiled into the database.

The format for specifying a program unit is explained in Chapter 11. Briefly, you can supply only the name, the *type:name* (as in "b:PLVio" for the body of the PLVio package), or even the *type:schema.name* (as in "s:scott.showemps" to see the specification of the **showemps** package owned by SCOTT).

The second argument supplies the string for which **code_after** will search. The **num_lines_in** argument is the number of lines after the keyword is found that will be displayed (default provided by the current value of the overlap count). The last argument, **nth_in**, specifies the number of occurrences to be located before displaying the subsequent lines of code.

The following calls to **code_after** demonstrate the use of these different arguments. In the first example, I ask to see the default number of lines (5) following the first occurrence of SUBSTR. In the second call to **code_after**, I request to see only three lines following the fifth occurrence of SUBSTR.

```
SQL> exec PLVvu.code_after('b:PLVio','SUBSTR');
 ----------------------------------------------------------------------
 Code Starting with "SUBSTR" in PACKAGE BODY PLVIO
 ----------------------------------------------------------------------
 Line#  Source
 ----------------------------------------------------------------------
   330          (SUBSTR (srcrep.select_sql, 1, loc-1) ||
   331          srcrep.where_clause || ' ' ||
  332          SUBSTR (srcrep.select_sql, loc));
  333       ELSE
  334          RETURN srcrep.select_sql;
  335       END IF;

SQL> exec PLVvu.code_after('b:PLVio','SUBSTR', 3, 5);
 ----------------------------------------------------------------------
 Code Starting with "SUBSTR" in PACKAGE BODY PLVIO
 ----------------------------------------------------------------------
 Line#  Source
 ----------------------------------------------------------------------
   704                  SUBSTR
   705                     (string_repos.text_in,
   706                      string_repos.start_pos);
   707               string_repos.start_pos :=
```

Now you know how to use the **code** and **code_after** procedures to display your source code. Section 15.4, "Implementing PLVvu" shows you the techniques used to obtain this information.

Special Notes on PLVvu

Currently PLVvu only reads from the ALL_SOURCE data dictionary view to show source text with the **code** and **code_after** procedures. You cannot, for example, redirect the "source repository" to a file.

PREVIOUS | HOME | NEXT
15.2 Displaying Compile Errors | BOOK INDEX | 15.4 Implementing PLVvu

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

15.4 Implementing PLVvu

PLVvu is a very handy utility; it is also an excellent example of the kind of software you can build yourself to get more out of the Oracle data dictionary. This section goes behind the scenes of PLVvu to help you understand how I built the package −− and, perhaps as importantly, how the package evolved over time into its final PL/Vision format.

First we'll look at the general task of finding and displaying source code stored in the data dictionary. Then we'll examine the steps required to build an alternative to SHOW ERRORS.

15.4.1 How to Find Source Code

When you "create or replace" a program (procedure, function, or package) into the Oracle database, the source code is saved to the SYS.SOURCE$ table. You can view the contents of this table for all of your stored programs by accessing USER_SOURCE view. The structure of this view is:

```
SQL> desc user_source
 Name            Null?    Type
 --------------- -------- -------------
 NAME            NOT NULL VARCHAR2(30)
 TYPE                     VARCHAR2(12)
 LINE            NOT NULL NUMBER
 TEXT                     VARCHAR2(2000)
```

The *Name* column contains the name of the object. The name is always stored in uppercase unless you enclose the name of your program in double quotation marks at creation time. I will assume in my help implementation that you don't do this and that your program name is always uppercased. *Type* is a string describing the type of source code, either PROCEDURE, FUNCTION, PACKAGE, or PACKAGE BODY (always uppercase). The *line* is the line number and the *text* is the line of text. Notice that a line of text may be up to 2000 bytes in length.

You can also access another data dictionary view, ALL_SOURCE, to see the source of all programs you can access, even if you do not own those program units. It isn't hard to write a SQL statement to view the contents of this table. The following SQL script (found in **`showsrc.sql`**) gets the basic idea across:

```
SELECT TO_CHAR (line) || text Line_of_code
  FROM user_source
 WHERE name=UPPER ('&1')
   AND line BETWEEN &2 AND &3
/
```

However, if you want to make it easy for developers to run such scripts in a flexible manner, you will probably want to move to a PL/SQL−based solution, as I did.

The implementation of the **`PLVvu.code`** procedure is shown below:

```
PROCEDURE code
```

```
   (name_in IN VARCHAR2 := NULL,
    start_in IN INTEGER := 1,
    end_in IN INTEGER := NULL,
    header_in IN VARCHAR2 := 'Code for')
IS
   line_rec PLVio.line_type;
   line_num INTEGER;
BEGIN
   set_object (name_in);

   PLVio.asrc (start_in, end_in);
   disp_header (header_in);
   LOOP
      PLVio.get_line (line_rec, line_num);
      EXIT WHEN line_rec.eof;
      disp_text
         (line_rec.line# + start_in − 1, line_rec.text);
   END LOOP;

   PLVio.closesrc;
END;
```

As you can see, PLVvu relies heavily on other PL/Vision packages. It calls a private procedure, **`set_object`**, to set the current object for PLVio. The **`set_object`** procedure in turn calls the **`PLVobj.setcurr`** program for either the name specified or the last object. The code procedure then calls **`PLVio.asrc`** to set the source repository to the ALL_SOURCE view and specifies that only lines between **`start_in`** and **`end_in`** be queried. It displays a header and then loops through each record retrieved with a call to the **`PLVio.get_line`** procedure. It calls a local procedure, **`disp_text`**, to format and display the code. When done, it closes the source repository with a call to the **`PLVio.closesrc`** procedure.

This code is certainly more complex than the single query of the **`showsrc.sql`** script. It is, on the other hand, much more powerful and flexible. For example, this approach makes it quite feasible to enhance the **`PLVvu.code`** procedure to read the source code from an operating system file (remove the call to **`PLVio.asrc`** and let the user establish her own source repository outside of **`PLVvu.code`**). Now how would you do that with a straight SQL solution?

15.4.2 Implementing the SHOW ERRORS Alternative

I could start off by showing you the source code for **`PLVvu.err`** and simply step you through this final, polished version of the program. That approach would, however, be both misleading and intimidating. The programs that you see in these pages have been massaged quite thoroughly over a period of months. If I simply explain my final version without giving you a sense of the process by which I arrived at this pristine result, you will be much less likely to develop the skills needed to come up with solutions to your own, very specific problems.[2] As a result, I will make every effort to show you the techniques of problem−solving and iterative coding I have employed en route to producing this package.

> [2] In the process of writing this chapter, I performed an unplanned code review of my PLVvu package and performed major surgery on it, improving performance and code reuse. When I wrote "final, polished version" two paragraphs back, I had no idea how much room for improvement was left in PLVvu!

15.4.2.1 Merging source and error information

One of the critical early steps in designing a new package or new kind of utility is to validate the feasibility of the idea. In the case of **`PLVvu.err`**, my core idea is:

> The USER_SOURCE view contains the source code, including line number. The USER_ERRORS view contains the compile error information, which includes line numbers

as well. So it seems as if I should be able to combine these two sources of information into a single useful presentation like the one you saw earlier.

Let's see if I can actually do this. First, I need the SQL statement that will retrieve the source code from the USER_SOURCE view. The following SQL*Plus script displays all the code associated with the specified program unit (**&1** and **&2** are parameters, allowing values to be passed into the script).

```
COLUMN line FORMAT 99999
COLUMN text FORMAT A80
SELECT line, text
  FROM user_source
 WHERE name = UPPER ('&2')
   AND type = UPPER ('&1')
 ORDER BY line;
```

I can call this script (I'll name it **showcode.sql**) in SQL*Plus as follows:

```
SQL> start showcode procedure greetings
  LINE TEXT
------ ----------------------------------------------------------------
     1 procedure greetings
     2 is
     3 begin
     4    dbms_output.put_line ('hello world!')
     5 end;
```

Do you notice anything amiss in the above procedure? There is no semicolon at the end of line 4! In fact when I tried to "create or replace" this procedure, I was informed that:

```
Warning: Procedure created with compilation errors.
```

and my good friend SHOW ERRORS revealed the following:

```
LINE/COL ERROR
-------- --------------------------------------------------------------
5/1      PLS-00103: Encountered the symbol "END" when expecting one of
         the following:
         := . ( % ;
         Resuming parse at line 5, column 5.
```

If SHOW ERRORS can show the information, that means that there is a row in the USER_ERRORS view. Let's use SQL similar to the code in **showcode.sql** to show the error information stored in the view. The following script (**showerr.sql**) succeeds in producing output which matches SHOW ERRORS:

```
COLUMN linedesc FORMAT A8 HEADING 'LINE/COL'
COLUMN text FORMAT A62 HEADING 'ERROR'
SELECT TO_CHAR (line) || '/' || TO_CHAR(position) linedesc,
       text
  FROM user_errors
 WHERE name = UPPER ('&2')
   AND type = UPPER ('&1')
 ORDER BY line;
```

15.4.2.2 Showing the line in error

So these two SQL statements separately give me what I need -- but I need to combine them. The big question is now: can I merge these two SQL statements successfully? To merge data in SQL, I perform a join. The following SELECT will, therefore, return a line of text from USER_SOURCE for every row in the USER_ERRORS view.

```
SELECT TO_CHAR (S.line) || '/' || TO_CHAR(E.position) linedesc,
       S.text
  FROM user_errors E, user_source S
 WHERE E.name = UPPER ('&2')
   AND E.type = UPPER ('&1')
   AND E.name = S.name
   AND E.type = S.type
   AND E.line = S.line
 ORDER BY S.line;
```

When I run this script as **`showmerg.sql`** in SQL*Plus, I see the following:

```
SQL>  start showmerg procedure greetings
LINE/COL CODE
-------- -----------------------------------------------------------------
5/1      end;
```

So now I can display just those lines of code for which there are errors −− but what is the error? I seem to have lost that part of the equation. I haven't really merged my information yet; I have only used the USER_ERRORS view to "filter" out all those lines for which there are no errors. Furthermore, it is worth pointing out right now that even if I displayed the error along with the line displayed above, I wouldn't be giving myself a whole lot with which to work. Line 5, after all, is not really in error. That's just where the compiler got lost. So if I actually want to get something out of my substitute for SHOW ERRORS, I need to display at least a couple of lines around where the error occurred.

15.4.2.3 Going beyond SHOW ERRORS

There are two different approaches I can use to accomplish this goal: enhance the SQL or switch to a PL/SQL−based implementation. My most natural inclination is to move on to PL/SQL when the SQL going gets the least bit tough. Others, of course, may stick it out longer in the SQL layer. For example, it would be a relatively simple matter to replace the following WHERE clause of the **`showmerg.sql`** script:

```
AND E.line = S.line
```

with this BETWEEN operator:

```
AND S.line BETWEEN E.line−2 AND E.line+2
```

Then the SQL statement would display up to two lines on either side of the error−infested line of code, as shown below:

```
LINE/COL CODE
-------- -----------------------------------------------------------------
4/1      begin
5/1         dbms_output.put_line ('hello world!')
6/1      end;
```

I could even get really fancy (for me, anyway) and use DECODE to slip the error text in along with the source listing:

```
SELECT TO_CHAR (S.line) || '/' || TO_CHAR(E.position) linedesc,
       DECODE (S.line,
               E.line,
               S.text|| LPAD ('*', E.position) || CHR(10) || E.text,
               S.text) text
  FROM user_errors E, user_source S
 WHERE E.name = UPPER ('&2')
   AND E.type = UPPER ('&1')
   AND E.name = S.name
   AND E.type = S.type
   AND S.line BETWEEN E.line−2 AND E.line+2
 ORDER BY E.line;
```

Obligatory DECODE translation: If the source line number equals the error line number, then display the source followed immediately by a line with an asterisk under the position identified by the compiler as containing the error, followed by the error text. Otherwise just display the source text. With the above script (named **`merge2.sql`**), I generate this output for the greetings compile error:

```
SQL> start merge2 procedure greetings
LINE/COL CODE
-------- --------------------------------------------------------------
4/1      begin
5/1         dbms_output.put_line ('hello world!')
6/1      end;
         *
         PLS−00103: Encountered the symbol "END" when expecting one of
         the following:
          := . ( % ;
         ; was inserted before "END" to continue.
```

Pretty neat, huh? SQL, especially the version provided by Oracle Corporation can be very entertaining and effective. So why, you must be asking, should we bother with a PL/SQL−based implementation? Looks like this does everything we want...or does it?

15.4.2.4 Pushing the limits of an SQL solution

So far we have only tested our script with a very basic procedure and error scenario. Let's throw a couple of monkey wrenches at the **`merge2`** script and see what we get. For starters, let's simply change the mistake in the greetings procedure. Rather than leaving off the semicolon, I will remove the underscore from the builtin name **`put_line`**:

```
create or replace procedure greetings
is
begin
   dbms_output.putline ('hello world!');
end;
/
```

The output from **`merge2`** then becomes:

```
LINE/COL CODE
-------- --------------------------------------------------------------
2/16     is
2/4      is
3/16     begin
3/4      begin
4/16        dbms_output.putline ('hello world!');
                        *
         PLS−00302: component 'PUTLINE' must be declared
4/4         dbms_output.putline ('hello world!');
            *
         PL/SQL: Statement ignored
5/16     end;
5/4      end;
```

Lots of redundant lines of code! The reason for this extra output is that the compiler has actually generated two compile errors for the same line (one at position 4 and one at position 16).

```
4/4      PL/SQL: Statement ignored
4/16     PLS−00302: component 'PUTLINE' must be declared
```

And it just gets worse and worse from there. For example, if I add an OPEN statement to my greetings procedure and reference an undefined cursor, I will get four error messages from SHOW ERRORS:

```
4/4      PL/SQL: Statement ignored
4/16     PLS−00302: component 'PUTLINE' must be declared
5/2      PL/SQL: SQL Statement ignored
5/7      PLS−00201: identifier 'BLOB' must be declared
```

The output from a call to **merge2**, however, totals 18 lines (!) featuring both redundant, consecutive lines and extra "overlap" lines (the two lines preceding the second error include both the earlier error and its preceding line, so we get to see way too much and in the wrong order).

At moments like these, I thank my lucky stars for PL/SQL. There are clearly way too many exception conditions and special cases to stay within the non−procedural SQL environment. I need some old−fashioned explicit cursors with loops and IF statements. I need a procedural language.

15.4.2.5 Moving to a PL/SQL−based solution

With PL/SQL, I can relax. I am no longer constrained to come up with a creative/ingenious/obscure way to pack all my logic into a single, set−at−a−time SQL statement. I can employ traditional top−down design techniques to think through each layer of complexity. I will approach a PL/SQL solution in two stages: a direct evolutionary process from the SQL to a PL/SQL procedure (called **showerr**) and then a full−blown, PL/Vision−based implementation (**PLVvu.err**).

Here is the basic algorithm of my **showerr** procedure:

> A cursor FOR loop retrieves each of the rows from USER_ERRORS for a given program unit. For each error line retrieved, another cursor fetches lines of source code from USER_SOURCE which surround the line in error. DBMS_OUTPUT.PUT_LINE is then used to send all of these lines to the screen (or DBMS_OUTPUT buffer).

In top−down PL/SQL code we have:

```
FOR err_rec IN err_cur
LOOP
   FOR line_ind IN
       err_rec.line−2 .. err_rec.line+2
   LOOP
      disp_line(line_ind);
      IF line_ind = err_rec.line
      THEN
         /* Point to error, show message. */
         DBMS_OUTPUT.PUT_LINE
           (LPAD ('*', err_rec.position+8));
         DBMS_OUTPUT.PUT_LINE
           (RPAD ('ERR', 8) || err_rec.text);
      END IF;
   END LOOP;
END LOOP;
```

where the error cursor looks like this:

```
   CURSOR err_cur
   IS
      SELECT line, position, text
        FROM user_errors
       WHERE name = UPPER (name_in)
         AND type = UPPER (type_in)
       ORDER BY line;
```

The **disp_line** procedure displays the source for the specified line number. It simply fetches the row from USER_SOURCE and displays it with indentation using PUT_LINE.

```
PROCEDURE disp_line (line_in IN INTEGER)
IS
   CURSOR src_cur
   IS
     SELECT S.line, S.text
       FROM user_source S
      WHERE S.name = name_in
        AND S.type = type_in
        AND S.line = line_in;
   src_rec src_cur%ROWTYPE;
BEGIN
  OPEN src_cur;
  FETCH src_cur INTO src_rec;
  DBMS_OUTPUT.PUT_LINE
     (RPAD (TO_CHAR (line_in), 4) || src_rec.text);
  CLOSE src_cur;
END;
```

15.4.2.6 The devil in the details

This rapid, top−down driven design process yields a working program in short order. When I run this initial version, however, I find a number of complications (try it yourself; the code is stored in **`showerr1.sp`**):

1. When code is stored in the database, a newline character is placed on the end of each line. So when I use DBMS_OUTPUT.PUT_LINE to display the line, I end up with "double−spacing". SQL*Plus automatically strips off those newlines and **`showerr`** would have to do the same thing.

2. This program still shows multiple instances of the same lines of code, because I apply my "surround the error" algorithm for each error, even when these errors apply to the same line or consecutive lines.

3. The program automatically indents the source and error text to the eighth column. This works for the first line of a long error message, but all other lines (broken up by embedded newline characters) are justified flush with the left margin. It just doesn't look very professional.

4. The **`showerr`** procedure uses LPAD to put just the right number of spaces (based on the position column value) in front of an asterisk so as to point to the location in the line where the error was found. Yet when the program runs, the **`*`** always ends up on the first column. It turns out that when you display output with PUT_LINE in SQL*Plus, all leading spaces are trimmed. The LPAD is rendered useless.

The reason I detail all of these issues for you is to emphasize a truth that I learn over and over again (unfortunately) as I develop my software:

Everything is always more complicated than it seems at first glance.

The conclusions you should draw from this process are:

- Design your software to be as flexible and extensible as possible.

- Use top−down design to ensure that you call modules rather than execute an endless, complex series of statements.

-

Use packages from the get−go, because you are going to find that your standalone module really does belong with other programs in a package.

- When you build functions, think about different ways that they might be used, and design that wider scope into the initial implementation (taking care to avoid gratuitous scope−creep).

In the meantime, though, I have a **showerr** procedure that simply doesn't make the grade. Let's see what can be done to this standalone procedure to at least make it more useful than the original SQL solution.

15.4.2.7 Showing code just once

We will now enhance the **showerr** procedure found in **showerr1.sp**. The result is stored in the **showerr2.sp** file in the *use* subdirectory. To my mind, a sensible requirement for **showerr** would be that it never display a line of code more than once. If more than one error occurs on a line, the line is displayed once and multiple error messages are placed below it. If errors occur on consecutive lines, then the overlap of code around those lines should include those lines.

The easiest way to detect whether a line has already been displayed is to keep track of the last line number displayed. In the following version of the procedure's loops, the **last_line** variable is initialized to 0 and then set to the current line number after it is displayed. A line is now only displayed if it is greater than the last line number.

```
FOR err_rec IN err_cur
LOOP
   FOR line_ind IN
       err_rec.line−2 .. err_rec.line+2
   LOOP
      IF last_line < line_ind
      THEN
         /*
         || Display the source & error...
         || This must be changed too!
         */
      END IF;
      last_line := GREATEST (last_line, line_ind);
   END LOOP;
END LOOP;
```

As noted in the comment above, the logic required to display the source and error must now also be adjusted. If I am going to display a line of source code only once, then I have to take special care to make sure that all error messages are displayed. In the first version of **showerr**, I displayed the error when the inner FOR loop's line index equaled the outer cursor FOR loop error record's line number:

```
IF line_ind = err_rec.line
THEN
   ...
END IF;
```

This same test will no longer work. Suppose I have an error on lines 4 and 6. While the outer loop is still working with the error on line 4, the inner loop will have displayed line 6 and set the **last_line** to 6. When the outer loop moves on to line 6, the inner loop will not display this line a second time. So when and how will I display the error information for line 6?

15.4.2.8 Fine−tuning code display

I will need to discard the simplistic check for a match on error line number and current source line number. Instead, whenever I have not yet displayed a line, I will call the **display_line** procedure to show it and then also call a new local module, **`display_err`**, to display the error information if that line does indeed have an associated row in the USER_ERRORS view:

```
FOR err_rec IN err_cur
LOOP
   FOR line_ind IN
       err_rec.line-2 .. err_rec.line+2
   LOOP
      IF last_line < line_ind
      THEN
         display_line (line_ind);
         display_err (line_ind);
      END IF;
      last_line := GREATEST (last_line, line_ind);
   END LOOP;
END LOOP;
```

The **`display_err`** program is virtually identical to **`display_line`**, except that it fetches from USER_ERRORS, not USER_SOURCE, and shows the error position as well as the error information. This procedure is shown below:

```
PROCEDURE display_err (line_in IN INTEGER)
IS
   CURSOR err_cur
   IS
     SELECT line, position, text
       FROM user_errors
      WHERE name = UPPER (name_in)
        AND type = UPPER (type_in)
        AND line = line_in;
   err_rec err_cur%ROWTYPE;
BEGIN
  OPEN err_cur;
  FETCH err_cur INTO err_rec;
  IF err_cur%FOUND
  THEN
     DBMS_OUTPUT.PUT_LINE
        (LPAD ('*', err_rec.position+8));
     DBMS_OUTPUT.PUT_LINE
        (RTRIM (RPAD ('ERR', 8) || err_rec.text, CHR(10)));
  END IF;
  CLOSE err_cur;
END;
```

If the supplied line does not have an error, then the **`err_cur%FOUND`** attribute returns FALSE and nothing is displayed.

15.4.2.9 Tracing showerr execution

To understand more clearly the way that output is controlled in **`showerr2.sp`**, consider this scenario: my ten−line program has compile errors on line 4 and then again on line 6. Table 15.1 shows the progression of the counters and their impact on the display of information.

Table 15.1: Progression of Counters and Display Actions

Error Line (outer loop)	Line Ind (inner loop)	Last Line	Source Displayed?	Error Displayed?
4	2	0	Yes	No
4	3	2	Yes	No

4	4	3	Yes	Yes
4	5	4	Yes	No
4	6	5	Yes	Yes
6	4	6	No	No
6	5	6	No	No
6	6	6	No	No
6	7	6	Yes	No
6	8	7	Yes	No

No line is ever displayed twice but there still is a problem with the **showerr** loop: it will sometimes display one or more blank lines with "phony" line numbers. Suppose that a program has five lines of code and there is an error on line 4. The inner FOR loop will execute for line numbers 2 through 6. There isn't any line 7, so the **display_line** program will not fetch any records. It will, however, still go ahead and call DBMS_OUTPUT.PUT_LINE (see source above for the **display_line** procedure). This hole can be plugged as follows:

```
OPEN src_cur;
FETCH src_cur INTO src_rec;
IF src_cur%FOUND
THEN
   /* display the line */
END IF;
CLOSE src_cur;
```

Now **showerr** only displays a line once and then only if it is actually in the USER_SOURCE view. In addition, it displays the USER_ERRORS information for each line with an error, even if it is first displayed as the surrounding code for an earlier error.

15.4.2.10 Cleaning up the output

I identified earlier a number of ways in which the output from **showerr** was deficient. We've handled some of the most critical deficiencies. Let's take up some of the more cosmetic adjustments:

1. Properly position the asterisk for error position.
2. Get rid of blank lines.

You just can't get away with padding a string with spaces on the left and displaying that indentation. The spaces will be stripped out by the time the user sees the output. How do you get around this problem? Prefix your string with some non−blank characters, then stick in the requisite number of spaces to move your string to the right position.

The **p** package of PL/Vision has, in fact, a builtin prefix feature. But for **showerr** and our currently non−PL/Vision based implementation, we will use a prefix which makes the error stand out from the lines of source code and has a meaning specific to this program.

My approach is to use ERR as a prefix, as shown below:

```
4          dbms_output.putline ('hello world!');
ERR                    *
ERR     PLS−00302: component 'PUTLINE' must be declared
```

It's not hard to accomplish this. I simply change these calls to PUT_LINE:

```
DBMS_OUTPUT.PUT_LINE (LPAD ('*', err_rec.position+8));
DBMS_OUTPUT.PUT_LINE (LPAD (err_rec.text, 8));
```

to the following calls:

```
DBMS_OUTPUT.PUT_LINE ('ERR' || LPAD ('*', err_rec.position+5));
DBMS_OUTPUT.PUT_LINE (LPAD ('ERR', 8) || err_rec.text);
```

Now let's get rid of those blank lines. By default, when you display a line of code from USER_SOURCE, you will end up with a blank line after the code. This occurs because there actually is a newline character (CHR(10)) at the end of each line. So if you want to end up with a display of error which is actually readable, you will need to get rid of those trailing newlines. The best way to do this is with RTRIM, as shown below for the two different calls to PUT_LINE for code:

```
DBMS_OUTPUT.PUT_LINE
   (RTRIM (RPAD (TO_CHAR (line_in), 8) || src_rec.text, CHR (10)));
DBMS_OUTPUT.PUT_LINE
   (RTRIM (RPAD ('ERR', 8) || err_rec.text, CHR(10)));
```

15.4.2.11 Consolidating redundant code

Notice all the redundancies between these two calls? They cry out to be modularized into a single procedure:

```
PROCEDURE err_put_line
   (prefix_in IN VARCHAR2, text_in IN VARCHAR2)
IS
BEGIN
   DBMS_OUTPUT.PUT_LINE
      (RTRIM (RPAD (prefix_in, 8) || text_in, CHR(10)));
END;
```

With this procedure, the two calls to DBMS_OUTPUT.PUT_LINE become:

```
err_put_line (TO_CHAR (line_in), src_rec.txt);
err_put_line ('ERR', err_rec.text);
```

By taking this approach, I hide the specific implementation required to get my output correct (right−padding to length of eight, right−trimming the newline characters). This guarantees that the format of both these lines will be consistent with each other. And if I need to add extra lines as I enhance the procedure, I can simply call **`err_put_line`** and not have to worry about remembering all the details.

The only problem I identified in **`showerr1.sp`** which is not now corrected in **`showerr2.sp`** has to do with the way that long error messages are wrapped and then pushed to the left margin. The ideal solution would take the long error message, strip out newline characters and then rewrap the text at the specified line length. The techniques and code required to do this are described in *Chapter 11* of *Oracle PL/SQL Programming*, in *Character Function Examples*. I have also incorporated string−wrapping into the PLVprs package (PL/Vision PaRSe), as you will see in my presentation below of the PL/Vision version of **`showerr`**, namely **`Plvvu.err`**.

Even without word wrap, the **`showerr`** procedure now does a pretty fair job of enhancing SHOW ERRORS, as shown below:

```
SQL> exec showerr ('procedure','greetings');
2       is
3       begin
4          dbms_output.putline ('hello world!');
ERR                  *
ERR     PLS-00302: component 'PUTLINE' must be declared
5          open blob;
ERR             *
ERR     PLS-00201: identifier 'BLOB' must be declared
6       end;
```

Example 15.1 shows the full implementation of **showerr2.sp**'s version of the **showerr** procedure. Notice that **showerr** does not in any way depend on or make use of PL/Vision packages.

Example 15.1: The Final Version of showerr

```
CREATE OR REPLACE PROCEDURE showerr
   (type_in IN VARCHAR2, name_in IN VARCHAR2)
IS
   last_line INTEGER := 0;

   CURSOR err_cur
   IS
      SELECT line, text
        FROM user_errors
       WHERE name = UPPER (name_in)
         AND type = UPPER (type_in)
       ORDER BY line;

   /* Local Modules */

   PROCEDURE err_put_line
      (prefix_in IN VARCHAR2, text_in IN VARCHAR2)
   IS
   BEGIN
      DBMS_OUTPUT.PUT_LINE
         (RTRIM (RPAD (prefix_in, 8) || text_in, CHR(10)));
   END;

   PROCEDURE display_line (line_in IN INTEGER)
   IS
      CURSOR src_cur
      IS
        SELECT S.line, S.text
          FROM user_source S
         WHERE S.name = UPPER (name_in)
           AND S.type = UPPER (type_in)
           AND S.line = line_in;
      src_rec src_cur%ROWTYPE;
   BEGIN
     OPEN src_cur;
     FETCH src_cur INTO src_rec;
     IF src_cur%FOUND
     THEN
        err_put_line (TO_CHAR (line_in), src_rec.text);
     END IF;
     CLOSE src_cur;
   END;

   PROCEDURE display_err (line_in IN INTEGER)
   IS
      CURSOR err_cur
      IS
        SELECT line, position, text
          FROM user_errors
         WHERE name = UPPER (name_in)
           AND type = UPPER (type_in)
           AND line = line_in;
      err_rec err_cur%ROWTYPE;
   BEGIN
     OPEN err_cur;
```

```
      FETCH err_cur INTO err_rec;
      IF err_cur%FOUND
      THEN
         DBMS_OUTPUT.PUT_LINE ('ERR' || LPAD ('*', err_rec.position+5));
         err_put_line ('ERR', err_rec.text);
      END IF;
      CLOSE err_cur;
   END;

BEGIN
   /* Main body of procedure. Loop through all error lines. */
   FOR err_rec IN err_cur
   LOOP
      /* Show the surrounding code. */
      FOR line_ind IN err_rec.line−2 .. err_rec.line+2
      LOOP
         IF last_line < line_ind
         THEN
            display_line (line_ind);
            display_err (line_ind);
         END IF;
         last_line := GREATEST (last_line, line_ind);
      END LOOP;
   END LOOP;
END;
/
```

15.4.2.12 From prototype to finished product

We now have in hand a working prototype of a program to display more useful compile error information. It is a relatively sophisticated piece of code; it has three local modules, manipulates the contents of the data dictionary, and compensates for several idiosyncrasies of DBMS_OUTPUT in SQL*Plus. There is still a big difference, though, between a program which validates "proof of concept" and a polished utility that handles all circumstances gracefully.

I will list a few of the ideas I have uncovered to improve upon **showerr**. Then I will show you how I implemented some of these improvements with the PLVvu package (PL/Vision VU), which builds upon many other PL/Vision packages.

Here are ways I could improve the functionality and usability of **showerr**:

1. Reduce to an absolute minimum the typing required in order to get error information. So far it has been necessary to type all of this:

   ```
   SQL> exec showerr('procedure','greetings');
   ```

   to obtain the needed feedback. This is lots more typing than SHO ERR (the minimum required by SQL*Plus). If developers are really going to use my alternative, I know that it has to be as easy and painless to use as possible.

   With the PLVvu package, I can replace the above verbose request to "show errors of last compile" with the following:

   ```
   SQL> exec PLVvu.err
   ```

2. Improve the flexibility of **showerr**. For example, I have hard−coded at 2 the number of lines of code by which the error is surrounded. Why not let the developer specify the number of lines desired (setting the default at 2)? The PLVvu package provides this flexibility as the following example

shows (the request results in five lines of code before and after the line with the error):

```
SQL> exec PLVvu.set_overlap (5);
```

3. Spruce up the error display. Two examples implemented by PLVvu are: (a) include a header explaining what is being shown and (b) place a border in the error listing when lines of code have been skipped. Both of these techniques are illustrated below:

```
-----------------------------------------------------------------------
PL/Vision Error Listing for PROCEDURE TEMPPROC
-----------------------------------------------------------------------
Line#  Source
-----------------------------------------------------------------------
   11    PROCEDURE display_line (line_in IN INTEGER)
   12    IS
   13       CURSOR src_cur
ERR                 *
    PLS−00341: declaration of cursor 'SRC_CUR' is incomplete or
    malformed
   14       IS
   15         SELECT S.line, S.text
-----------------------------------------------------------------------
   32  end;
   33 BEGIN
   34    FOR err_rec IN err_cur1
ERR                      *
    PLS−00201: identifier 'ERR_CUR1' must be declared
   35    LOOP
   36       FOR line_ind IN err_rec.line−fff .. err_rec.line+2
-----------------------------------------------------------------------
```

`PLVvu.err` achieves improvements in ease of use and output appearance through two means:

1. Tweaking the logic used in **`showerr`** to handle certain conditions more gracefully. Compare the source code in **`plvvu.spb`** (in the *install* directory) with that of **`showerr2.sp`** (in the *use* directory) for examples of these differences.

2. Leveraging existing PL/Vision packages to both consolidate code and improve functionality. PLVobj is used to obtain the name of the last object compiled. PLVio and PLVobj are used to read the text from USER_SOURCE. PLVprs adds paragraph−wrapping capability to the display of long error messages.

There isn't anything really complicated in the body of **`PLVvu.err`**, because so much of it has been modularized −− either elsewhere within the PLVvu package or into other, prebuilt packages like PLVobj and PLVio.

As you build increasing numbers of generalized, reusable packages, you will find that the development and debugging time required for new programs decreases dramatically. Even though I do not explain in this chapter the complete implementations of my various PL/Vision packages, you should be able to see how they "plug−and−play". You can do the same thing in your own environment, to meet your own application−specific requirements.

PREVIOUS HOME NEXT
15.3 Displaying Source Code BOOK INDEX 16. PLVgen: Generating PL/SQL Programs

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8*i* Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

16. PLVgen: Generating PL/SQL Programs

Contents:

The PLVgen (PL/Vision GENerator) package provides a set of procedures you can use to generate your own PL/SQL code. With PLVgen you can:

- Implement coding standards and best practices by incorporating those elements into the code generated by the package.
- Improve developer productivity. You don't have to do as much typing; the code is generated for you −− and it comes free of typos.
- Take full advantage of PL/Vision programming components. When you generate code with PLVgen, it includes calls to PLVhlp, PLVtrc, and PLVexc package elements. PLVgen can make practical the widespread deployment of a library like PL/Vision.

16.1 Options for Best Practices

I teach a series of classes called "Achieving PL/SQL Excellence." I spend a lot of time in those classes talking about "best practices," the guidelines and techniques you should follow to write *excellent* PL/SQL programs. It's not enough to simply know how to write a procedure, function, or package. You need to know how to write those modules so that you are productive and so that the code is readable, efficient, and maintainable. That is a far more challenging task.

There are a couple of different options to implementing best practices:

1. *The manual (a.k.a. "hard") way:* Write or obtain a document that lists those best practices. Make sure that all developers study this guide and then set up a code review process to make sure that the standards and techniques have been followed.
2. *The automatic way:* Build or obtain a development environment that incorporates, generates, and automatically promotes the use of your best practices.

It should be pretty obvious to everyone which of these two options is preferable. Yet it is more than a matter of preference. The manual approach is also thoroughly impossible to apply with any degree of success. It requires a level of discipline and commitment from each developer that simply isn't practical. In addition, there are no tools available that allow you to do your code review in any practical fashion.

The automatic way is undeniably the way to go −− but who's going to get you going? After years of developer agony, third−party tools vendors and Oracle Corporation itself just getting around to offer a debugger. No one is addressing seriously how to improve the code *construction* phase. So the issue then becomes: what can you build yourself (or get from someone else) to improve your development environment and the quality of code written in your shop?

The answer, it turns out, is that you can build an amazingly useful array of utilities and components. While these home−grown solutions are not as powerful and easy to use as the real products we will eventually be able to purchase, they can have a dramatic impact on your work now. Various chapters in this book provide examples of such components, including an alternative to SHOW ERRORS and a mechanism to deliver online help for PL/SQL programs.

In this chapter, I present a package that generates PL/SQL code that can conform to an organization's standards; it enforces best practices by making it extremely easy to follow those best practices. I built PLVgen because I got tired of doing all the typing necessary to follow my own standards. I started to feel like a robot, and when that happens I know that there must be a way to automate what I am doing. You can easily build upon this package to support your own approaches to code.

16.1.1 Generating a Best Practice

I'll give you an example of how PLVgen has improved my life. One very important best practice in package construction is to *never* declare variables in the package specification (see Chapter 2, *Best Practices for Packages*). This means that the variable is "public" and can both be read and be modified directly by any user with execute authority on the package. Instead, you should declare variables inside the body of the package. Once you do this, however, you must provide get−and−set or "gas" routines in the specification to retrieve the value of the variable (get) and change the value of the variable (set). You must, in addition, declare the private variable and build the get−and−set routines in the package body.

Each of these programs is straightforward, but also each takes time to write −− especially if I want to get my formatting correct (use of upper− and lowercase, indentation, comments, etc.) and follow my templates for program structure. Before the advent of PLVgen, I would take the time to write this code, but would feel the minutes ticking by.

Now with the PLVgen package, I can generate all the code I need to do it right −− to hide my data and build the get−and−set code. The following command executed in SQL*Plus, for example, generates a get−and−set for a packaged variable named **`pagesize`**:

```
SQL> exec PLVgen.gas ('pagesize', 1, 25);
```

Depending on the toggles I have turned on for generated code content, the **`PLVgen.gas`** procedure could produce anywhere from 18 to 50 lines of perfectly formatted, bug−free code −− in seconds.

I am a rapid typist and I know my standards inside and out. Regardless, a call to PLVgen is far more efficient than anything I can achieve with the old−fashioned (pre−PL/Vision) approach. The advantages, however, even go beyond this efficiency. The PLVgen package makes use of another PL/Vision package, PLVio, to put a line on the screen. If you so desire, you can redirect the output of the package to a different target repository, including a PL/SQL table, a database table, and, with Release 2.3 of PL/SQL, a file. So you can plug−and−play PLVgen within a GUI interface that allows a developer to construct standards−smart and library−aware PL/SQL programs.

That should give you a feel for the advantages of generating code with PLVgen. In the next section I review the full set of program units and code fragments you can generate with PLVgen. Later, I'll present the techniques used to implement PLVgen.

PREVIOUS 15.4 Implementing PLVvu | HOME BOOK INDEX | NEXT 16.2 Code Generated by PLVgen

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8*i* Features | Oracle Built−in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built−ins Pocket Reference

16.2 Code Generated by PLVgen

With PLVgen, you can generate functions, procedures, packages, and variations on those. You have already seen examples of functions and procedures. The examples in this section demonstrate the other kinds of code generated from this package. You can also modify the contents of generated code by using the many different toggles.

16.2.1 Generating a Package

Don't get your hopes up too high. I can't generate a complete package for you. PLVgen can, on the other hand, generate a skeleton of the syntax required to build a package and also load it up with help text stubs and banners to break up the different elements of the package.

The header for the package generator is:

```
PROCEDURE pkg (name_in IN VARCHAR2);
```

You provide the name of the package, the procedure produced, and the code for a package (specification and body). Consider the following two commands executed in SQL*Plus:

```
SQL> exec PLVgen.usemax
SQL> exec PLVgen.pkg ('PLVfile');
```

Notice that I call **PLVgen.usemax** before generating my code. **PLVgen.usemax** turns on all available toggles for code content, including the CREATE OR REPLACE syntax, line numbers, program header, and help text stubs. These individual toggles and **usemax** are explained later in the chapter. The code generated by these commands is shown below.

```
CREATE OR REPLACE PACKAGE PLVfile
/*
|| Program: PLVfile
||  Author: Steven Feuerstein
||    File: PLVfile.SQL
|| Created: May 30, 1996 13:34:59
*/
/*HELP
Add help text here...
HELP*/

/*EXAMPLES
Add help text here...
EXAMPLES*/

IS
/* Public Data Structures */

/* Public Programs */

   PROCEDURE help (context_in IN VARCHAR2 := NULL);
```

```
22
23 END PLVfile;
24 /
25
26 CREATE OR REPLACE PACKAGE BODY PLVfile
27 IS
28 /* Private Data Structures */
29
30 /* Private Programs */
31
32 /* Public Programs */
33
34    PROCEDURE help (context_in IN VARCHAR2 := NULL)
35    IS
36    BEGIN
37       PLVhlp.show ('s:PLVfile', context_in);
38    END help;
39 END PLVfile;
40 /
```

As you can see, this generated package has all the syntax required to create package specifications and bodies (which is not, after all, very much syntax). It also contains a single procedure, **`help`** (lines 34 through 38), which relies on the PLVhlp package to provide online help about this package. It also creates stubs for the help text (lines 8 through 14), to remind you to add the text and make that information available (see Chapter 17, *PLVhlp: Online Help for PL/SQL Programs*, for more information on how PLVhlp works).

Finally, **`PLVgen.pkg`** provides banners in comments to delineate the different kinds of code one normally finds inside a package. These banners (lines 17, 19, 28, 30, and 32) help developers organize their code. This organization will, in turn, make it easier to develop, debug, and enhance the package.

16.2.2 Generating a Procedure

Once you have generated a package, you will most likely want to fill it up with procedures and functions. PLVgen provides the **`proc`** procedure to generate (as you might expect) procedures. Its header is:

```
PROCEDURE proc
   (name_in IN VARCHAR2,
    params_in IN VARCHAR2 := NULL,
    exec_in IN VARCHAR2 := NULL,
    incl_exc_in IN BOOLEAN := TRUE,
    indent_in IN INTEGER := 0);
```

The arguments of the **`proc`** procedure are explained below:

`name_in`
: The name of the procedure. The only required argument.

`params_in`
: The list of parameters to be enclosed in parentheses after the procedure name. The default is NULL (no parameters). The syntax of this argument must be a valid parameter list as would appear in the procedure definition (minus the parentheses).

`exec_in`
: One or more executable statements to place in the body of the procedure. If you want to provide more than one statement, you should concatenate a CHR(10) or **`PLVchr.newline_char`** between the statements to place them on separate lines. The default is NULL (no executable statements).

`incl_exc_in`

Pass TRUE (the default) to include an exception section and initial handler, FALSE to skip the exception section. Why give the user a choice? Sometimes an exception handler is just plain unnecessary and little more than clutter.

indent_in

The incremental indentation by which the entire procedure's code should be indented. The default is 0.

16.2.2.1 Maximum content of the generated procedure

Let's look at an example of procedure generation to get a feel for the way you can use the different arguments: generate a procedure with all the default argument values and the full set of additional code elements turned on by the call to **usemax** (explained later in this chapter).

```
SQL> exec PLVgen.usemax
SQL> exec PLVgen.proc ('calc_totals');
    1
    2    CREATE OR REPLACE PROCEDURE calc_totals
    3    /*
    4    || Program: calc_totals
    5    ||  Author: null
    6    ||    File: calc_totals.SQL
    7    || Created: May 29, 1996 13:36:20
    8    */
    9    /*HELP
   10    Add help text here...
   11    HELP*/
   12
   13    /*EXAMPLES
   14    Add help text here...
   15    EXAMPLES*/
   16
   17    IS
   18    BEGIN
   19       PLVtrc.startup ('calc_totals');
   20       PLVtrc.terminate;
   21
   22    EXCEPTION
   23       /* Call PLVexc in every handler. */
   24       WHEN OTHERS
   25       THEN
   26          PLVexc.rec_continue;
   27    END calc_totals;
   28    /
```

Let's go through this code line by line so that you can understand clearly the different elements of code that are generated for a procedure and function. I won't repeat this explanation (or the volume of code) in the following sections.

Line 2 contains the header for the procedure header, including the CREATE OR REPLACE syntax. As you will see in later examples, you can provide a parameter list as part of this header as well.

Lines 3 through 8 contain the standard program header provided by PLVgen. Notice that the author is automatically set to "Steven Feuerstein". I make sure of this setting by including the following statement in my **login.sql** script:

```
exec PLVgen.set_author ('Steven Feuerstein');
```

Lines 9 through 15 are the stubs for help text on two topics: general help and examples help. These comment blocks are used both for inline documentation of the package and also for online help text for users (through the PLVhlp package).

Lines 19 and 20 call PLVtrc **`startup`** and **`terminate`** procedures. These programs provide an execution trace that is of particular use in the PLVexc exception− handling package. Again, you do not have to include these programs in your own generated code, but it is an available option.

Lines 22 through 26 show the standard PL/Vision exception handler section. A single WHEN OTHERS clause calls the **`PLVexc.rec_continue`** program, which records the error and continues processing. You can replace this PLVexc handler with another one (such as **`PLVexc.rec_halt`** or "record and halt"); the point to recognize here is that the generator creates an exception handler as a starting point, which reinforces best practices.

Lines 27 and 28 bring the procedure to a close. Notice that PLVgen automatically supplies an END label. It also provides the forward slash, that in conjunction with the CREATE OR REPLACE syntax results in a script that can be used immediately in SQL*Plus to store and compile the procedure.

> ***NOTE:*** In most of the examples of generated code in the rest of this chapter, I turn off the many comments and other optional elements of the source code. By doing this, you can focus more easily on how PLVgen constructs the required elements of the PL/SQL programs. Section 16.3, "Modifying PLVgen Behavior" explains how to use some or all (**`PLVgen.usemax`**) of the optional elements of the code listed previously.

16.2.2.2 A procedure with parameter list and executable code

The following SQL*Plus session generates a procedure using a minimum of additional features, but with a parameter list and a single line of code in the body of the procedure.

```
SQL> exec PLVgen.usemin
SQL> exec PLVgen.proc ('disptot', 'comp_in in integer', 'get_total;');

PROCEDURE disptot (comp_in IN INTEGER)
IS
BEGIN
   get_total;

EXCEPTION
   WHEN OTHERS
   THEN
      NULL;
END disptot;
```

Notice that the keywords IN and INTEGER are automatically uppercased by PLVgen. This package makes use of the **`PLVcase.string`** program to convert the case of the parameter list and the executable line of code.

16.2.2.3 Changing indentation of generated code

In this final example of the procedure generator, I make use of the indentation parameter to add three spaces to the default three blanks provided by PLVgen. I could then easily cut and paste this text into the body of a package or into the declaration section of a nested, anonymous block.

```
SQL> exec PLVgen.usemin
SQL> exec PLVgen.proc ('disptot',indent_in=> 3);
      PROCEDURE disptot
      IS
      BEGIN

      EXCEPTION
         WHEN OTHERS
         THEN
            NULL;
      END disptot;
```

Notice that I use named notation (the => symbol associating **`indent_in`** with the value 3) so that I can avoid specifying values for all the other arguments.

16.2.3 Generating a Function

PLVgen offers seven overloadings of the function generator. Why? Because a function has a RETURN datatype and I want my package to automatically generate functions of the requested type. **`PLVgen.func`** generates functions with the following datatypes: BOOLEAN, DATE, NUMBER, and VARCHAR2. Four datatypes and seven overloadings...am I leaving something out? Not at all; PLVgen supports four different datatype functions, but provides additional overloadings to handle two kinds of default values for the functions (covered later in this section).

The format of the function generated by **`PLVgen.func`** follows my guidelines for a "template" function, the generic structure of which is shown below:

```
FUNCTION func_name (...) RETURN datatype
IS
   /* Variable for RETURN */
   return_value datatype;
BEGIN
   <executable statements>
   /* Last line always: */
   RETURN return_value;
EXCEPTION
   WHEN OTHERS
   THEN
      RETURN NULL;
END func_name;
```

A brief recap of the function template: There is a single RETURN statement in the body of the program and it is the last line of code. There is a local variable (in this case, **`return_value`**) that is always the same datatype as the function itself and is RETURNed in that single, successful RETURN statement. There is an exception section that returns NULL if something goes wrong.

Rather than show the headers for all of the overloadings here and then again in the package specification section, I will show you the header for the string function generator and the headers for the Boolean function generators. All the other nonstring function generators are structured and used like the Boolean version.

16.2.3.1 Function generator headers

Here is the header for the program that generates a string or VARCHAR2 function:

```
PROCEDURE func
   (name_in IN VARCHAR2,
    datadesc_in VARCHAR2,
    defval_in IN VARCHAR2 := NULL,
    length_in IN INTEGER := c_def_length,
    incl_exc_in IN BOOLEAN := TRUE);
```

Here are the headers for the two programs that generate Boolean functions:

```
PROCEDURE func
   (name_in IN VARCHAR2,
    datadesc_in BOOLEAN,
    defval_in IN BOOLEAN := NULL,
    incl_exc_in IN BOOLEAN := TRUE);

PROCEDURE func
```

```
   (name_in IN VARCHAR2,
    datadesc_in BOOLEAN,
    defval_in IN VARCHAR2,
    incl_exc_in IN BOOLEAN := TRUE);
```

In all of these programs, the **`name_in`** is the name of the function, and **`datadesc_in`** is an expression or value that is the same datatype as the function you want to generate. The **`defval_in`** argument provides a default value for the RETURN statement of the function. Notice that this is the only parameter that is different for the two overloaded Boolean function generators. In one case, the default value is a Boolean. In the second, the default value is a string. This distinction is explained below.

When generating a string function, you can also provide (through the **`length_in`** argument) the length of the VARCHAR2 variable to be declared. The default is provided by the constant **`c_def_length`** and is set to 100.

The **`incl_exc_in`** argument indicates whether or not an exception section should be included in the function. The default is "Yes! Include an exception section!" (and that is a very important thing to do in functions).

Now I will explain how to use these and the other function generators. The only information you have to provide to a function generator is the name and the datatype of the function.

16.2.3.2 Generated function examples

In this first example, I generate a string function named **`full_name`** simply by providing a string −− any string −− as the second argument.

```
SQL> exec PLVgen.func('full_name','a');
   FUNCTION full_name RETURN VARCHAR2
   IS
      retval VARCHAR2(100) := NULL;
   BEGIN
      RETURN retval;

   EXCEPTION
      WHEN OTHERS
      THEN
         RETURN NULL;
   END full_name;
```

> ***NOTE:*** See Section 16.2.2, "Generating a Procedure" for a more complete description of the elements of code that can be included in a generated function.

Notice that this function conforms to the template structure described earlier. Based on the datatype of the second argument, PLVgen has automatically determined that the RETURN datatype should be VARCHAR2. It has also declared a local variable with matching datatype and appropriate default value.

In the next call to **`PLVgen.func`**, I generate a Boolean function, this time setting the default value to FALSE and rejecting the use of an exception section.

```
SQL> exec PLVgen.func ('valid_account', true, false, FALSE);
   FUNCTION valid_account RETURN BOOLEAN
   IS
      retval BOOLEAN := FALSE;
   BEGIN
      RETURN retval;
   END valid_account;
```

Now I will generate this same function, but instead of providing a literal default value, I pass in a string that contains an expression (and include an exception section):

```
SQL> exec PLVgen.func
             ('valid_account', true, 'sysdate < account.start_date');
   FUNCTION valid_account RETURN BOOLEAN
   IS
      retval BOOLEAN := sysdate < account.start_date;
   BEGIN
      RETURN retval;

   EXCEPTION
      WHEN OTHERS
      THEN
         RETURN NULL;
   END valid_account;
```

Since the second argument to **`PLVgen.func`** was a Boolean and the third a string, the PL/SQL runtime engine executed the version of **`func`** that interprets the default value not as a literal, but as an expression.

And that is the reason for the overloading of the two different Boolean function generators. When the default value is a Boolean, that value is applied directly (as a TRUE, FALSE, or NULL) to the local variable declaration. If the default is a string, that string becomes the *unevaluated* expression used as the default value for the local variable.

The function generators for dates and numbers are overloaded and work in the same way as the Boolean procedures do.

16.2.3.3 Default values for string functions

PLVgen provides two distinct overloadings for Boolean, date, and number function generators to handle literals and expressions for default values. This same approach is not possible for the procedure that generates string functions. The way **`PLVgen.func`** supports expressions for default values is that they are passed as strings. But since the default value for a string function is already a string, another overloaded version with a default value of datatype VARCHAR2 to handle string expressions would simply not work.

A different approach is required for the string function generator. There is just one version of **`func`** to generate string functions, but the following special rule applies to the default value argument, namely: if the first character of the default value string is an equal sign (**`=`**), then the string is to be interpreted as an expression. In this case, the value in the string is not evaluated, but simply placed in the default value area for the local variable of the function.

Let's look at a couple of examples. In this first call to **`PLVgen.func`**, I generate a VARCHAR2 function with a default return value of ABC. Notice that since this is a string function, I also specify the maximize length of the return value and request that the function not have an exception section.

```
SQL> exec PLVgen.func ('full_name', 'a', 'ABC', 50, FALSE);
   FUNCTION full_name RETURN VARCHAR2
   IS
      retval VARCHAR2(50) := 'ABC';
   BEGIN

      RETURN retval;
   END full_name;
```

In this next call to **`PLVgen.func`**, I change the default value to prefix the "ABC" with the equal sign. In the resulting declaration of a local variable, the default value is set to the PL/SQL identifier named abc, and not to a literal with that value.

```
SQL> exec PLVgen.func ('full_name', 'a', '=ABC', 50, FALSE);
   FUNCTION full_name RETURN VARCHAR2
   IS
      retval VARCHAR2(50) := abc;
   BEGIN

      RETURN retval;
   END full_name;
```

Let's look at an example that involves a more real world use of this expression default value. In the following call to **PLVgen.func**, I create a string function that sets the default value to a package−based constant.

```
SQL>  exec PLVgen.func ('full_name', 'a', '=Names.formal_address');
   FUNCTION full_name RETURN VARCHAR2
   IS
      retval VARCHAR2(100) := names.formal_address;
   BEGIN
      RETURN retval;

   EXCEPTION
      WHEN OTHERS
      THEN
         RETURN NULL;
   END full_name;
```

If you ever want to generate a string function with a nonliteral default value, don't forget to prefix the default value string with an equal sign.

16.2.4 Generating Get−and−Set Routines

One of the most important of my best practices for package construction is to always hide package data behind a programmatic interface, otherwise known as get−and−set routines. Instead of declaring a variable directly in the specification, you would move that declaration to the body of the package and then build (a minimum of) two programs: a function to retrieve the current value of this variable and a procedure to change the value of the variable.

The **gas** (get−and−set) procedure of PLVgen generates the get−and−set code needed to hide variables of datatypes VARCHAR2, DATE, NUMBER, and Boolean. The overloading for the **gas** procedure is similar to that of the **func** procedure, which makes sense since **PLVgen.gas** does generate a function, as well as a variable declaration of the correct datatype. There are two versions of **gas** for every datatype *except* VARCHAR2. For this string datatype, there is only one overloading. The header for the string version of **gas** is shown below:

```
PROCEDURE gas
   (name_in IN VARCHAR2,
    valtype_in VARCHAR2,
    defval_in IN VARCHAR2 := NULL,
    length_in IN INTEGER := c_def_length);
```

The header for the NUMBER version of **gas** is shown below. The headers for the other datatypes are exactly the same as for the NUMBER version, except for the datatype of the **valtype_in** and **defval_in** arguments.

```
PROCEDURE gas
   (name_in IN VARCHAR2,
    valtype_in NUMBER,
    defval_in IN NUMBER := NULL);

PROCEDURE gas
   (name_in IN VARCHAR2,
    valtype_in NUMBER,
         defval_in IN VARCHAR2);
```

Generally, the first argument provides the name of the variable. The second argument determines the type of the function and the variable being hidden inside the package body. The third argument provides a default value for the variable. When generating a string get−and−set, you can also provide the length of the VARCHAR2 variable to be declared. The default is provided by the constant **`c_def_length`** and is set to 100.

The listing below gives you an idea of the kind of code that is generated by a call to the **`PLVgen.gas`** procedure. I call the **`PLVgen.useln`** to turn on line numbers to use as a reference (after turning off all other toggles).

```
SQL> exec PLVgen.usemin
SQL> exec PLVgen.useln
SQL> exec PLVgen.gas ('pagesize', 1, 25);
    1
    2    PROCEDURE set_pagesize (pagesize_in IN NUMBER);
    3    FUNCTION pagesize RETURN NUMBER;
    4
    5    v_pagesize NUMBER := 25;
    6
    7    PROCEDURE set_pagesize (pagesize_in IN NUMBER)
    8    IS
    9    BEGIN
   10       v_pagesize := pagesize_in;
   11    END set_pagesize;
   12
   13    FUNCTION pagesize RETURN NUMBER
   14    IS
   15       retval NUMBER := v_pagesize;
   16    BEGIN
   17       RETURN retval;
   18    END pagesize;
```

Lines 2 and 3 contain the headers for the set and get programs. These should be cut and pasted into the package specification. Line 5 contains the declaration of the variable that is to be protected by the get−and−set programs. This declaration is placed in the body of the package, before any of the program definitions are listed. Lines 7 through 11 contain the definition of the set program. Lines 13 through 18 contain the definition of the get program. Both of these should be cut and pasted into the package body. Notice that even in the simple get function, the template approach is still followed.

16.2.4.1 Generating a Boolean get−and−set

The naming scheme and structure for programs generated for a Boolean get−and−set is a bit different from that for strings, dates, and numbers. When building a get−and−set around a Boolean variable, you can and should take into account the fact that a user can only set the variable to one of three values (TRUE, FALSE, and NULL). In fact, in many cases, a NULL value is not allowed. If **`PLVgen.gas`** used the same syntax for Booleans as it does for numbers (see above listing), then the set program would look like this:

```
PROCEDURE set_show_changes (show_changes_in IN BOOLEAN);
```

and this program would be used as follows:

```
PKG_NAME.set_show_changes (TRUE);
PKG_NAME.set_show_changes (FALSE);
```

While there is certainly nothing wrong with this style, it strikes me as a cumbersome interface. A much cleaner, more natural style would allow me to call programs like this:

```
PKG_NAME.show_changes; −− Set variable to TRUE.
PKG_NAME.noshow_changes; -- Set variable to FALSE.
```

This is the very approach taken with the Boolean **gas** procedure. The full set of generated code (specification, variable declaration, and body code) is shown below for a **show_changes** variable:

```
SQL> exec PLVgen.gas ('show_changes', true, true);
   PROCEDURE show_changes;
   PROCEDURE noshow_changes;
   FUNCTION show_changesing RETURN BOOLEAN;

   v_show_changes BOOLEAN := TRUE;

   PROCEDURE show_changes
   IS
   BEGIN
      v_show_changes := TRUE;
   END show_changes;

   PROCEDURE noshow_changes
   IS
   BEGIN
      v_show_changes := FALSE;
   END no_show_changes;

   FUNCTION show_changesing RETURN BOOLEAN
   IS
      retval BOOLEAN := v_show_changes;
   BEGIN
      RETURN retval;
   END show_changesing;
```

Notice the third program, a function, in the get−and−set routines for a Boolean. This function is the get program and it returns the current value of the Boolean variable. I use the "ing" structure for the name of this function, as in: "Am I currently showing the changes?" PLVgen automatically appends an "ing" to the name of the variable passed in the call to the **gas** procedure. It is smart enough to convert some formats to a readable name. For example, if you request a get−and−set for the **use_lines** variable, **PLVgen.gas** generates a function with this header:

```
FUNCTION using_lines RETURN BOOLEAN;
```

and if you use **PLVgen.gas** with a variable name **propose**, it generates a function whose header is:

```
FUNCTION proposing RETURN BOOLEAN;
```

but PLVgen does not pretend to handle all nuances of the infinitely nuanced English language. As a result, variable names like **show_changes** result in a function named **show_changesing** and you will just have to rename it yourself!

16.2.4.2 Generating a toggle

A toggle is an on−off switch; it is actually a special case of the Boolean get−and−set routines. The toggle procedure generates the code required to implement this switch in PL/SQL.

The header for toggle is:

```
PROCEDURE toggle (name_in IN VARCHAR2 := NULL);
```

where **name_in** is the name of the toggle. If you do not specify a toggle name, the following code is generated:

```
SQL> exec PLVgen.usemin
SQL> exec PLVgen.useln
SQL> exec PLVgen.toggle
   PROCEDURE turn_on;
   PROCEDURE turn_off;
   FUNCTION turned_on RETURN BOOLEAN;

   v_onoff BOOLEAN := TRUE;

   PROCEDURE turn_on
   IS
   BEGIN
      v_onoff := TRUE;
   END turn_on;

   PROCEDURE turn_off
   IS
   BEGIN
      v_onoff := FALSE;
   END turn_off;

   FUNCTION turned_on RETURN BOOLEAN
   IS
      retval BOOLEAN := v_onoff;
   BEGIN
      RETURN retval;
   END turned_on;
```

You have generated, in other words, a generic toggle to turn something off or turn it on. You can also generate more specific toggles by providing a toggle name. If you pass a non−NULL toggle name, the get−and−set code matches those lines of source code generated by a call to **PLVgen.gas** for a Boolean variable.

16.2.5 Generating Help Stubs

PL/Vision provides a mechanism for delivering online help for your PL/SQL programs (see Chapter 17). To take advantage of this mechanism you need to (a) put comments with an appropriate format in your source code, and (b) provide an easy way for developers to ask for that help. PLVgen allows you to generate the code and comments to handle both these tasks.

16.2.5.1 Generating help text stubs

The **helptext** procedure generates a comment stub of the correct format so that the PLVhlp package can find and display the information. The header for **helptext** is:

```
PROCEDURE helptext
   (context_in IN VARCHAR2 := PLVhlp.c_main);
```

where **context_in** is the context or keyword that indicates the topic of the help text. The default value for this context is the main help topic constant from PLVhlp: HELP.

You can call **helptext** directly; it is also called by the various program unit generators to include stubs for help text in generated code.

To create a stub of help text for the main topic, you would do the following:

```
SQL> exec PLVgen.helptext;
   /*HELP
   Add help text here...
   HELP*/
```

To create a stub of help text for a different topic, such as KNOWN PROBLEMS, you would execute the following command:

```
SQL> exec PLVgen.helptext ('known problems');
   /*KNOWN PROBLEMS
   Add help text here...
   KNOWN PROBLEMS*/
```

You can then cut and paste these stubs anywhere in your program definition, add some meaningful help text, and CREATE OR REPLACE the program. Users of the program can then access this text through the **`PLVhlp.show`** procedure. To make it easier to get at this information, however, you will most likely generate and include a special help procedure in your program unit (if it is a package, anyway). You do this with the **`PLVgen.helpproc`** program.

16.2.5.2 Generating a help procedure

The **`helpproc`** procedure generates a program that provides help for a given program unit. The header for **`helpproc`** is:

```
PROCEDURE helpproc
   (prog_in IN VARCHAR2 := NULL, indent_in IN INTEGER := 0);
```

where **`prog_in`** is the name of the program and **`indent_in`** is an optional additional indentation. By default the generated code already has an indentation of three spaces. The following SQL*Plus session generates a program to show help text from the specification of the PLVio package.

```
SQL> exec PLVgen.helpproc ('s:PLVio');
   PROCEDURE help (context_in IN VARCHAR2 := NULL)
   IS
   BEGIN
      PLVhlp.show ('s:PLVio', context_in);
   END help;
```

Let's see how you would use **`helpproc`** to provide help to your users. Suppose you have built a package named **`emp_maint`**. You want to allow developers to ask for general help on the employee maintenance tasks available. So you first execute **`PLVgen.helptext`** to create a stub help block:

```
SQL> exec PLVgen.helptext;
   /*HELP
   Add help text here...
   HELP*/
```

You cut and paste this text into the specification of **`emp_maint`** and add some real text. Then you generate a help program as follows:

```
SQL> exec PLVgen.helpproc ('s:emp_maint');
   PROCEDURE help (context_in IN VARCHAR2 := NULL)
   IS
   BEGIN
      PLVhlp.show ('s:emp_maint', context_in);
   END help;
```

You then cut and paste this program into the body of the package. You must also cut only the header (first line) of the procedure and paste that into the specification of the package. Once the code is recompiled, a user of **`emp_maint`** can enter the following command to see the associated help text:

```
SQL> exec emp_maint.help
```

If, by the way, you used **`PLVgen.pkg`** to generate the first version of the **`emp_maint`** package, it would come with the help program and help text stubs already in place.

16.2.6 Generating a Cursor Declaration

Many of us have gotten pretty good about using consistent indentation and white space for our PL/SQL code. Yet when we come to writing a SQL statement, we abandon any pretense of formatting and disgorge an absolutely *awful* mish−mash of SQL text into our otherwise orderly procedural logic. I believe that a consistent, readable format for the SQL in our PL/SQL programs is even more important than a good format for the procedural part of the programs. To help you satisfy my whim, PLVgen provides the **curdecl** procedure, which generates the declaration statement for a cursor.

The header for **curdecl** is as follows:

```
PROCEDURE curdecl
   (cur_in IN VARCHAR2,
    ind_in IN INTEGER := 0,
    table_in IN VARCHAR2 := NULL,
    collist_in IN VARCHAR2 := NULL,
    gen_rec_in IN BOOLEAN := TRUE);
```

where **cur_in** is the name of the cursor, **ind_in** is an additional amount of indentation, **table_in** is the name of the table (or list of tables), and **collist_in** is the list of columns in the SELECT list of the SQL statement. The **gen_rec_in** argument specifies whether or not you want a record declaration to be generated with the cursor.

This procedure implements several best practices and naming conventions for cursors: it automatically appends "**_cur**" to the end of the cursor name and "**_rec**" to the end of the record name. It formats the SQL statement to make it easier to read. It provides a record to go with the cursor as a discouragement against declaring individual variables into which data is then fetched.

The more you take advantage of these different arguments, the more likely it is that you will generate a cursor declaration that is very close to being ready for execution. Let's look at some examples.

16.2.6.1 Cursor declaration examples

1. Generate a cursor declaration relying on all default parameter values.

```
SQL> exec PLVgen.curdecl('emp');
   CURSOR emp_cur
   IS
      SELECT
        FROM
       WHERE
         AND
       ORDER BY ;
   emp_rec emp_cur%ROWTYPE;
```

2. Generate a cursor against the order table to select the **order_id** and **ship_date**. Do not generate a record declaration since I am going to use this cursor in a FOR loop.

```
SQL> exec PLVgen.curdecl ('orders', 0, 'order', 'order_id, ship_date', FALSE);
   CURSOR orders_cur
   IS
      SELECT order_id, ship_date
        FROM order
       WHERE
         AND
       ORDER BY ;
```

The backbone of an SQL SELECT statement generated by **`curdecl`** supplies all the main clauses. You could greatly expand upon the PLVgen package to produce all manner of SQL statements.

16.2.7 Generating a "Record Found?" Function

The **`curdecl`** procedure generates a cursor declaration, which is certainly useful. Yet there are also many common kinds of code fragments and programs that use cursors in specific ways. One example is the "record found?" function. How many times have you written a function to return TRUE if the desired record exists, and FALSE otherwise?

PLVgen offers the **`recfnd`** procedure to generate a version of this "record found?" function that will save you many keystrokes if and when you need this functionality. The header for **`recfnd`** is, quite simply:

```
PROCEDURE recfnd (table_in IN VARCHAR2);
```

In other words, you provide the name of the table whose contents you wish to check, and **`recfnd`** does the rest of the work from there.

The following session generates a function that returns TRUE if the employee is found, and FALSE otherwise.

```
SQL> exec PLVgen.recfnd ('emp');
         FUNCTION empexists (empkey_in IN NUMBER) RETURN BOOLEAN
         IS
            CURSOR emp_cur
            IS
               SELECT 1
                 FROM emp
                WHERE
                  AND
                ORDER BY ;
            emp_rec emp_cur%ROWTYPE;
            retval BOOLEAN := FALSE;
         BEGIN
            OPEN emp_cur;
            FETCH emp_cur INTO emp_rec;
            retval := emp_cur%FOUND;
            CLOSE emp_cur;
            RETURN retval;

         EXCEPTION
            WHEN OTHERS
            THEN
               RETURN NULL;
         END empexists;
```

Notice that the procedure creates appropriate names for the function, cursor, and record from the table name -- and places that table name into the cursor's query. It follows the template approach for functions, so that you have an exception handler to return a NULL in case of problems and a single RETURN in the body of the function. Sure, you have to change the SELECT statement and the key might not be a NUMBER. Regardless, with a simple cut and paste operation, you have more than twenty lines of consistent, readable code.

16.2.8 Generating a Cursor FOR Loop

Another common programming construct for PL/SQL developers is the cursor FOR loop, so it will come as no surprise that PLVgen offers a procedure to generate this loop. The header for the **`cfloop`** procedure is:

```
PROCEDURE cfloop (table_in IN VARCHAR2);
```

You provide the name of the cursor and **cfloop** does the rest. The following SQL*Plus session demonstrates the kind of code generated, in this case for a cursor FOR loop based on the **dept** table.

```
SQL> exec PLVgen.cfloop('dept');
   CURSOR dept_cur
   IS
      SELECT *
        FROM dept
       WHERE
         AND
       ORDER BY ;

   << dept_cur >>
   FOR dept_rec IN dept_cur
   LOOP

   END LOOP dept_cur;
```

The **cfloop** procedure does several convenient things for us: it defaults the SELECT list to ***** or all columns. It creates a loop label, which is also applied to the END LOOP statement to improve readability. It *avoids* declaring the record used by the FOR loop −− an unnecessary and potentially hazardous step often taken by poorly informed PL/SQL programmers.

You cut and paste the cursor declaration into the declaration section; move the FOR loop itself into the body of the program; put some executable code in the body of the loop; and off you go.

16.2.9 Generating a Timer Script

The PLVtmr package offers an easy−to−use mechanism for timing PL/SQL code execution down to a hundredth of a second. I found that I often used this package to time the performance of the same line or lines of code executed more than once −− in other words, inside a loop. Rather than write these loops again and again, I added a procedure to PLVgen to generate this code for me.

The **timer** procedure's header is as follows:

```
PROCEDURE timer (plsql_in IN VARCHAR2);
```

You provide the line of PL/SQL code you want to execute and PLVgen provides a SQL*Plus script as shown below:

```
SQL> exec PLVgen.timer ('calc_totals;');
   BEGIN
      PLVtmr.set_factor (&1);
      PLVtmr.capture;
      FOR rep IN 1 .. &1
      LOOP
         calc_totals;
      END LOOP;
      PLVtmr.show_elapsed ('MSG');
   END;
   /
```

These lines of code should be placed in a file with the *.sql* extension. Change the text in the call to **show_elapsed**, add other lines of code to the body of the loop if desired. Upon execution, you provide the number of iterations (or, in PLVtmr terms, the factor), and the script executes (in this case) the **calc_totals** procedure that number of times. **PLVtmr.show_elapsed** then displays the total elapsed time and the per−execution time as well.

PREVIOUS HOME NEXT

16.1 Options for Best Practices | BOOK INDEX | 16.3 Modifying PLVgen Behavior

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

16.3 Modifying PLVgen Behavior

The PLVgen package is a tool to improve the quality of life of developers. It can be used directly in an environment like SQL*Plus. It can also be utilized from within a GUI interface to make it easier to select options and combine different code fragments. Given the wide variety of circumstances under which PLVgen could be used, and the variety of coding styles used by developers, I face a big challenge in implementing PLVgen: make it flexible!

I cannot, for example, force anyone to use my PLVexc package to handle exceptions. An application team might not want to use the PLVtrc package to build in an execution trace. If I don't give users of PLVgen the option to use and ignore these different elements, I will not have very many users of the package. Consequently, I supply a variety of get−and−set programs in PLVgen to toggle various aspects of generate behavior. I even used early versions of PLVgen to generate get−and−set programs for the PLVgen package itself!

PLVgen offers a set of toggles to turn on or off the inclusion of various elements of code. It also provides a set of programs to modify the appearance of output, particularly as regards indentation. These programs are listed in Table 16.1 and are explained in the following sections.

Table 16.1: Programs to Modify PLVgen Behavior and Output

Behavior	Programs	Description
Set author	`set_author author`	Sets and retrieves the current string used as the author in a program header. The default is **`NULL`**.

Set indentation	`set_indent indent`
	`incr_indent`
		Sets and retrieves the two types of indentation: initial and incremental. The default for the indent is 0 and for the incremental is 3.

Use program trace	`usetrc nousetrc`	Inserts calls to **`PLVtrc.startup`** and **`PLVtrc.terminate`** in generated procedures and functions. Default is OFF.
Use exception handling	`useexc nouseexc`	Inserts a WHEN OTHERS exception handler that calls the exception handler **`PLVexc.rec_continue`**, a generic program that records the error and then continues execution. You can then add other handlers as well. Default is OFF.
Use program header	`usehdr nousehdr`	Places a header (within a comment block) for procedures and functions. This header uses the author name set with a call to **`set_author`**. Default is OFF.
Use comments	`usecmnt nousecmnt`	Places comments inside the generated code. Examples of comments include banners for the different components of a package and descriptions of the components of get−and−set routines. Default is ON.

Use online help	`usehlp nousehlp`	Puts stubs for help text in programs and also creates a procedure in your generated package to provide help for that package. Default is ON.
Add CREATE OR REPLACE	`usecor nousecor`	Adds the syntax necessary to CREATE OR REPLACE the generated procedure, function, or package. Default is OFF.
Use line numbers	`useln nouseln`	Displays the line number next to the source code.

16.3.1 Indentation in PLVgen

There are two elements to indentation: the initial indentation and the next or incremental indentation. The initial value is the number of spaces inserted before any line of code is generated. The incremental indentation is the number of spaces indented for each successive indent (code within a loop, declarations within the declaration section, and so on). PLVgen provides a single set program for both of these values:

```
PROCEDURE set_indent
   (indent_in IN NUMBER,
    incr_indent_in IN NUMBER := c_incr_indent);
```

The default starting indentation is 0. The default incremental indentation is 3. Use the **`set_indent`** procedure to change either or both of these values.

PLVgen provides two functions to return the current indentation values. The **`indent`** function returns the current starting indentation value. The **`incr_indent`** function returns the current incremental indentation value. The headers for these functions are shown below:

```
FUNCTION indent RETURN NUMBER;
FUNCTION incr_indent RETURN NUMBER;
```

When might you call **`set_indent`**? If you are going to generate a function to stick inside a package, you want to set the starting indentation at 3 (or whatever your standard is) so that the resulting code is indented properly within the context of the package. This saves you editing time in which you insert spaces at the beginning of each line.

16.3.2 Setting the Code Author

If you are generating a header for your code, you can set the name of the author placed in the header. The default for the author is NULL. You can call the **`set_author`** procedure to change the author. Call the **`author`** function to retrieve the current author name.

The headers for these two functions are shown below:

```
PROCEDURE set_author (author_in IN VARCHAR2);
FUNCTION author RETURN VARCHAR2;
```

The following execution of **`set_author`**, for example, sets the name to "Steven Feuerstein".

```
SQL> exec PLVgen.set_author ('Steven Feuerstein');
```

To make certain that this author value is always set for me when generating code, I include the above command in my **`login.sql`** script. This file is executed automatically on startup of SQL*Plus and initializes my environment.

16.3.3 Using the Program Header

You can decide if you want PLVgen to include a standard header in your generated program units (packages, functions, and procedures). The format for this header is:

```
/*
|| Program:
||  Author:
||    File:
|| Created:
*/
```

The program name is usually taken from other inputs to the generator program. The author string is set through a call to PLV**`gen.author`**. The file name is constructed from the program string. The create date/time stamp is SYSDATE.

You can toggle the header on or off. In addition, you can call a function to determine the current status of the "use program header" toggle. The headers for these three programs are:

```
PROCEDURE usehdr;
PROCEDURE nousehdr;
FUNCTION using_hdr RETURN BOOLEAN;
```

You call **`usehdr`** to turn on use of the header and **`nousehdr`** to turn off the header. Call the **`using_hdr`** function if you want to know whether the header is being used. (This is included mostly as a courtesy and for completeness. Usually you simply turn the feature on or off and be done with it.)

16.3.4 Using the Program Trace

The PLVtrc package offers the ability to maintain a trace of the programs that are currently on the execution stack of PL/SQL. PL/SQL itself provides this information with the DBMS_UTILITY.FORMAT_CALL_STACK function, but that stack does not show the names of programs within a package −− a serious drawback for package−centered PL/SQL code development. So you can call **`PLVtrc.startup`** to indicate that a particular program has started. And you call **`PLVtrc.terminate`** to signal that the program has ended. See Chapter 21, *PLVlog and PLVtrc: Logging and Tracing*, for more information on how to use PLVtrc.

The PLVgen package is "PLVtrc−aware." It automatically inserts calls to the startup and terminate programs of the trace facility if you turn on this feature. You can toggle the trace on or off. In addition, you can call a function to determine the current status of the "use program trace" toggle. The headers for these three programs are:

```
PROCEDURE usetrc;
PROCEDURE nousetrc;
FUNCTION using_trc RETURN BOOLEAN;
```

You call **`usetrc`** to turn on use of the trace and **`nousetrc`** to turn off the trace. Call the **`using_trc`** function if you want to know whether the trace is being used.

When you are using the program trace, a generated procedure has at a minimum the following body or execution section:

```
PROCEDURE calc_totals
IS
BEGIN
   PLVtrc.startup ('calc_totals');
   PLVtrc.terminate;
END calc_totals;
```

Notice that PLVgen automatically inserts the name of the current program unit to **PLVtrc.startup**; you don't have to mess with this. You just insert all of your application−specific code between the startup and terminate lines. By using PLVgen, you can make the use of PLVtrc practical and comprehensive. This is especially worthwhile when you are going to take advantage of the high−level exception handlers provided by PLVexc.

16.3.5 Using the PLVexc Exception Handler

The PLVexc package provides a high−level, declarative approach to exception handling in PL/SQL programs. It is a powerful facility for consistent, high−quality handling of errors.

The PLVgen package is also "PLVexc−aware." It automatically creates an exception section and provides a single WHEN OTHERS handler that calls a PLVexc handler procedure. There are two possible handlers that will be placed in the exception section. If you are using PLVtrc (you have called **PLVgen.usetrc**), then the more abstract **rec_continue** procedure will be used. If you are not using PLVtrc, PLVgen inserts a call into the lower−level **PLVexc.handle** procedure (see Chapter 22, *Exception Handling*, for more information about the differences between these two programs).

You can toggle the inclusion of PLVexc on or off. In addition, you can call a function to determine the current status of the "use program PLVexc" toggle. The headers for these three programs are:

```
PROCEDURE useexc;
PROCEDURE nouseexc;
FUNCTION using_exc RETURN BOOLEAN;
```

You call **useexc** to turn on use of the PLVexc and **nouseexc** to turn off the PLVexc. Call the **using_exc** function if you want to know if PLVexc is being used.

When you are using exception handling and the trace facility (enabled by a call to **PLVgen.usemax** or to both **PLVgen.useexc** and **PLVgen.usetrc**), the exception section for a procedure looks like this:

```
EXCEPTION
   /* Call PLVexc in every handler. */
   WHEN OTHERS
   THEN
      PLVexc.rec_continue;
END calc_totals;
```

Notice the comment line before the exception handler. If you are going to use PLVtrc in conjunction with PLVexc, you must call a PLVexc exception handler procedure in every exception handler section.

If you have turned on exception handling in PLVgen, but have turned off use of PLVtrc, the exception section for your generated programs looks like this:

```
EXCEPTION
   WHEN OTHERS
   THEN
      PLVexc.handle (calc_totals, SQLCODE, PLVexc.rec_continue);
END calc_totals;
```

> ***NOTE:*** If you want to use a different PLVexc handler, such as **rec_halt**, you need to cut and paste and then edit your generated code (you could, alternatively, change the name of the default handler, that is stored in **c_PLVexc_handler** in the PLVgen body).

When you are not using PLVexc−based exception handling (you have called **PLVgen.nouseexc**), the exception section for a procedure looks like this:

```
EXCEPTION
   WHEN OTHERS
   THEN
      NULL;
END calc_totals;
```

So even when you do not take advantage of PL/Vision−based exception handling, PLVgen still generates an exception section in your code. This is an important element of best practices for module construction and should almost never be compromised.

16.3.6 Generating Comments

PLVgen includes a number of different kinds of comments in your generated code. You can toggle the generation of these comments on or off. In addition, you can call a function to determine the current status of the "use comments" toggle. The headers for these three programs are:

```
PROCEDURE usecmnt;
PROCEDURE nousecmnt;
FUNCTION using_cmnt RETURN BOOLEAN;
```

You call **`usecmnt`** to turn on use of the comment lines and **`nousecmnt`** to turn it off. Call the **`using_cmnt`** function if you want to know whether the comments are being used.

> ***NOTE:*** The header and the help text stubs are not affected by the setting of this toggle. They are PL/SQL comments, but they act as specialized text within the PLVgen package and so are treated differently (with their own toggles).

16.3.7 Generating Online Help Text Stubs

You can control the generation of online help text stubs (available through use of the PLVhlp package) with the "use help" toggle. You can toggle the generation of these stubs on or off. In addition, you can call a function to determine the current status of the "use help" toggle. The headers for these three programs are:

```
PROCEDURE usehlp;
PROCEDURE nousehlp;
FUNCTION using_hlp RETURN BOOLEAN;
```

You call **`usehlp`** to turn on use of the comment lines and **`nousehlp`** to turn it off. Call the **`using_hlp`** function if you want to know whether the comments are being used.

If you know that you are not going to use PLVhlp to make online help available to users, you can keep these extraneous comments out of your programs with a call to the **`PLVgen.nousehlp`** procedure. If you generate these stubs into your programs and then never use them, you will have absolutely no adverse impact on your program execution.

16.3.8 Generating Line Numbers

PLVgen lets you generate line numbers for your programs. This may not mean much to any of you, but it sure was important to me. I wanted to be able to point out specific lines of generated code for your attention, so I built this facility into the package itself.

You can toggle the generation of these line numbers on or off. In addition, you can call a function to determine the current status of the "use line numbers" toggle. The headers for these three programs are:

```
PROCEDURE useln;
PROCEDURE nouseln;
FUNCTION using_ln RETURN BOOLEAN;
```

You call **useln** to turn on use of the line numbers and **nouseln** to turn it off. Call the **using_ln** function if you want to know whether the line numbers are being generated.

16.3.9 Including CREATE OR REPLACE

When you are generating code to be compiled and stored in the database through SQL*Plus, you need to use the CREATE OR REPLACE syntax. In all other situations (generating code for inclusion in an Oracle Developer/2000 environment or to be pasted into a GUI development environment like Oracle Procedure Builder), you will *not* use CREATE OR REPLACE. That part of the syntax is done for you.

PLVgen gives you the ability to choose whether you want to generate a program with the CREATE OR REPLACE syntax. The three programs managing this toggle are listed here:

```
PROCEDURE usecor;
PROCEDURE nousecor;
FUNCTION using_cor RETURN BOOLEAN;
```

Call the **usecor** program to request that a CREATE OR REPLACE be put before the program unit name −− and a forward slash be added as the last line of the generated output. Call **nousecor** to ignore this DDL syntax. Finally, call **using_cor** to find out the current status of this toggle.

16.3.10 Setting Groups of Toggles

As you can see, there are many options when it comes to modifying the output and behavior of the PLVgen package. Without this flexibility, I doubt that anyone would ever find a package like PLVgen useful. On the other hand, there may be one thing worse than offering no options: offering too many options.

Suppose, for example, you wanted to turn on all of the different toggles available in PLVgen. You would then have to execute all of the following commands:

```
SQL> exec PLVgen.usehdr
SQL> exec PLVgen.useexc
SQL> exec PLVgen.usehlp
SQL> exec PLVgen.usecor
SQL> exec PLVgen.usetrc
SQL> exec PLVgen.usecmt
SQL> exec PLVgen.useln
```

There are at least two big problems with this: you would have to know about all of these options −− and the set of options will quite likely be expanding in the foreseeable future. You would also have to do an awful lot of typing −− you simply wouldn't bother!

This is not a recipe for usability. So in addition to providing each of the individual toggles, I provide two other "master switches" for setting the toggles:

```
PROCEDURE usemin;
PROCEDURE usemax;
```

The **usemin** program turns off all the toggles; the **usemax** turns them all on. If you want to make sure that you generate line numbers, but otherwise have everything else turned off, you would execute these two commands before generating code:

```
SQL> exec PLVgen.usemin;
SQL> exec PLVgen.useln;
```

As PLVgen is expanded to support new toggles and areas of flexibility, I expect that it will be necessary to create other programs to turn on or off various sets of toggles. These combinations will create "preferences" and further improve the usability of the package.

PREVIOUS | HOME | NEXT
16.2 Code Generated by PLVgen | BOOK INDEX | 16.4 Implementing PLVgen

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

16.4 Implementing PLVgen

The PLVgen package is a large package. The body alone contains more than 20,000 characters spread over 1,390 lines of code. It is not, however, a terribly complicated package. Each of the "public" programs (the code generators) is, at heart, a sequence of commands to output various combinations of text.

The biggest challenge in constructing PLVgen was to minimize the amount of redundant code. There are many common lines of text, for example, between the template for a function and procedure. And my package allows you to generate many different kinds of functions and procedures. If I simply hard−coded the lines of text to be output for each different program unit, the PLVgen package would run into two problems:

1. It would grow to monstrous proportions and eventually (soon?) hit the upper limit for code size in PL/SQL.

2. It would be difficult to maintain and enhance. I would have to constantly cut and paste to build new generator procedures, since there would be little shared code. And what if I discovered a mistake in all of my different function generators? Or, even more likely, what if I want to add another toggle to fine−tune the look−and−feel of the generated code? I would have to make changes across many different program units in the package.

PLVgen is definitely one of those packages where a building−block approach was absolutely critical to a successful implementation. To explain PLVgen, I first review some of the best practices and coding styles I support through the package. Then I show the implementation of the high−level procedure generator (the **`proc`** procedure). Finally, I shift down to the lowest level of the package (the **`put_line`** procedure) and build my way up from there.

16.4.1 PLVgen Coding Guidelines

You will find implemented in PLVgen the following best practices and coding styles:

- Use consistent indentation to reveal the logical flow of the program and delineate the different sections of the block structure.

- Include in all programs an exception section to handle abnormal behaviors. Taking this guideline to the next level of reusability, PLVgen offers programs (implemented through the PLVexc package) to call in the exception handlers.

- Code all reserved words in the PL/SQL language in uppercase. Use lowercase for all application−specific identifiers. Generally, this is accomplished with hard−coded literals and the use of UPPER and LOWER. This guideline presents more of a challenge when applied to complex expressions passed to PLVgen as default values.

- Employ a "template" approach to functions in which there is a single successful RETURN statement that returns a locally declared variable of the same datatype as the function.

Keep these guidelines in mind as you examine the program units of PLVgen.

16.4.2 Implementing the Procedure Generator

This **proc** program generates a procedure as described in Section 16.2.2. Let's now go over the implementation of **proc**, as shown in Example 16.1.

Example 16.1: The Code for the proc Procedure

```
PROCEDURE proc
   (name_in IN VARCHAR2,
    params_in IN VARCHAR2 := NULL,
    exec_in IN VARCHAR2 := NULL,
    incl_exc_in IN BOOLEAN := TRUE,
    indent_in IN INTEGER := 0,
    blank_lines_in IN VARCHAR2 := c_before)
IS
   v_name PLV.plsql_identifier%TYPE := LOWER (name_in);
BEGIN
   initln;
   put_line
      (cor_start || proc_header (v_name, params_in),
       indent_in,
       blank_lines_in);

   put_header_cmnt (v_name, indent_in);

   put_all_help (indent_in);

   put_is_begin (v_name, indent_in);

   IF exec_in IS NOT NULL
   THEN
      put_line
         (RTRIM (exec_in, ';') || ';',
          indent_in + v_incr_indent);
   END IF;

   put_terminate (v_name, indent_in + v_incr_indent);

   IF incl_exc_in
   THEN
      put_when_others (v_name, indent_in + v_incr_indent);
  END IF;

  put_end (v_name, indent_in);
END;
```

The very first thing I do is declare a local variable, **v_name**, and set it to the lower−casing of the specified program name:

```
v_name PLV.plsql_identifier%TYPE := LOWER (name_in);
```

This step enforces the style guide in which application−specific identifiers are entered in lowercase. It also takes advantage of the predefined datatype for PL/SQL identifier variables.

From this point onwards, the body of **`proc`** is composed of calls to a series of highly specialized procedures and functions defined in the body of the package. These programs are:

Procedure **`put_line`** *(line 12)*
: Puts out a line of generated code.

Function **`cor_start`** *(line 13)*
: Returns the CREATE OR REPLACE syntax if **`using_cor`** returns TRUE. Otherwise, returns NULL.

Function **`proc_header`** *(line 13)*
: Returns a string with the header for the specified procedure. The header is all of the definition of the procedure up to the IS keyword.

Procedure **`put_header_cmnt`** *(line 17)*
: Puts a program header (comment block with author, program name, etc.) after the procedure header.

Procedure **`put_all_help`** *(line 19)*
: Puts in all current comment blocks for online help.

Procedure **`put_is_begin`** *(line 21)*
: Puts the IS and BEGIN keywords in the procedure definition. This procedure also adds a call to the **`PLVtrc.startup`** procedure if **`using_trc`** returns TRUE.

Procedure **`put_terminate`** *(line 30)*
: Puts a call to **`PLVtrc.terminate`** at the end of the procedure if **`using_trc`** returns TRUE.

Procedure **`put_when_others`** *(line 34)*
: Puts an exception section with a WHEN OTHERS clause if the **`incl_exc_in`** argument is TRUE.

Procedure **`put_end`** *(line 37)*
: Puts the END statement at the, well, the end of the procedure.

16.4.2.1 Benefits of internal modularization

Once I have created all of these specialized variations on **`put_line`**, the body of the **`proc`** procedure is very short and simple. Why do I go to all this trouble? Why not simply issue calls to **`put_line`** with the appropriate combinations of strings? There are two very good reasons:

1. Each of these are called in other programs in the PLVgen package. If I did not encapsulate the logic inside a variation of **`put_line`**, I would be repeating my logic. This would make debugging and enhancement of the package difficult, if not impossible.

2. The calls to the put procedures now reflect the structure of a PL/SQL procedure.

Let's look at the second reason in detail. First, there is the program header:

```
put_line
   (cor_start || proc_header (v_name, params_in), indent_in);
```

While it is true that I don't have a separate **`put_header`** program, this call to **`put_line`** uses two functions to encapsulate much of the header logic. Most importantly, the **`proc_header`** function returns the header string. This function is called in several different programs in PLVgen.

After the header of the function, I need the IS and BEGIN clauses. These are provided by the **put_is_begin** procedure:

```
put_is_begin (v_name, indent_in);
```

Then it is time for executable statements:

```
IF exec_in IS NOT NULL
THEN
   put_line
      (RTRIM (exec_in, ';') || ';',
       indent_in + v_incr_indent);
END IF;
```

Notice that I make sure there is a single semicolon at the end of the supplied string. That way, the user can leave it off and it won't make any difference (one example of "self−correcting" software; the smarter I make my code, the more likely and widely it is going to be used). I also indent this executable statement by an additional amount to offset it from the BEGIN keyword (a section delimiter).

I then terminate the procedure in three steps. First, I insert a call to **PLVtrc.terminate** if the trace is in use (again, notice the incremental indentation):

```
put_terminate (v_name, indent_in + v_incr_indent);
```

Then I put a WHEN OTHERS exception section (unless told not to):

```
IF incl_exc_in
THEN
   put_when_others (v_name, indent_in + v_incr_indent);
END IF;
```

Finally, it is time to issue the END statement to close the procedure:

```
put_end (v_name, indent_in);
```

By modularizing my code in this way, I am able to avoid superfluous and time−consuming inline documentation. The breakout of the modules explains much of what I am doing in the program. In addition, it is much easier to maintain and enhance this program. If I want to add code to the executable section, I make changes within that IF statement. If I need to enhance the way I terminate my generated program units, I will do so in **put_terminate** and/or **put_end**. The main body of the **proc** procedure can remain as it is.

Now that I have walked you through one of my high−level generators, let's dive down into the lowest level put program: **put_line**.

16.4.3 put_line Procedure

At the very core of the PLVgen package is the **put_line** procedure (see Example 16.2). This program is called by other private put procedures to output various combinations of text. It is also called directly from the high−level, public programs such as **proc** and **toggle**. The **put_line** procedure is the only way to generate code text from the PLVgen package.

Example 16.2: The put_line Procedure

```
PROCEDURE put_line
   (stg_in IN VARCHAR2 := NULL,
    incr_indent_in IN INTEGER := 0,
    blanks_in IN VARCHAR2 := c_none)
IS
```

```
BEGIN
   IF blanks_in IN (c_both, c_before)
   THEN
      PLVio.put_line (stg_with_ln);
   END IF;

   PLVio.put_line
      (stg_with_ln (indent_stg (incr_indent_in) || stg_in));

   IF blanks_in IN (c_both, c_after)
   THEN
      PLVio.put_line (stg_with_ln);
   END IF;
END;
```

Since **`put_line`** is used in so many different ways, it must be flexible and, therefore, it must take several arguments. The header for **`put_line`** is as follows:

```
PROCEDURE put_line
   (stg_in IN VARCHAR2 := NULL,
    incr_indent_in IN INTEGER := 0,
    blanks_in IN VARCHAR2 := c_none)
```

The three arguments are described below:

`stg_in`
: The string to be displayed. The default is NULL, which would produce a blank line.

`incr_indent_in`
: The incremental indentation to be applied to the string. The default is 0, which means that the current level of indentation (set with a call to **`set_indent`**) is employed.

`blanks_in`
: A string value which indicates the type of white space to generate around the line being displayed. The options are: **`c_none`** (no blanks), **`c_before`** (one line before), **`c_after`** (one line after), or **`c_both`** (one line before and after).

16.4.3.1 Flexibility of put_line

The following examples of calls to **`put_line`** give you an idea of its flexibility:

1. Display one blank line.

   ```
   put_line;
   ```

2. Display the word NULL; surrounded by blank lines.

   ```
   put_line ('NULL;', 0, c_both);
   ```

3. Indent a call to **`PLVtrc.startup`** by the standard incremental indentation and display a blank line after that program call.

   ```
   put_line ('PLVtrc.startup', v_incr_indent, c_after);
   ```

Notice that in the second example I had to include a value of 0 for the incremental indentation. After a while, I found this practice annoying. The zero value is really just filler −− a placeholder so I could specify the blank−line behavior without using named notation. To get around this artificial coding, I have also overloaded **put_line** as follows:

```
PROCEDURE put_line
   (stg_in IN VARCHAR2, blanks_in IN VARCHAR2)
IS
BEGIN
   put_line (stg_in, 0, blanks_in);
END;
```

16.4.3.2 Output mechanism of put_line

Did you notice that **put_line** does not call DBMS_OUTPUT.PUT_LINE? It doesn't even call **p.l**, that ubiquitous displayer of output from a PL/SQL program. In fact, PLVgen relies on the PLV**io.put_line** to generate the code. Why did I bother with PLVio? What does the user have to gain from this extra layer of code? A tremendous amount of flexibility, namely with the ability to redirect the output of generated code.

What if I want to plug−and−play my code generator from within a GUI interface (e.g., Oracle Forms or PowerBuilder)? If I rely on DBMS_OUTPUT.PUT_LINE, this GUI tool would have to be able to read information from the DBMS_OUTPUT buffer and then manipulate that data. I don't know about PowerBuilder, but I have not been successful in reading the DBMS_OUTPUT buffer from Oracle Forms (if you want to try, check out the GET_LINE procedure in the DBMS_OUTPUT package).

With the PLVio package, I can redirect my output to a database table, PL/SQL table, or (with Release 2.3) an operating system file. To accomplish this I do not have to change PLVgen. I do not have to change any of the programs in which I have embedded calls to PLVgen elements. All I have to do is "flip a switch" by calling the **settrg** procedure of PLVio. For example, if I want to send my output to a PL/SQL table, I would issue this command before any calls to the PLVgen package:

```
PLVio.settrg (PLV.pstab);
```

Then all calls to **PLVgen.put_line** would add another row to the PL/SQL table defined in PLVio. The GUI environment can then extract the text from these rows and manipulate them within the GUI environment. You would not have to make a single change to the PLVgen package to accomplish this switch!

> ***NOTE:*** If a PLVio target repository has not been selected by the time PLVgen is first called, the initialization section of the PLVgen package body automatically sets the target to "standard out," as shown below:
>
> ```
> PACKAGE BODY PLVgen
> IS
>    /* All the package elements */
> BEGIN
>    /* If the target has not been set, use standard output. */
>    IF PLVio.notrg
>    THEN
>       PLVio.settrg (PLV.stdout);
>    END IF;
> END;
> ```

16.4.3.3 Modularization inside put_line

The **put_line** procedure calls PLV**io.put_line** to send the line of generated code to the designated repository. What text is sent to **PLVio.put_line**? The **PLVgen.put_line** procedure actually makes use of two other private functions, **stg_with_ln** and **indent_stg**, to construct the string for output.

The **stg_with_ln** function incorporates logic required to display a line number before the line of code. It hides the logic and private variables shown below:

```
FUNCTION stg_with_ln (stg_in IN VARCHAR2 := NULL)
RETURN VARCHAR2
IS
BEGIN
   IF usingln
   THEN
      v_currln := v_currln + 1;
      RETURN
         (LPAD (TO_CHAR (v_currln), 5) || ' ' || stg_in);
   ELSE
      RETURN stg_in;
   END IF;
END;
```

The **indent_stg** function encapsulates the logic required to properly indent the line of code according to the current indent setting (initial and incremental). The body of **indent_stg** is shown below:

```
FUNCTION indent_stg (incr_indent_in IN INTEGER := 0)
   RETURN VARCHAR2
IS
BEGIN
   RETURN (RPAD (' ', v_indent + incr_indent_in));
END;
```

The **put_line** program is the only module in PLVgen that calls either **stg_with_ln** or **indent_string**. One could, therefore, argue that these modules represent an unnecessary layer of code. That may be the case for **indent_stg**. When I first wrote **put_line**, however, I did not realize that I would be so successful at funneling all output through **put_line**. I only knew that I did not want to embed the RPADding logic required for indentation right into **put_line**. It felt safer to me to hide that implementational detail behind a function. I applied the same reasoning to handling line numbers −− and that paid off immediately.

The **put_line** procedure of PLVgen does, in fact, call **stg_with_ln** three times in its short body. Since a blank line should have a line number as well, I needed to apply that logic in all three calls to PLV**io.put_line**. If I did not consolidate line number handling inside a separate function, I would have repeated the formula and code in **put_line**.

This instinct to hide code behind a procedure or function is one you should develop and then cultivate actively. You may in some cases end up writing a program or two that is only used once. The vast majority of your modules will, however, be reused and reused often. If you are sufficiently fanatical about modularization you eventually reach a critical mass of code: a strong development foundation that enables you to implement complex programs quickly and with few bugs.

16.4.3.4 Building on put_line

Once the **put_line** procedure was in place, I could create many other, more specialized put programs to handle different aspects of PL/SQL code. You have already seen this specialization in the **proc** procedure with such programs as **put_terminate**. There are, in fact, ten different put programs:

```
put_all_help
put_begin
put_comment
put_cor_end
put_end
put_header_cmnt
put_help
put_is_begin
put_terminate
put_when_others
```

Some of the programs, such as **`put_header_cmt`** shown below, make direct calls to **`put_line`**:

```
PROCEDURE put_header_cmnt
   (name_in IN VARCHAR2,
    indent_in IN INTEGER := 0,
    file_in IN VARCHAR2 := NULL,
    author_in IN VARCHAR2 := v_author)
IS
BEGIN
   IF using_hdr
   THEN
      put_line ('/*', indent_in);
      put_line ('|| Program: ' || name_in, indent_in);
      put_line ('||  Author: ' || author_in, indent_in);
      put_line ('||    File: ' || file_in, indent_in);
      put_line ('|| Created: ' || PLV.now, indent_in);
      put_line ('*/');
   END IF;
END;
```

Other programs call both **`put_line`** and other put procedures, such as **`put_end`**:

```
PROCEDURE put_end
   (prog_in IN VARCHAR2,
    indent_in IN INTEGER := 0,

    incl_term_in IN BOOLEAN := FALSE)
IS
BEGIN
   IF incl_term_in
   THEN
      put_terminate (prog_in, indent_in+v_incr_indent);
   END IF;
   put_line ('END ' || prog_in || ';', indent_in);
   put_cor_end;
END;
```

The conclusion to be drawn from all these layers of code is that you should always take fullest possible advantage of the opportunity to modularize. No user of PLVgen will ever know about all those different, private modules. But when it is time to add another code generator or enhance an existing program, those internal modules make it much easier to implement the changes.

Another moment when all of this internal modularization will come in handy is when I enhance PLVgen so that it can be used by other developers to build their own customized code generators. To accomplish this, I need to publicize many of my private modules by putting their headers in the PLVgen specification. Then you can more easily construct code generators that reflect your own coding standards and specific program units −− without modifying the base PLVgen package itself.

16.4.4 Overloading in PLVgen

PLVgen takes advantage of the overloading feature of packages in a very interesting way to accomplish two important objectives:

1. Make it as easy as possible to generate code, particularly functions.
2. Minimize the volume of code required to implement a wide array of generators.

Let's take a close look at the overloading in PLVgen. As we've mentioned before, when you overload you define more than one program with the same name. These programs differ in other ways (usually the number and types of parameters) so that at runtime the PL/SQL engine can figure out which of the programs to execute.

The **func** procedure, which generates functions, is overloaded seven times in PLVgen. The **gas** procedures, which generate get−and−set programs for a variable, is also overloaded seven times and in the same way. The techniques I employ in PLVgen to accomplish the overloading are quite interesting and informative. Let's examine the overloading in more detail for the **func** procedure and draw out some lessons.

First, let's take a look at the outcome of my overloaded **func** procedure. The following execution of **func** generates a numeric function called **totals** whose return value defaults to NULL.

```
SQL> exec PLVgen.func ('totals', 1);
   FUNCTION totals RETURN NUMBER
   IS
      retval NUMBER := NULL;
   BEGIN
      RETURN retval;

   EXCEPTION
      WHEN OTHERS
      THEN
         RETURN NULL;
   END totals;
```

How do I know that a numeric function will be generated? More importantly, how does PLVgen know that it should create a function with a RETURN clause datatype of NUMBER −− and declare the **retval** variable to be of type NUMBER as well? It's got to be the overloading! The second argument passed to **func** was the value 1. Notice that this value does not appear anywhere in the generated function. It was simply used to direct the PL/SQL runtime engine to execute the appropriate **func** generator.

I could have entered any of the following calls to **PLVgen.func** and generated the very same function:

```
SQL> exec PLVgen.func ('totals', −16007.459);
SQL> exec PLVgen.func ('totals', INSTR ('abc', 'Q'));
```

In both of these cases, the second argument evaluates to a number. As a result, the following version of the **func** procedure would be executed:

```
PROCEDURE func
   (name_in IN VARCHAR2,
    datadesc_in NUMBER,
    defval_in IN NUMBER := NULL,
    incl_exc_in IN BOOLEAN := TRUE);
```

Notice that the second argument has a datatype of NUMBER.

16.4.4.1 Implementing the func procedures

Examine the set of four overloaded definitions of **func** in Example 16.3. The version shown above is the only one which has a string as the first argument and a number as the second argument. As a result, the PL/SQL engine executes the code for *that* procedure, which follows:

```
PROCEDURE func
   (name_in IN VARCHAR2,
    datadesc_in NUMBER,
    defval_in IN NUMBER,
    incl_exc_in IN BOOLEAN := TRUE)
```

```
IS
BEGIN
   ifunc
      (name_in,
       c_number,
       NVL (TO_CHAR (defval_in), 'NULL'),
       NULL,
       incl_exc_in);
END;
```

Lo and behold, the entire body of the **`func`** procedure is nothing more than a call to **`ifunc`**, which is the internal version of the **`func`** procedure. In fact, every single one of the other seven overloaded versions of **`func`** also does nothing more than call **`ifunc`**. Here, for example, is the body of the version of **`func`** used to generate a date function:

```
PROCEDURE func
   (name_in IN VARCHAR2,
    datadesc_in DATE,
    defval_in IN DATE,
    incl_exc_in IN BOOLEAN := TRUE)
IS
BEGIN
   ifunc
      (name_in,
       c_date,
       NVL (TO_CHAR (defval_in), 'NULL'),
       NULL,
       incl_exc_in);
END;
```

There is, in fact, only one difference between the bodies of these procedures.

Example 16.3: First Set of Overloaded Definitions for func

```
PROCEDURE func
   (name_in IN VARCHAR2,
    datadesc_in VARCHAR2,
    defval_in IN VARCHAR2 := NULL,
    length_in IN INTEGER := c_def_length,
    incl_exc_in IN BOOLEAN := TRUE);

PROCEDURE func
   (name_in IN VARCHAR2,
    datadesc_in NUMBER,
    defval_in IN NUMBER := NULL,
    incl_exc_in IN BOOLEAN := TRUE);

PROCEDURE func
   (name_in IN VARCHAR2,
    datadesc_in BOOLEAN,
    defval_in IN BOOLEAN := NULL,
    incl_exc_in IN BOOLEAN := TRUE);

PROCEDURE func
   (name_in IN VARCHAR2,
    datadesc_in DATE,
    defval_in IN DATE := NULL,
    incl_exc_in IN BOOLEAN := TRUE);
```

16.4.4.2 Translating data to constant

In the number version, the second argument passed to **`ifunc`** is a constant: **`c_number`**. In the date version, I pass **`c_date`** in the second position. What I have done is convert the datatype of the second argument in

func (the **datadesc_in** parameter) into a string that indicates the type of function to generate. In this way I am able to implement all of the different function generators with a single procedure (**ifunc**), greatly reducing the size of PLVgen and making it easy for me to maintain and enhance all of the function generators at once.

Here are the definitions of the datatype constants:

```
c_varchar2 CONSTANT VARCHAR2(8) := 'VARCHAR2';
c_date CONSTANT VARCHAR2(8) := 'DATE';
c_boolean CONSTANT VARCHAR2(8) := 'BOOLEAN';
c_number CONSTANT VARCHAR2(8) := 'NUMBER';
```

Notice that another aspect of calling **ifunc** is that I convert the default value into a string. Furthermore, if the default value is NULL, I pass a string NULL. Again, this conversion process allows me to implement all of the function generators with a single procedure that has the following header:

```
PROCEDURE ifunc
   (name_in IN VARCHAR2,
    datadesc_in VARCHAR2,
    defval_in IN VARCHAR2,
    length_in IN INTEGER,
    incl_exc_in IN BOOLEAN := TRUE)
```

The arguments of **ifunc** are the same as those for **func**, with the following differences:

1. The **datadesc_in** and **defval_in** arguments are always and only VARCHAR2. All datatype differences have at this point been converted to constants and merged in the bodies of the **func** procedures.

2. The fourth argument, **length_in**, is required only for VARCHAR2 functions so it didn't appear in the headers for number and date versions of **func** (it is present, on the other hand, in the string version −− see Example 16.3). The calls to **ifunc** in those procedures simply pass NULL for the length argument.

The body of **ifunc** closely parallels that of the **proc** procedure. Differences reflect the special structure of a function: the RETURN clause and RETURN statements, the declaration of a local "return value" variable (necessary to conform to my coding standards). I will not go over this implementation here, since the focus is on overloading. I direct your attention, however, to the way that the IS and BEGIN keyword are put separately in a function, since I declare a local variable in between, using the **var_declare** private function.

16.4.4.3 Overloading for unevaluated default values

So far I have examined overloadings of **func** for each of VARCHAR2, NUMBER, DATE, and BOOLEAN datatypes in which the default value passed in is of the same datatype as the function. These versions of **func** allow me to specify a default value which is evaluated and then placed in the local variable declaration.

Suppose I want to create a date function that contains as a default value the first day of 1996. I would call **func** as follows:

```
SQL> exec PLVgen.func ('day_offset', SYSDATE, TRUNC (SYSDATE, 'YYYY'));
   FUNCTION day_offset RETURN DATE
   IS
      retval DATE := '01−JAN−96';
   BEGIN
      RETURN retval;
   EXCEPTION
      WHEN OTHERS
      THEN
         RETURN NULL;
   END day_offset;
```

In this situation, the expression TRUNC (SYSDATE, 'YYYY') was evaluated by **ifunc**. The resulting value was then placed after the assignment operator in the declaration. That works just fine. What if, on the other hand, I don't want the default value to be the first day of 1996? What if, instead, I want the default to be the first day of the *current* year −− whatever that might be? I wouldn't want the default value evaluated. Rather, it should be treated as a literal −− a string, in fact −− and passed on to the assignment without parsing and evaluation.

In this scenario, my call to **func** would look like this:

```
SQL> exec PLVgen.func
             ('day_offset', SYSDATE, 'TRUNC (SYSDATE, ''YYYY'')');
```

and the resulting declaration of the return value variable in the generated function would look like:

```
IS
   retval DATE := TRUNC (SYSDATE, 'YYYY');
BEGIN
```

This kind of default value is surely going to be a common occurrence when generating code in the real world. So if PLVgen is going to be truly useful, it needs to be able to handle this variation.

16.4.4.4 The final overloading frontier

Fortunately, the flexibility provided by overloading lets me get the job done in a straightforward manner. Did you notice that the last call to **func** (passing the string version of TRUNC (SYSDATE, 'YYYY')) contains a sequence of arguments not supported by the overloadings of **func** shown in Example 16.3 (I pass string−date−string instead of string−date−date). To handle this combination, I need to create another overloading of **func**, one that accepts a string default value. This version of **func** is shown below:

```
PROCEDURE func
   (name_in IN VARCHAR2,
    datadesc_in DATE,
    defval_in IN VARCHAR2,
    incl_exc_in IN BOOLEAN := TRUE);
```

Notice that in this case I do not provide a default value of NULL for the **defval_in** parameter. If I did so, then I would have an ambiguous overloading. A call to **func** that only supplied the first two parameter values would be syntactically valid, but would generate the following runtime error:

```
PLS−00307: too many declarations of 'FUNC' match this call
```

The PL/SQL engine would not know which of the two versions of **func** to execute (in both cases only the first arguments are required and they are string−date in both versions). By leaving off a default for the **defval_in** parameter, I force a user to provide three values, the third of which is a string, thereby ensuring that any valid execution of **func** identifies uniquely one of the **func** overloadings.

Example 16.4 shows the additional overloadings for DATE, NUMBER, and BOOLEAN. Together with the versions shown in Example 16.3, I have now presented and explain the full set of overloadings (seven) for the **func** procedure. The same number of and rationale for overloadings is, by the way, applicable to the **PLVgen.gas** procedures.

Example 16.4: Last Three Overloadings of Func

```
PROCEDURE func
   (name_in IN VARCHAR2,
    datadesc_in NUMBER,
    defval_in IN VARCHAR2,
    incl_exc_in IN BOOLEAN := TRUE);

PROCEDURE func
   (name_in IN VARCHAR2,
    datadesc_in BOOLEAN,
    defval_in IN VARCHAR2,
    incl_exc_in IN BOOLEAN := TRUE);

PROCEDURE func
   (name_in IN VARCHAR2,
    datadesc_in DATE,
    defval_in IN VARCHAR2,
    incl_exc_in IN BOOLEAN := TRUE);
```

16.4.4.5 Special casing unevaluated string values

I have presented all of the overloadings, but have not yet finished explaining the full range of functionality available with the **func** procedures. Have you wondered why I have only (!) seven overloadings of **func**? Why don't I offer another overloading to support nonevaluated default values for string functions, for a total of eight overloadings? That would provide a symmetry one might expect in the PLVgen code.

Contrary to first impressions, however, I cannot overload two different versions of the VARCHAR2 function generator as I did for the other datatypes. Recall that the third argument of the original VARCHAR2 func is already a string. An overloading that followed the same approach would simply be a duplicate. You can see my dilemma in the two calls to **func** shown below:

```
SQL> exec PLVgen.func ('full_name', 'A', 'SMITH, SALLY');
SQL> exec PLVgen.func ('full_name', 'A', 'LPAD (last_name_in)');
```

These two uses of **func** look very different to you and me; it is easy to see how and why they should be treated differently in the generated code. To the PL/SQL engine, however, there is no distinction. As a result, I needed to come up with a way to tell my package when it had an expression that should not be evaluated.

The approach I took was to set the following rule: if you want the default value to be passed untouched to the generated function, prefix your default value with a **=**. With this convention, I would change the last example of a call to **PLVgen.func** to:

```
SQL> exec PLVgen.func ('full_name', 'A', '=LPAD (last_name_in)');
```

This special case is recognized and handled in the **var_declare** function. This function is called within **ifunc** to define a local variable to RETURN from the function, as shown below:

```
put_line
   ('v_' ||
    var_declare (v_name, datadesc_in, defval_in, length_in));
```

Inside **var_declare**, the following IF statement is then executed:

```
IF SUBSTR (defval_in, 1, 1) = c_literal AND
   LENGTH (defval_in) > 1
THEN
   v_defval := SUBSTR (defval_in, 2);
ELSIF datadesc_in IN (c_varchar2, c_date)
THEN
```

```
      v_defval := PLVchr.quoted1 (defval_in);
   END IF;
```

Translation: if the first character is an equal sign and there is more to the default than simply an equal sign, set the default value to the expression following the equal sign. Otherwise, if the datatype of the variable is a string or a date, embed the default value in single quotes. The default value is then concatenated *into* the variable declaration statement.

The multiple versions of **`func`** and **`gas`** in PLVgen offer several interesting lessons in package−based overloading. First and most importantly, overloading provides a smooth and easy to use interface. The user of PLVgen only has to remember **`func`** in order to generate a function, regardless of the datatype (within the range of supported datatypes, of course). This is much simpler than remembering different names, such as **`string_func`** and **`date_func`**.

Second, from the implementational view, PLVgen shows how to merge all those different public **`func`** procedures into a single, internal **`ifunc`** procedure. By converting user−entered values to constants that are recognized by the package, I can keep the code required to implement all these variations down to an absolute minimum.

Finally, the steps taken to allow for nonevaluated defaults for the VARCHAR2 function illustrate the kind of creative thinking (or is it just a workaround kludge?) in which you must sometimes engage in order to surmount obstacles in PL/SQL development.

16.4.5 Applying the Toggles

With all of these toggles modifying the look−and−feel of the generated code, it is extremely important to find a way to apply the toggles without cluttering up the code. I accomplish this mostly through the use of those same specialized put programs discussed earlier.

Consider the **`put_comment`** program. This procedure accepts as input a string, any incremental indentation, and also a specifier for surrounding blank lines (the same three arguments as **`put_line`** itself). **`put_comment`** simply surrounds the string with the comment markers, **`/*`** and **`*/`**, and then passes this commented string to **`put_line`**.

The following statement shows an example of a call to **`put_comment`** that outputs the string **`/* Public Modules */`** indented three spaces past the default with a blank line both before and after the comment.

```
   put_comment ('Public Modules', 3, c_both);
```

Yet if the user has executed either of the following lines:

```
   SQL> exec PLVgen.usemin
   SQL> exec PLVgen.nousecmnt
```

then I do not want **`put_comment`** to display anything. There are two different solutions to this situation:

1. Nest every call to **`put_comment`** inside an IF statement like this:

   ```
   IF using_cmnt
   THEN
      put_comment ('Public Modules', 3, c_both);
   END IF;
   ```

2. Put the IF statement inside the **`put_comment`** procedure. This approach is shown below:

```
PROCEDURE put_comment
   (stg_in IN VARCHAR2 := NULL,
    incr_indent_in IN INTEGER := 0,
    blanks_in IN VARCHAR2 := c_none)
IS
BEGIN
   IF using_cmnt
   THEN
      put_line (comment (stg_in), incr_indent_in, blanks_in);
   ELSIF blanks_in != c_none
   THEN
      put_line (NULL, incr_indent_in, c_none);
   END IF;
END;
```

I recommend the second approach. By hiding the IF statement inside **put_comment**, the code in each program that calls **put_comment** is tighter and cleaner. In addition, I do not have to remember the name of the function that tells me whether or not to use comments each time I call **put_comment**. Instead, I code it once inside **put_comment** and make the toggle transparent in my code.

You see this same approach used throughout PLVgen. Again and again you find that each toggle is checked and applied as close as possible to the **put_line** the toggle is supposed to affect.

16.4.6 Leveraging PL/Vision in PLVgen

You may have noticed several references in PLVgen programs to modules from other PL/Vision packages, such as PLVio. In fact, PLVgen takes advantage of the following programs in the PL/Vision library:

PLVio.put_line
Called by **PLVgen.put_line**, this program sends a line of output to the current default target for the PLVio package. If this target has not been set before using this package, the initialization section of the package sets the target to be standard output (display to terminal).

PLV.now
Returns the current date and time as a formatted string.

PLVhlp.comment_start
Returns a string that conforms to the guidelines used by PLVhlp to mark the beginning of a block of help text.

PLVhlp.comment_end
Returns a string that conforms to the guidelines used by PLVhlp to mark the ending of a block of help text.

PLVcase.string
Applies the UPPER−lower method to the parameter list of a program, and to the default value of a variable

PLVchr.quoted1
Embeds the specified string inside single quotes.

PLV.boolstg
Returns the string TRUE if a Boolean evaluates to TRUE. Otherwise returns the string FALSE.

PLVhlp.show

Shows any help text defined in the PLVgen package to help developers understand and use this package.

`PLVio.settrg`
: Sets the target for calls to **`PLVio.put_line`** to standard output if that target has not already been set in the current session.

In most of the uses listed above, the PL/Vision modules play modest roles. They mostly serve to encapsulate logic which, while uncomplicated, should not have to be known outside of the package. Two of the programs, **`PLVio.put_line`** and **`PLVcase.string`**, offer major added−value to the PLVgen package. I have already examined how **`PLVio.put_line`** is used in the **`Plvgen.put_line`** procedure to enhance the flexibility of the code generator to write code out to different repositories. The usage of **`PLVcase.string`** increases the elegance of the code generator and its ability to support best practices, in this case the automatic upper−casing of reserved words.

PREVIOUS | HOME | NEXT
16.3 Modifying PLVgen Behavior | BOOK INDEX | 17. PLVhlp: Online Help for PL/SQL Programs

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

17. PLVhlp: Online Help for PL/SQL Programs

Contents:

This chapter explores the PLVhlp (PL/Vision HeLP) package. You can use PLVhlp to provide online help for your own PL/SQL programs. Just imagine: If you or someone else in your development team forgets how to call a program or can't remember the name of the function to calculate the total something or other in the **`sales_analysis`** package, you can simply type:

```
SQL> exec sales_analysis.help
```

and find text scrolling across the screen, one page at a time, giving you just the information you need. Why, you might almost be able to actually share PL/SQL code across the enterprise in a practical way!

In the following sections, you will learn how to use PLVhlp, both in terms of creating text that can be made available for online help, and in terms of retrieving and displaying the online help text. I then take you step by step through the thought process and implementation behind PLVhlp.

17.1 Who Needs Online Help?

PL/Vision contains many packages, with many programs in each package. Despite its complexity, since I am the author of PL/Vision I should be intimately familiar with all aspects of the product. So you can imagine my frustration when I can't remember precisely how to call one of my own packaged programs, and have to go scrounging through the source code to figure it out. I fume to myself: "You *wrote* this stuff! Can't you remember what it does and how it does it?" But the sad fact is that I cannot remember all the finer nuances of my finer creations. As a result, valuable moments are lost from my life.

What I really need is online help so that when I am confused, I can type in a short, easy−to−remember command (along the lines of HELP!) and get the information I need.

To give you an understanding of my frustration, I offer the following transcript of a session with SQL*Plus (words in italics reflect the inner thoughts of yours truly trying to get something done).

*I fire up SQL*Plus. On the one hand, I sigh at having to deal with this command−line, non−GUI interface. On the other hand, I know that this is the fastest, leanest environment in which to compile and run PL/SQL programs. Here I go...*

```
SQL>
```

All right, now I want to parse a string into separate tokens. Let's see, that's gotta be the PLVlex package and the **`getnext`** *function.*

```
SQL> VARIABLE toke VARCHAR2(30);
SQL> exec :toke := PLVlex.getnext ('this is it');
begin :toke := PLVlex.getnext ('this is it'); end;
 *
ERROR at line 1:
ORA−06550: line 1, column 23:
PLS−00302: component 'GETNEXT' must be declared
ORA−06550: line 1, column 7:
```

```
PL/SQL: Statement ignored
```

Well, that didn't work. I guess it's not called **`getnext`** *after all. Well, what the heck is it called? Oh yeah! It's* **`get_next_token`**.

```
SQL> exec :toke := PLVlex.get_next_token ('this is it');
begin :toke := PLVlex.get_next_token ('this is it'); end;
 *
ERROR at line 1:
ORA−06550: line 1, column 23:
PLS−00302: component 'GET_NEXT_TOKEN' must be declared
ORA−06550: line 1, column 7:
PL/SQL: Statement ignored
```

#$%!! No, that's not correct either! Now I am really frustrated. Sigh... guess it is time to go into Codewright, open up the source code and take a look around... well, I'm lucky to at least have ready access to my code...

Have you ever had this kind of problem? The database is jam−packed with all sorts of goodies, but how are you supposed to know what's there and what it's good for? Stored code is wonderful, but there is no easy way at the moment to view that stored source code −− a necessary step to figure out how to call a particular program or understand what happens when you do run the program. This is most crucial when working with packages. In this case, the package specification is supposed to contain all the information you will ever need to use that package, but how do you find and view that specification?

PREVIOUS | HOME | NEXT
16.4 Implementing PLVgen | BOOK INDEX | 17.2 Current Sources of Information

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

17.2 Current Sources of Information

OK, you've got the DESCRIBE command in SQL*Plus. This command displays the parameters of a stored program (either standalone or package−based). For example, if I want to know about the call interface to my **`PLVvu.code`** procedure, this is what I do:

```
SQL> desc PLVvu.code
PROCEDURE PLVvu.code
 Argument Name    Type            In/Out Default?
 ---------------- --------------- ------ --------
 NAME_IN          VARCHAR2        IN     DEFAULT
 START_IN         NUMBER(38)      IN     DEFAULT
 END_IN           NUMBER(38)      IN     DEFAULT
 HEADER_IN        VARCHAR2        IN     DEFAULT
 TYPE_IN          VARCHAR2        IN     DEFAULT
```

The DESCRIBE command even tells me the return datatype of a function:

```
SQL> desc PLVtkn.is_keyword
FUNCTION PLVtkn.is_keyword RETURNS BOOLEAN
 Argument Name    Type            In/Out Default?
 ---------------- --------------- ------ --------
 TOKEN_IN         VARCHAR         IN
 TYPE_IN          VARCHAR2        IN     DEFAULT
```

The DESCRIBE command, by the way, makes use of the DESCRIBE_PROCEDURE procedure of the builtin DBMS_DESCRIBE package. This program returns a program's arguments in a series of PL/SQL tables. It's a great utility, except it doesn't tell me anything about what the program does −− only how to call it. Furthermore, you need to know the name of the program to use DESCRIBE. The DESCRIBE command does not, in other words, offer online documentation or help for this code.

Another option is to view the source code (assuming that you have access to it, which is far from certain). The PLVvu package provides the **`code`** and **`code_after`** procedures for just this purpose. For example, I can view the first ten lines of the PLVvu package itself as follows:

```
SQL> exec PLVvu.code ('b:PLVvu', 1, 10);
-----------------------------------------------------------------------
Code for PACKAGE BODY PLVVU
-----------------------------------------------------------------------
Line#  Source
-----------------------------------------------------------------------
    1 PACKAGE BODY PLVvu
    2 IS
    3    c_product_header VARCHAR2(30) := 'PL/Vision';
    4    c_linelen INTEGER := 77;
    5    v_last_border BOOLEAN := FALSE;
    6    v_overlap INTEGER := c_overlap;
    7    /*--------------- Private Modules -----------------*/
    8    PROCEDURE disp_border
    9       (line_in IN INTEGER := NULL,
   10        err_line_in IN INTEGER := NULL,
```

Unfortunately, both DESCRIBE and the **`PLVvu.code`** approach are still fairly crude and heavy−handed solutions to a basic problem in PL/SQL: the lack of an online help facility for one's code.

In the remainder of this chapter I present an architecture (and, of course, a PL/SQL package) for implementing online help for PL/SQL programs. The resulting PLVhlp package may not the most elegant approach one would want to take, but it has lots of potential for improving the lot of developers.

PREVIOUS	HOME	NEXT
17.1 Who Needs Online Help?	BOOK INDEX	17.3 What Is "Online Help" for Stored Code?

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8*i* Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

17.3 What Is "Online Help" for Stored Code?

First of all, let me make a distinction and set some scope for this chapter: I am not talking about online help for PL/SQL itself (such as "what is the syntax of the OPEN statement?"). Instead, I want to address how to provide online help for PL/SQL programs that you or I might write and then make available to others. So when I say "online help for PL/SQL," I mean online instructions on how to use custom−built and prebuilt PL/SQL programs that have been stored in a database and on which you have execute authority.[1]

> [1] You could, by the way, also use this package and techniques to provide online help for builtins of the PL/SQL language.

Let's now run through the same scenario I presented at the beginning of the chapter, this time with online help for PL/SQL available.

All right, now I want to parse a string into separate tokens. Let's see, that's gotta be the PLVlex package and the **`getnext`** *function.*

```
SQL> VARIABLE toke VARCHAR2(30);
SQL> exec :toke := PLVlex.getnext ('this is it');
begin :toke := PLVlex.getnext ('this is it'); end;
  *
ERROR at line 1:
ORA−06550: line 1, column 23:
PLS−00302: component 'GETNEXT' must be declared
ORA−06550: line 1, column 7:
PL/SQL: Statement ignored
```

Well, that didn't work. Admit it, Steven, you forgot how to use your own code. Time to get some help.

```
SQL> exec PLVlex.help
Help for PLVLEX
Overview: provides lexical analysis and functions for strings.
==================================================================
Element − Description
==================================================================
FUNCTION is_delimiter − Returns TRUE if string is a delimiter.
PROCEDURE get_next_token − Returns next token in string.
PROCEDURE get_next_atomic − Returns next atomic in string.
```

Now I know what's in the package. What I'd really like are examples of the ways these programs can be used.

```
SQL> @examples PLVlex
Examples for PLVLEX

Here is a call to get_next_token as used by PLVcase,
which UPPER−lower cases PL/SQL programs:

   LOOP
      PLVlex.get_next_token
        (text, curr_pos, token, next_pos, FALSE, text_len, TRUE);
      EXIT WHEN v_token IS NULL OR line_in.pos > line_in.len;

   END LOOP;

The FALSE indicates that I do not want to skip over spaces. This
maintains the program's indentation. The TRUE indicates that I want
qualified names (X.Y.Z) to be returned as one token.
```

I am now ready to roll. That is just the information I needed to move into high gear!

That is the kind of help I would love to be able to get from a PL/SQL environment. Of course, the help itself is only as good as the text you enter. In addition, you need to at least know the name of the package you want to be using... well, I take that back. After all, once the help mechanism is in place, you could provide a very high−level "table of contents" or index into your programs, as I show below.

```
SQL> @help
Overview of PL/Vision

PL/Vision is a development accelerator for PL/SQL programmers.
It is made up of a set of packages which you can use as
plug−and−play components in your own applications. Here is a
quick overview of some of the available packages:

PLVdyn − performs dynamic SQL and PL/SQL operations.
PLVexc − High−level exception handling capabilities.
PLVlex − Lexical analysis and parsing of strings.
PLVlog − Generic log mechanism.
PLVvu − View stored code and compile errors.
```

At this point I imagine you understand the idea −− and I hope you like it. If not, skip the rest of this chapter. If so, I first provide the basic information ("user's guide") you need to use the PLVhlp package. After that I move on to the underlying principles, architecture, and implementation of online help for your PL/SQL programs so that you can get a better understanding of how I went about constructing this utility.

PREVIOUS | HOME | NEXT
17.2 Current Sources of Information | BOOK INDEX | 17.4 Using PLVhlp

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8*i* Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

17.4 Using PLVhlp

There are two aspects to using the PLVhlp package:

1. Display help text using the **show** and **more** procedures. In the PLVhlp architecture, help text is stored with the program unit it documents in specially formatted comment areas. The **show** and **more** procedures then access that text. You can also fine−tune the display of help text by setting the size of a page of text.

2. Create help text to be inserted into stored code for use as online help text. For more details on how to set up this help text, see the sections "Creating Help Text" and "Implementing PLVhlp."

These aspects are covered in the following sections.

17.4.1 Showing the Help Text

The show program initiates the display of help text on a topic. Its specification is as follows:

```
PROCEDURE show (context_in IN VARCHAR2, part_in IN VARCHAR2 := c_main);
```

where **context_in** is the context or program unit that contains the text and **part_in** is the specific topic to be displayed in response to this request. The following call to **show**, for example, requests that the help text in the PLVprs package specification covering the topic "examples" be displayed.

```
SQL> exec PLVhlp.show ('s:PLVprs', 'examples');
```

PLVhlp predefines two constants for commonly used help text topics: top−level or main help and examples help. The last call to **PLVhlp.show** could be rewritten using one of those constants as follows:

```
SQL> exec PLVhlp.show ('s:PLVprs', PLVhlp.c_examples);
```

You might even consider adding constants to the PLVhlp package to make it easier to create and access consistently named areas of help text.

The more procedure's specification is even simpler than that of show:

```
PROCEDURE more;
```

This procedure simply requests that the next page of text for the current help topic be displayed. If there is no more help text or if you have not previously called **PLVhlp.show**, **more** displays the following message:

```
SQL> exec PLVhlp.more
No more help available...
```

PL/Vision also offers some SQL*Plus scripts to make it easier to request help. These scripts are **`help.sql`** and **`more.sql`**. The following lines compare the different approaches using PL/SQL program execution and SQL*Plus script execution.

Instead of typing:

```
SQL> exec PLV.help
```

you can simply enter:

```
SQL> @help plv
```

And instead of typing:

```
SQL> exec more
```

you can simply enter:

```
SQL> @more
```

17.4.2 Setting the Page Size

One of the more interesting aspects of PLVhlp is that it allows developers to see limited amounts of text at a time (a single page). This makes the reading of help text much less frantic; users do not have to feel panicky about pages of text scrolling past their eyes.

The default page size is 25 lines, so that a page of text fits comfortably on a monitor. You can change this value by calling the **`set_pagesize`** procedure, whose header is shown below:

```
PROCEDURE set_pagesize (pagesize_in IN NUMBER);
```

In the following SQL*Plus session, for example, I move the pagesize up to 60, since SQL*Plus in Windows supports vertical scrolling:

```
SQL> exec PLVhlp.set_pagesize(60);
```

You can also find out the current size of the page by calling the **`pagesize`** function. The following little utility (not a part of PLVhlp) makes sure, for instance, that the **`pagesize`** is no larger than the default:

```
PROCEDURE limit_ps (size_in IN INTEGER := 25)
IS
BEGIN
   IF PLVhlp.pagesize > size_in
   THEN
      PLVhlp.set_pagesize (size_in);
   END IF;
END;
```

The **`set_pagesize`** program is available to end users of PLVhlp. It can also be used by developers who construct help environments for PL/SQL development to arrange a comfortable viewing area.

17.4.3 Creating Help Text

A user cannot access help text unless you place that text inside the appropriate program unit. PLVhlp reads its text from the ALL_SOURCE data dictionary; the text is, in other words, stored with the program about which help is needed (the implementational aspects of this technique are explored later in the chapter).

To give you an idea of the format of this help text, consider the following fragment of a package specification:

```
PACKAGE call
IS
   PROCEDURE maintain;
/*ABOUT
The maintain procedure maintains the current
set of calls in the system.
ABOUT*/
END call;
```

The strings **`/*ABOUT`** and **`ABOUT*/`** follow the standard for the starting and ending strings, respectively, of a block of text that will be retrieved by this call to **`PLVhlp.show`**:

```
SQL> exec PLVhlp.show ('s:call', 'about');
```

PLVhlp provides two functions to help you follow the standard for help text comment blocks: **`help_start`** and **`help_end`**. With both functions, you provide the keyword for text block and receive in return the strings you need to wrap around your help text.

The PLVgen package uses these two functions to generate a block of help text you can then cut and paste into your own code. Here is the implementation of the **`PLVgen.helptext`** procedure:

```
   PROCEDURE put_help
      (context_in IN VARCHAR2, indent_in IN INTEGER := 0)
   IS
   BEGIN
      IF using_hlp
      THEN
         put_line (PLVhlp.help_start (context_in), indent_in);
         put_line ('Add help text here...', indent_in);
         put_line
            (PLVhlp.help_end (context_in), indent_in, c_after);
      END IF;
   END;
```

and here is an example of the code it generates:

```
SQL> exec PLVgen.helptext ('constraints');
   /*CONSTRAINTS
   Add help text here...
   CONSTRAINTS*/
```

You can also embed blank lines in your help text, which is useful in making text easy to read. If you CREATE OR REPLACE programs with these blank lines in SQL*Plus, however, those lines will be discarded before saving the source code to the database. You can avoid this white space destruction by either not using SQL*Plus or by using a "line separator" in place of a truly blank line. For more information about the line separator feature in PL/Vision, see Chapter 7, *p: A Powerful Substitute for DBMS_OUTPUT*.

17.4.4 A PLVhlp Tutorial

The following scenario illustrates how to use these different elements to obtain help about the PLVprs package (PL/Vision PaRSe):

Step 1. Before I try to access the help text, I build a few components to make it easier to request the help. First, I add a help program to my PLVprs package specification and body as follows:

```
PACKAGE PLVprs
IS
   ... other elements ...
   PROCEDURE help;
END PLVprs;
PACKAGE BODY PLVprs
IS
   ... other elements ...

   PROCEDURE help IS
   BEGIN

      PLVhlp.show ('s:PLVprs');
   END help;
END PLVprs;
```

This provides a layer over the help package that allows me to ask for help within the context of the PLVprs package.

Step 2. I also create a SQL*Plus script (**`more.sql`**) that allows me to call the **`PLVhlp.more`** procedure with a minimum of typing:

```
SET FEEDBACK OFF
exec PLVhlp.more;
```

Step 3. I start up a SQL*Plus session, set the **`pagesize`** to 10, and access the help text for the parsing package.

```
SQL> exec PLVhlp.set_pagesize(10);
SQL> exec PLVprs.help
Help for PLVPRS

Overview: PLVprs provides a variety of PaRSing programs. It
   performs parsing for "generic" strings, but also performs
   parsing specifically for "PL/SQL strings", ie, lines of
   code from a PL/SQL program.

Programs in PLVprs:

   next_atom_loc − Returns location of next atomic.
...more...
```

The first page of text has been displayed; the "...more..." indicates that there is more text to be viewed on this topic.

Step 4. So I will ask for more help:

```
SQL> @more
Help for PLVPRS
   display_atomics − Displays distinct atomics in string.
   nth_atomic − Returns Nth atomic in string.
   number_of_atomics − Returns number of atomics in string.
   numinstr − Returns number of occurrences of substring in string.

   string − Returns string's atomics into a PL/SQL table.
   plsql_string − Returns PL/SQL code atomics into PL/SQL table.

   wrap − wraps a long string into a PL/SQL table.
   wrapped_string − wraps a long string into a string
...more...
```

And even more help....

```
SQL> @more
Help for PLVPRS
     with carriage returns embedded.
```

```
   display_wrap − displays the wrapped string
```

Notice that I no longer receive the "...more..." indicator. Now when I ask for more help, I am told that the current topic is depleted:

```
SQL> @more
No more help available...
```

17.4.4.1 Getting specialized help

What I've shown so far is general help for the PLVprs package. I can also provide more specialized areas of help. My PLVlex package, for example, contains the following help procedure:

```
PROCEDURE examples IS
BEGIN
   PLVhlp.show ('s:PLVprs', PLVhlp.c_examples);
END examples;
```

With this procedure in place I can now also ask for information about examples for the PLVlex package:

```
SQL> exec PLVlex.examples
Help for PLVLEX
Examples
```

Here is a call to get_next_token from PLVcase, which UPPER−lower cases PL/SQL programs:

```
   LOOP
      PLVlex.get_next_token
        (text, curr_pos, token, next_pos, FALSE, text_len, TRUE);

      EXIT WHEN v_token IS NULL OR   line_in.pos > line_in.len;
   END LOOP;

The FALSE indicates that I do not want to skip over spaces. This
maintains the program's indentation. The TRUE indicates that I want
qualified names (X.Y.Z) to be returned as one token.
```

You are not restricted to creating help topics of "main" and "examples." You can use whatever designators you want. Suppose, for example, that you wanted to provide help on cursors available in the company package. You would create a block of comment text in that package specification as follows:

```
/*CURSORS
Company−related Cursors:
...
CURSORS*/
```

Then you could add another help−delivering procedure to that package:

```
PROCEDURE on_cursors IS
BEGIN
   PLVhlp.show ('s:PLVprs', 'cursors');
END on_cursors;
```

With these pieces in place, the following command in SQL*Plus delivers the help you want:

```
SQL> exec company.on_cursors;
```

And that is how developers can use PLVhlp to get a handle on the stored code available to them. Obviously, someone has to spend the time to enter the help text, breaking it up into useful sections, etc. Once that is done, however, PLVhlp makes that information readily available to any user of the package.

PREVIOUS
17.3 What Is "Online Help" for Stored Code?

HOME
BOOK INDEX

NEXT
17.5 Implementing PLVhlp

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8*i* Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

17.5 Implementing PLVhlp

I hope that you put PLVhlp to use in your environment; the earlier sections in this chapter should provide a clear guide to doing so. My objective with this book, however, is to also help you develop a philosophy and perspective on your PL/SQL development so that you can build your own utilities like PLVhlp. The rest of this chapter explores the issues, technical challenges, and solutions I encountered in my implementation of PLVhlp.

To show the iterative, spiraling nature of software development, I will step through the implementation of PLVhlp in two stages. In stage one, I build a relatively simple, but effective working model of a script to deliver online help. This version is based on SQL and works only in the SQL*Plus environment. In the second stage, I build a more comprehensive, flexible, and feature−rich online help architecture based on PL/SQL packages (PLVhlp, of course).

17.5.1 Principles for Online Help

The great thing about PL/SQL is that you can use it darn near anywhere: UNIX servers in SQL*Plus, Oracle Forms, home pages on the Internet... you name it and PL/SQL rears its pretty face. Furthermore, the scope of PL/SQL will also almost certainly expand over the years as well, as it becomes one of the key enabling technologies for application partitioning in n−tier environments (now it's three−tier, but why not four− or five−tier? The more layers the better?).

The ubiquitousness of PL/SQL implies to me at this point in time that I want an online help approach to be, shall we say, "lowest common denominator." The "lowest" of LCD refers to (a) the execution environment, (b) the location of the help text in one's environment, and (c) the methodology for retrieving and displaying the help text.

As of the publication of this book, the most common execution environment, and the most generic, is SQL*Plus. The lowest, or deepest, location in an Oracle environment is the database. The most widely available delivery mechanism for PL/SQL online help is... PL/SQL!

Given this analysis, here is my vision of how an online help system for PL/SQL would work: provide a programmatic interface (via a package) to help text. This interface (a set of procedures and functions) could be called from within SQL*Plus with an execute statement. But, heck, since it's based on PL/SQL, you could also build a frontend in PowerBuilder or Oracle Forms to access this same information. You could also execute calls to the interface from within a tool like Procedure Builder. For me, however, the most important baseline is to construct that procedural interface and make it work in good, old SQL*Plus.

Let's talk about the help text itself. This is actually the single greatest challenge in building an online help system; someone has to take the time and make the effort to enter text. I know that I have very little patience for this kind of polishing effort, so at a minimum I believe that it is critical that I be able to enter my help text only once and yet have it satisfy the following needs:

1.

Function as inline documentation. The text should be useful to developers who are enhancing and maintaining the source code itself. In this situation, developers do not access this text via a programmatic interface. Rather, they edit the code, and (if they are especially methodical) they simultaneously update any help text/documentation that is affected by their changes.

2. *Be available for online help.* The same text should also be accessible and readable so that it can be displayed online to users of the code.

The only way to satisfy both of these needs with one version of text is to place the documentation directly within the definition of the program. This text is then stored in the database along with the program. That text is then available to a developer and also to a user (through the programmatic interface).

To summarize my principles of online help:

1. *Centralize help text; avoid redundant entry of text.* Do this by storing help information with the program itself, in the data dictionary.

2. *Make the help text accessible from any environment in which* PL/SQL *programs can be executed.* In other words, implement the delivery mechanism for help through PL/SQL programs.

Working from these principles, I first examine how my help text is stored in the data dictionary and how I can best retrieve it. Then I build a working prototype of a module that delivers online help for PL/SQL programs.

17.5.2 Locating Text in the Data Dictionary

When you CREATE OR REPLACE a program (procedure, function, or package) into the Oracle database, the source code is saved to the SYS.SOURCE$ table. You can view the contents of this table for all of your stored programs by accessing USER_SOURCE view. The structure of this view is as follows:

```
SQL> desc user_source
 Column                  Column
 Name           Null?    Type
 -------------- -------- -------------
 NAME           NOT NULL VARCHAR2(30)
 TYPE                    VARCHAR2(12)
 LINE           NOT NULL NUMBER
 TEXT                    VARCHAR2(2000)
```

The NAME column contains the name of the object. The name is always stored in upper case unless you surround the name of your program in double quotation marks at creation time. I assume in my help implementation that you don't do this and that your program name is always upper−cased. The TYPE is a string describing the type of source code, either PROCEDURE, FUNCTION, PACKAGE, or PACKAGE BODY (*always* upper case). The LINE is the line number and the TEXT is the line of text. Notice that a line of text may be up to 2000 bytes in length.

Quiz Time

Can the USER_SOURCE view contain blank lines? If you only use SQL*Plus to compile and store programs, you might not think this was possible. SQL*Plus automatically strips out blank lines before passing them on to the SQL layer. So all blank lines from your file−based code are compressed in USER_SOURCE. If you use another method for compiling and saving, such as Oracle Procedure Builder, this is not necessarily the case.

Suppose that I issue the following command in SQL*Plus:

```
SQL> create or replace PROCEDURE temp IS
  2  BEGIN
  3     DBMS_OUTPUT.PUT_LINE ('hello world');
  4  END;
  5  /
```

My program will be stored in the database. The following query then retrieves the source code for display:

```
SQL> SELECT text FROM USER_SOURCE
  2   WHERE name = 'TEMP' AND type = 'PROCEDURE'
  3   ORDER BY line;
TEXT
---------------------------------------------------
PROCEDURE temp IS
BEGIN
   DBMS_OUTPUT.PUT_LINE ('hello world');
END;
```

Notice that the CREATE OR REPLACE and / are not stored. These are part of the SQL*Plus command environment only.

Now suppose that I also include a comment in the code:

```
SQL> create or replace PROCEDURE temp IS
  2  BEGIN
  3     /* Send message to standard output. */
  4     DBMS_OUTPUT.PUT_LINE ('hello world');
  5  END;
  6  /
```

Then my query could just as easily extract only that line of text, as shown in each of the two SQL statements below. The first query returns any text that starts with a comment. The second query retrieves only the third line.

```
SELECT text
  FROM USER_SOURCE
 WHERE name = 'TEMP' AND type = 'PROCEDURE'
   AND LTRIM (text) LIKE '/*'
 ORDER BY line;

SELECT text
  FROM USER_SOURCE
 WHERE name = 'TEMP' AND type = 'PROCEDURE'
   AND line = 3;
```

In both cases, I will have retrieved my comment. This capability forms the core of the technique for online help in PL/SQL programs. Comments, which contain documentation about the program and, therefore, potential help text as well, can be stored and manipulated just as actual lines of code can be.

From here on in, it's all in the details −− but wait until you see these details!

17.5.3 Implementing Online Help

It's one thing to write a query to extract a line that starts with a comment marker. It's quite another challenge to generalize that query into a SELECT statement that selectively retrieves only those lines related to a specific topic. Suppose, for example, that my source code looks like this:

```
PROCEDURE sayhi IS
/*
|| Overview: the sayhi procedure uses the
||   DBMS_OUTPUT builtin to say hello
||   to the world.
*/
BEGIN
   DBMS_OUTPUT.PUT_LINE ('hello world');
END;
```

My task is now to write a query that displays only the comment information. How is that range of information defined? In this simple case, I want to display all lines after the first instance of /* and up to the very next first occurrence of */ after that line. In pseudo−SQL, I have something like this:

```
SELECT text
  FROM USER_SOURCE
 WHERE name = 'SAYHI'
   AND type = 'PROCEDURE'
   AND line BETWEEN
       startcmnt ('sayhi', 'PROCEDURE') AND
       endcmnt ('sayhi' 'PROCEDURE')
 ORDER BY line;
```

Actually, this isn't even pseudo−SQL. I am simply calling PL/SQL functions (**`startcmnt`** and **`endcmnt`**) from within a SQL statement, which you can and should do from Oracle Server 7.1 and onwards. This is a form of top−down design, because at this time I'm not really sure what these functions actually need to do in order to pass back their line numbers. The SQL statement looks right, which means that I am ready to take it to the next level of detail.

17.5.3.1 Functions to scan USER_SOURCE

The **`startcmnt`** and **`endcmnt`** functions both scan the USER_SOURCE view for specific lines.[2] Below is the header for the **`startcmnt`** function. (By the way, it might seem that I worked all this out in advance. The reality is that I am developing this implementation as I write this chapter. My production, PL/Vision−based implementation of online help, uses a completely different approach.)

[2] All of the code described in this section is contained in the **`showhelp.all`** file in the text subdirectory.

```
FUNCTION startcmnt
   (name_in IN VARCHAR2,
    type_in IN VARCHAR2,
    nth_in IN INTEGER := 1)
RETURN INTEGER;
```

Notice that I have three arguments listed, yet only two arguments were provided to the calls to **`startcmnt`** inside the SQL statement. As I started to build this function, I realized that it would probably be reasonable to ask for the third comment block in a program. I have added the parameter, but supplied it with a default value that makes the online SQL work properly.

I make one important assumption to ease implementation in **`startcmnt`**: the comment blocks for which I search always have the start−comment symbol (**`/*`**) on a new line. I take advantage of this rule by performing an LTRIM on the text I search in the view. Here is the cursor I set up to find the start of a comment block:

```
CURSOR line_cur
IS
   SELECT line
     FROM USER_SOURCE
    WHERE name = UPPER (name_in)
      AND type = UPPER (type_in)
      AND LTRIM (text) LIKE '/*%';
```

Once I have the cursor in place, the implementation is fairly straightforward. I open a cursor into the USER_SOURCE view for all lines of text for the specified program that start with **`/*`**. Then I fetch from that cursor until the %ROWCOUNT matches the **nth_in** argument value (thereby reaching the n th comment block in the program) or until I run out of records. Here is the very concise loop:

```
OPEN line_cur;
LOOP
   FETCH line_cur INTO line_rec;

   IF line_cur%NOTFOUND
   THEN
      retval := 0;
   ELSIF line_cur%ROWCOUNT = nth_in
   THEN
      retval := line_rec.line;
   END IF;

   EXIT WHEN retval IS NOT NULL;
END LOOP;
CLOSE line_cur;
```

Should I use a cursor FOR loop in **startcmnt**? I could then avoid the explicit open, fetch, and close. That would not, however, be a good choice here, since I perform a conditional exit out of the loop. You should only use a cursor FOR loop if you truly are going to touch every record retrieved by the cursor.

The **endcmnt** function is virtually identical to **startcmnt** (see the **showhelp.all** file). The only differences show up in the SQL statement of the explicit cursor. This is the **endcmnt** cursor:

```
CURSOR line_cur
IS
   SELECT line
     FROM USER_SOURCE
    WHERE name = UPPER (name_in)
      AND type = UPPER (type_in)
      AND text LIKE '%*/' || CHR(10);
```

There are two differences:

1. I need to look for a match on **%*/** instead of **/*%** since I am looking for the termination of a comment, rather than its beginning.

2. I append a CHR(10) or newline character in my match. Why do I do this? It turns out that there is a newline character at the end of very line of code stored in the database (at least when SQL*Plus is used!). There are a number of ways one can handle this detail. I simply append the CHR(10) to my search criteria. You could also RTRIM from text.

This complication of the newline character is a good example of how a seemingly simple task can become more complicated as you deal with the reality of an implementation. PL/SQL and Oracle software in general is full of these kinds of surprises. Everything you ever do is always going to be more complicated than you first imagined, and it is a good idea to plan for this in your work

Challenge for the reader: do I really need two different functions, when they are so alike? I suggest that you practice your modularization skills by transforming **startcmnt** and **endcmnt** into a single function, thereby reducing your code volume and easing maintenance.

17.5.4 Building a SQL*Plus Help Script

Once I have both of the PL/SQL functions in place, I can return to my original SQL statement and enhance it into a generic SQL*Plus script to deliver help. To achieve this change, I remove all references to specific program names and types to SQL*Plus parameters (numbers of strings prefixed by the **&** character). I also add some SET commands to limit output from the SELECT statement to just my help text.

The following script is stored in **`showhelp.sql`**:

```
SET FEEDBACK OFF
SET VERIFY OFF
SELECT text
  FROM USER_SOURCE
 WHERE name = UPPER ('&1')
   AND type = UPPER ('&2')
   AND line BETWEEN
       startcmnt ('&1', '&2') AND endcmnt ('&1', '&2')
 ORDER BY line;
```

Notice that I automatically perform an uppercase conversion on the name and type. I am simply enforcing an assumption of my utility. By calling UPPER, I liberate the user from having to remember this kind of detail.

Now if I run this script in SQL*Plus, I get my online help:

```
SQL> @showhelp sayhi procedure
/*
|| Overview: the sayhi procedure uses the
||   DBMS_OUTPUT builtin to say hello
||   to the world.
*/
```

With **`showhelp`**, I have in place a rudimentary prototype of a help−deliverer for PL/SQL programs. It acts as a proof of concept for my technique, but it isn't really a full featured help system. There are a number of weaknesses in this implementation that I want to address:

1. *It is an SQL script specific to SQL*Plus.* It cannot be executed from within PL/SQL at all and would have to be modified even to execute in any other development environment as SQL (i.e., a non−SQL*Plus frontend to SQL).

2. *The* **`showhelp`** *procedure isn't very flexible.* It assumes that any comment is help, which may not really be acceptable. You'd want a way to clearly identify comments for use as online help.

3. *The* **`showhelp`** *procedure requires lots of typing by the user.* The more you have to type, the less likely you are to use it.

4. *Since* **`showhelp`** *is in an operating file script, users must have to access to that file.* Chances are that this would lead to distribution of copies of the script, which would make upgrades difficult. A stored PL/SQL solution would rely on the granting of execute privileges to make the utility available.

5. *The script can only read help text from the USER_SOURCE view.* Now, wait just a minute, you might find yourself protesting. Isn't that supposed to be one of the script's features? It is, but it is also a limitation. What if you want to provide help text directly from operating system files containing the original source code. A SQL−based solution would have a tough time indeed with this level of flexibility.

Let's now look at what it takes to convert this SQL*Plus script into a comprehensive packaged solution that overcomes these weaknesses.

17.5.5 Constructing an Online Help Package

I will start by exploring the implementation of the **show** procedure of PLVhlp. This procedure displays the first page of help text. A simplified version of the procedure is shown below:

```
PROCEDURE show
   (context_in IN VARCHAR2, part_in IN VARCHAR2 := c_main)
IS
BEGIN
1  PLVobj.savecurr;
2  PLVobj.setcurr (context_in);
3  PLVio.usrc;
4  PLVio.initsrc (help_start (part_in), help_end (part_in));
5  PLVio.settrg (PLV.pstab);
6  PLVio.src2trg;
7  set_more (2, 0);
8  more;
END;
```

As you can see, **PLVhlp.show** makes extensive use of PL/Vision packages, most importantly PLVio. This makes sense, since the help text is stored in USER_SOURCE. PLVio was designed to allow me to read PL/SQL source code from database tables and other repositories. If I could not use PLVio in **PLVhlp.show**, I might as well not write a book about PL/Vision.

The following sections explain each of the lines of code in the **show** procedure.

17.5.5.1 Setting the current object

Before I can use PLVio to read from the USER_SOURCE data dictionary view, I must define the current object with PLVobj. The first line of the show procedure's body saves the current settings for the PLVobj current object. I do this so that if PLVobj were being used, I could restore the current object when done showing help text.

Then I call **setcurr** to set the current object. I pass in to **setcurr** the module that was provided in the call to show:

```
  PLVobj.setcurr (context_in);
```

Now the PLVio program units can be called.

17.5.5.2 Setting up PLVio to read help text

First, I inform PLVio that I will be reading from the USER_SOURCE view:

```
PLVio.usrc;
```

Then I initialize the source so that I only read the rows in USER_SOURCE for the current object that correspond to the specified help topic or part:

```
PLVio.initsrc (help_start (part_in), help_end (part_in));
```

The first argument to **PLVio.initsrc** passes the string that should be the first line read. The second argument contains the string that signals the end of the help text. The two functions, **help_start** and **help_end**, are private to the PLVhlp body; they are not listed in the specification at all. These are used to format the start and end strings of the help text block, doing little more than attach the comment markers to the specified context string:

```
FUNCTION help_start (topic_in IN VARCHAR2 := NULL)
   RETURN VARCHAR2
IS
BEGIN
   RETURN '/*' || topic_in;
END;

FUNCTION help_end (topic_in IN VARCHAR2 := NULL)
   RETURN VARCHAR2
IS
BEGIN
   RETURN topic_in || '*/';
END;
```

I create these little functions to hide my particular implementation of the start and end indicators of a comment block. I may well want to change my approach in subsequent implementations. By using these functions, I only have to make the changes there and not scattered throughout my package.

17.5.5.3 Moving the help text to a PL/SQL table

Now that the PLVio source repository has been set and initialized, I can set the target repository and then transfer all help text to that target. This call to **`settrg`** tells PLVio that I want all calls to **`PLVio.put_line`** to put text in the PLVio target PL/SQL table:

```
PLVio.settrg (PLV.pstab);
```

Then I call the high−level **`src2trg`** procedure that simply batch transfers all rows of help text from USER_SOURCE to the PLVio target:

```
PLVio.src2trg;
```

This procedure hides all the **`get_line`** and **`put_line`** logic of PLVio and lets me very easily move the PL/SQL source code to my choice of target for further manipulation.

17.5.5.4 Showing the first page

There are two steps involved in showing the first page of help text: set page management variables and then display the text. This first statement:

```
set_more (2, 0);
```

sets the values of variables to control the behavior of the PLVhlp more program. In this case, the call to **`set_more`** is saying: start at line 2 (the first line is the designator, such as **`/*HELP`**, and can be ignored) and display the first page of text.

The program that actually generates the output is the **`more`** procedure, which is called as the last line in the **`PLVhlp.show`** procedure.

17.5.5.5 Leveraging and enhancing PL/Vision

The **`show`** procedure offers an excellent example of how I am able to leverage the prebuilt packages of PL/Vision to very quickly assemble new and often richly featured programs. This shows that I have reached a "critical mass" of code in PL/Vision; my earlier investment in building reusable, low−level layers of code is paying off.

Even at critical mass, however, I still find myself enhancing the base layers of code. For example, I call **`src2trg`** in **`PLVhlp.show`** to move en masse all the rows of help text to the PLVio PL/SQL table. When I was writing **`PlVhlp.show`**, **`src2trg`** hadn't yet been written. As I confronted the task at hand in

PLVhlp.show, I realized that I did not want to have to bother with all the internals of PLVio. I simply need to move all the rows out of USER_SOURCE and into the PL/SQL table so I could display the text in a highly controlled fashion.

So I stopped my development in PLVhlp and shifted gears into PLVio. I built **src2trg**, tested it, and then used it in the **PLVhlp.show** procedure. By doing so, I not only produced the functionality I needed in PLVhlp, but also expanded the capabilities of PLVio.

As I've mentioned before, this process is typical of the way I have been developing my code over the last year. Rather than work on any one package at a time, I find myself simultaneously enhancing several different packages. As I encounter the need for a new program, I check to see if there is a package already in PL/Vision into which this program should logically fall. If so, I add to that package. If not, I create a new package. With this approach, I constantly increase the amount of reusable code in my library and achieve the broadest possible impact with each new development.

Now that you have seen how I identify, read, and store my help text, let's move on to figuring out how best to display that help text.

17.5.6 Page Pausing with PLVhlp.more

One of the big issues I encountered in designing PLVhlp was to implement a "pause" feature. It is very difficult to fit in a single screen size all the information you want or need to present about any reasonably complex program. And yet it is also very hard for a user to watch thirty, sixty, ninety lines of text scroll rapidly by without feeling the onset of panic.

SQL*Plus handles this situation very nicely with the SET PAUSE ON and SET PAGESIZE environment commands. You simply specify the number of lines in a page and turn pause "on." Then whenever you execute a query in SQL*Plus, it automatically halts output after *n* lines until you press Enter. If I was using the very first implementation of online help I shared with you in the last issue, I could (and did) rely on the SQL*Plus pause feature to implement page pausing for online PL/SQL help.

I discovered, however, that while the single−query solution to online help was simple and easy, it lacked a wide variety of features I needed to implement. The solution was to move to a PL/SQL program that queried rows from the database and then displayed each row of text with a call to the builtin DBMS_OUTPUT.PUT_LINE. Could I rely on SQL*Plus's pause facility to control this output? Not a chance. For one thing, this feature was designed to work only with SELECT statement output. For another, when you start or execute a PL/SQL program from within SQL*Plus, all control is turned over to the runtime engine of PL/SQL. And no output is generated until the program finishes. So if the **hlp** procedure found 2000 rows of help text, it would all come spewing out uninterrupted (unless it first exceeded the size of the buffer in SQL*Plus!).

17.5.6.1 A futile first effort

What's an obsessed developer to do? One idea I had was this: Inside the cursor FOR loop of the **hlp** procedure, which reads and displays a line of text, call the DBMS_LOCK.SLEEP program to pause execution of the program for perhaps 10 seconds every 25 lines. This gives the developer time to read the help text. And you wouldn't even have to press Enter to continue. It would figure it out all by itself! A loop that paused every 10 rows would look like this:

```
FOR text_rec IN text_cur (v_name, v_type)
LOOP
   DBMS_OUTPUT.PUT_LINE
     (RTRIM (text_rec.text, CHR(10)));
   IF MOD (text_cur%ROWCOUNT, 10) = 0
   THEN
```

```
            DBMS_LOCK.SLEEP (10);
         END IF;
      END LOOP;
```

Is this a clever use of the SLEEP program or what? I strutted like a peacock in front of my computer as I set up this implementation. Then it was time to test. So I ran the modified **hlp** program against a block of 100 lines of text. Normally it took about three seconds to display this text. How long do you think I had to wait before I saw the first ten lines of code? One second? Three seconds? Would you believe one minute and forty−three seconds? Yes, that's right: I waited 103 long, bewildering seconds −− and then all 100 lines of text blew by me without a single pause. What had happened?

Everything worked just the way I'd asked it to −− I just hadn't fully understand what it was I had asked for. The **hlp** program did write ten lines to the DBMS_OUTPUT buffer and then did go to sleep for ten seconds −− ten times straight. Remember: you don't ever see any output at all from DBMS_OUTPUT until the entire program terminates and returns control to the host environment, be it SQL*Plus or Procedure Builder.

My conclusion from this fruitless effort? If I was going to interrupt successfully the output from a PL/SQL program, I would have to actually stop that program so that it could dump its buffer −− and then run it again to display more text. At this point, then, there was no doubt that I would need to move to a package−based implementation. Why? Because I was talking about executing more than one program that would share information about the help text (the text itself, the last row displayed, etc.). Only the package allows me to create persistent, memory−based data.

17.5.6.2 A successful pause architecture for PL/SQL

There were two basic architectures I could use to display *n* lines of help text before terminating execution of the **PLVhlp.show** program:

1. Use a package−based cursor. If I define the cursor that accesses USER_SOURCE at the package level, it remains open in between calls to programs that fetch from the cursor and display the text. The **PLVhlp.show** program would open the cursor, fetch the first n rows, display them, and terminate. When **PLVhlp.more** is called, it just keeps fetching the next set of records from the cursor.

2. Transfer all lines of help text into a PL/SQL table and then display the next n rows each time **PLVhlp.more** is called.

In both cases I take advantage of a central feature of PL/SQL−based data structures: they persist for the duration of an Oracle session. The cursor stays open, and the PL/SQL table remains populated, in between program calls. The first, cursor−centered approach is simpler since it does not involve an intermediate PL/SQL table. It is probably the technique I would have used had I not already built PLVobj, PLVio, and PLVtab. Given these prebuilt components, however, it was a no−brainer for me to pursue the PL/SQL table solution.

Having gone with the PL/SQL table, however, another significant advantage became obvious: help text deposited in this data structure could be managed in a very flexible and efficient way. The text could, for example, be passed on to another environment for display, such as Oracle Forms or a third−party PL/SQL development environment.

17.5.6.3 Implementing more

The **more** procedure of PLVhlp, shown below, contains the logic necessary to display a page of rows of the PL/SQL table, as shown in Example 17.1:

Example 17.1: The PLVhelp.more Procedure

```
 1   PROCEDURE more IS
 2   BEGIN
 3      IF v_more
 4      THEN
 5         PLVio.disptrg
 6            ('Help for ' || PLVobj.currname,
 7             v_startrow, v_endrow);
 8
 9         IF v_endrow = PLVio.target_row-1
10         THEN
11            PLVobj.restore_object;
12            v_more := FALSE;
13         ELSE
14            p.l ('...more...');
15            set_more (v_endrow + 1, v_endrow);
16         END IF;
17      ELSE
18         p.l ('No more help available...');
19      END IF;
20   END;
```

As with **PLVhlp.show**, the more program is short and relies heavily on the PLVobj and PLVio packages. It also makes two calls to the **p.l** procedure. As I've described in Chapter 7, the **p** package and its **l** procedure provide a substitute for DBMS_OUTPUT.PUT_LINE that requires much less typing and offers additional functionality (such as displaying rather than ignoring NULL text and automatically substringing the text to a maximum length of 255 bytes to avoid VALUE_ERROR exceptions.)

Now let's go examine the **more** program, so that you can fully understand the implementation of a PL/SQL−based, page−pausing mechanism. There are three private package variables used to manage behavior in the **PLVhlp.more** package, as listed below:

v_more	TRUE if there are more lines of help text to display.
v_startrow	The first row in the PL/SQL table of the next page of text.
v_endrow	The last row in the PL/SQL table of the next page of text.

At the very start of **more**, I check the value of **v_more**. If it is FALSE, I display an appropriate message. If **v_more** evaluates to TRUE, then I display the next page of text using the PLVio package (lines 5 through 7 of Example 17.1):

```
PLVio.disptrg ('Help for ' || PLVobj.currname, v_startrow, v_endrow);
```

The **disptrg** or "display target" program displays the contents of the PL/SQL table maintained as a target by the PLVio package (I never have to declare it or directly manipulate its contents; the PLVio package takes responsibility for this work). I provide a header and a range of rows to display. This is another good example of how the abstraction in my lower−level packages makes programming a breeze.

17.5.6.4 Maintaining the current page variables

Now that I have displayed this latest page, I reset the triad of variables for the next call to **more**. First, I check to see if there are any more rows. I know I am done when the end row matches the second−to−last row in the target table. I ignore the very last line, because it is simply the comment marker (**HELP*/**, for example). I then restore the current object in PLVobj and set **v_more** to FALSE:

```
IF v_endrow = PLVio.target_row-1
THEN
   PLVobj.restore_object;
   v_more := FALSE;
```

If, on the other hand, I have not displayed all rows, then I display a message indicating that there is more to come (when and if the user executes **PLVhlp.more** again) and then call the **set_more** program to set up my variables:

```
p.l ('...more...');
set_more (v_endrow + 1, v_endrow);
```

The **set_more** program is also not very complicated:

```
PROCEDURE set_more
   (start_in IN INTEGER, end_in IN INTEGER)
IS
BEGIN
   v_startrow := start_in;
   v_endrow :=
      LEAST (v_pagesize + end_in, PLVio.target_row−1);
   v_more := TRUE;
END set_more;
```

Translation: the start row is set to the next row after the current end row. The end row is set to the smaller of these two values: (a) the current end row plus another page's worth of lines or (b) the last row to be displayed in the PL/SQL table. Finally, **v_more** is set to TRUE just to make sure.

To help you understand how the page−management variables shift the rows being displayed, let's step through a scenario. Suppose that I have set the pagesize to 10 lines (**v_pagesize**) and I have 24 lines of text in my help section for the PLVprs package (PLV**io.target_row**). When I call **PLVprs.help**, the initializing call to **set_more** sets the page−management variables as follows:

```
set_more (2, 0);                  v_startrow := 2
                                  v_endrow := LEAST (10+0, 24−1) = 10
```

As a result, lines 2 through 10 are displayed on the screen. Then **set_more** is called again with the following arguments and results:

```
set_more (v_startrow, v_endrow);  v_startrow := v_endrow + 1 = 10 + 1 = 11
                                  v_endrow : = LEAST (10+10, 24−1) = 20
```

When I call **PLVhlp.more** to see the next page of help, lines 11 through 20 are displayed and **set_more** is then called with these results:

```
set_more (v_startrow, v_endrow);  v_startrow := v_endrow + 1 = 20 + 1 = 21
                                  v_endrow : = LEAST (10+20, 24−1) = 23
```

And since PLVhlp tells me so, I call **PLVhlp.more** to see the next page of help, lines 21 through 23; **set_more** is then called. This time around, **v_endrow** does equal the last line of text to be displayed, so **v_more** is set to FALSE, the PLVobj current module values are restored, and **PLVhlp.more** is, effectively, disabled.

This PL/SQL table−based technique for page−pausing is not as easy to use as the one built into SQL*Plus. With SQL*Plus, you simply press the Enter key to see the next page. With PLVhlp, you have to type:

```
SQL> exec PLVhlp.more
```

or, if you create a standalone procedure as I have, simply:

```
SQL> exec more
```

or, if you create a SQL script that performs the above **exec**, you could even simplify that to:

```
SQL> @more
```

Regardless, it's more work than what you do with SQL*Plus, but considering that it is an add−on layer of code and functionality, it's not too odious. Of course, if you do not use SQL*Plus, then we can skip the comparison and simply celebrate the ability to build such utilities in PL/SQL.

17.5.7 The Component Approach to PL/SQL Utilities

In the earlier part of the chapter, I showed you how to construct with a minimum of coding and fuss a functional utility to provide online help for PL/SQL programs. This utility consisted of a single SQL statement and it got the job (narrowly defined) done. This script did, on the other hand, have its limitations. The solution I offered in the second implementation of online help went beyond simply providing a handy tool. Instead, it took what I call a "component" approach to providing a robust environment in which PL/SQL developers can make their requests.

I take advantage of the PL/SQL package structure to allow a user of PLVhlp to modify the way help is delivered to that user. You don't simply ask to view the help text. You determine the size of a logical page of text and you control what you see when.

This shift of control to the user, this anticipation of user needs, this robustness of implementation all distinguish the powerful plug−and−play component in PL/SQL from its poor cousin, the utility. Building a component is certainly more challenging. You have to be more creative and more flexible. You have to write more code and make what you write more sophisticated. I cannot, however, understate the payoff. People (including yourself!) will use −− and reuse −− your code. They will be more productive and their programs will be noticeably less buggy and easier to maintain.

PREVIOUS | HOME | NEXT
17.4 Using PLVhlp | BOOK INDEX | 18. PLVcase and PLVcat: Converting and Analyzing PL/SQL Code

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

18. PLVcase and PLVcat: Converting and Analyzing PL/SQL Code

Contents:

PL/Vision provides two packages to convert and analyze PL/SQL source code: PLVcase and PLVcat. The PLVcase (PL/Vision CASE) package converts the case of PL/SQL source code so that it conforms to the UPPER−lower method (reserved words in uppercase, application−specific identifiers in lowercase). The PLVcat (PL/Vision CATalogue) package catalogues PL/SQL source code so that you can analyze the contents of your program for cross−references, dependencies, and so on. These two packages are explored in this chapter.

18.1 PLVcase: Converting the Case of PL/SQL Programs

The PLVcase conversion package converts the case of PL/SQL source code according to the UPPER−lower method: all reserved words are converted to upper−case, while all application−specific identifiers are converted to lowercase. The consistency of case in a program −− and, in particular, the UPPER−lower standard −− aids greatly in the ability of developers to read, understand, and maintain their (and others') code.

Consider the following package body definition:

```
create or replace package body testcase
is
   procedure save (string_in in varchar2)
   is
      n integer := dbms_sql.open_cursor;
   begin
      update PLV_output set program = string_in;
      if sql%rowcount = 0
      then
         insert into PLV_output values (string_in);
      end if;
      PLVcmt.perform_commit;
   end;
end testcase;
/
```

This program is consistently indented to reveal the logical flow of the save procedure defined in the testcase package. Yet it is still difficult to read. With all the text in lowercase, the syntax and logic blurs into an indistinguishable stream of characters.

Here is the same package body after being passed through PLVcase:

```
PACKAGE BODY testcase
IS
   PROCEDURE save (string_in IN VARCHAR2)
   IS
      n INTEGER := DBMS_SQL.OPEN_CURSOR;
   BEGIN
      UPDATE PLV_output SET program = string_in;
      IF SQL%ROWCOUNT = 0
      THEN
         INSERT INTO PLV_output VALUES (string_in);
      END IF;
      PLVcmt.perform_commit;
   END;
```

```
END testcase;
```

PLVcase is a general−purpose engine for case conversion. It relies on many different packages in PL/Vision to achieve a high degree of flexibility. For example, PLVcase can read and convert PL/SQL source code from the data dictionary, an operating system file (with PL/SQL Release 2.3), and even individual string variables. You can redirect PLVcase for both read and write by making the appropriate calls to the PLVio package.

PLVcase also relies heavily on the PLVtkn package. After all, if PLVcase is going to uppercase only keywords, it has to know which identifiers in a PL/SQL program are reserved words. This information is maintained in the **PLV_token** table, which in turn is made available through PLVtkn (see Chapter 10, *PLVprs, PLVtkn, and PLVprsps: Parsing Strings*).

The body of the PLVcase package is actually rather simple considering the complexity of its task. The following sections show how to use each of the different elements of the PLVcase package.

18.1.1 The Various Conversion Procedures

PLVcase offers a sequence of procedures to convert increasingly complex text. The **token** procedure converts a single token.

line
: Converts the text found with a record of the **PLVio.line_type** structure.

string
: Converts the text in a simple string.

module
: Converts all the lines of code in the specified module.

modules
: Converts multiple programs in a single pass.

As you would expect, **line** uses **token**, **string** uses **line**, **module** uses **line**, and **modules** uses **module**.

It is easy to use the conversion programs. The more complicated aspect of PLVcase arises in determining the source of the program (data dictionary view, file, string, etc.) and the target for the converted code. This is discussed later in the chapter.

18.1.1.1 Converting a single token

The **token** function can take two arguments as shown in the header:

```
FUNCTION token (token_in IN VARCHAR2, pkg_in IN VARCHAR2 := NULL)
RETURN VARCHAR2
```

The second argument is an optional name of a package. If **pkg_in** is provided, that package name is prefixed onto the token value and then that string is case−converted.

Here is an example of using **PLVcase.token** to convert a single identifier:

```
v_newtoken := PLVcase.token (v_oldtoken);
```

In this second example, I request conversion of a program from the DBMS_SQL package:

```
v_newtoken := PLVcase.token ('open_cursor', 'dbms_sql');
```

In this case, **`v_newstring`** is set to OPEN_CURSOR. In other words, the package name is not prefixed onto the token name.

18.1.1.2 Converting a string

To convert the case of all tokens in a string, use one of the **`PLVcase.string`** program units. The **`string`** program is overloaded as follows:

```
PROCEDURE string (string_inout IN OUT VARCHAR2);
FUNCTION string (string_in IN VARCHAR2) RETURN VARCHAR2;
```

I provide these two versions of **`string`** to support different applications of this functionality. In some cases you just want to pass a string to the PLVcase package for conversion. In this scenario, use the string procedure as follows:

```
PLVcase.string (v_header);
```

Under other circumstances, you may want to preserve the original string value (it might, for example, be an IN parameter), as well as generating a case−converted version of the string. In this scenario, use the string function as follows:

```
v_newstring := PLVcase.string (original_in);
```

Another use of PLVcase is illustrated by the PLVgen package. This package uses PLVcase to apply the proper case to any symbolic default values provided by the user. For example, I can call **`PLVgen.func`** to generate a function with a default return value that would have mixed cases as follows:

```
SQL> exec PLVgen.func ('tot_sales', 1, 'sales_to_date (sysdate)');
```

The function name, **`sales_to_date`**, should be in lowercase, while the argument to that function, SYSDATE, should be in uppercase. The construction of the initial return value inside the PLVgen package uses the **`PLVcase.string`** function to accomplish this effect:

```
retval :=
   v_name || ' ' || v_datatype || ' := ' ||
   PLVcase.string (v_defval) || ';';
```

18.1.1.3 Converting a line

PLVcase is "PLVio−aware." It allows you to convert the text contained in a line record defined in the PLVio package. The header for the **`line`** procedure is:

```
PROCEDURE line
   (line_in IN OUT PLVio.line_type,
    line_out IN OUT PLVio.line_type,
    found_out OUT BOOLEAN);
```

Notice that the original line record is left unchanged. Instead, the modified line text is deposited in an OUT record of the same line type. The **`found_out`** parameter is set to TRUE if at least one token is found in the line's text.

The **`PLVcase.line`** procedure makes use of **`PLVlex.get_next_token`** to parse out the next token in the line's text and then convert the case using PLVtkn. The **`PLVcase.string`** function, in turn, calls **`PLVcase.line`**. Finally, the **`PLVcase.string`** procedure calls the **`PLVcase.string`** function.

To give you a sense of the layering involved, here is the body of the **`PLVcase.string`** function:

```
FUNCTION string (string_in IN VARCHAR2)
```

```
   RETURN VARCHAR2
IS
   line1 PLVio.line_type;
   line2 PLVio.line_type;
   code_found BOOLEAN := FALSE;
BEGIN
   first_token := TRUE;
   PLVio.initline (line1, string_in, LENGTH (string_in), 1);
   PLVio.initline (line2);
   line (line1, line2, code_found);
   IF code_found
   THEN
      RETURN line2.text;
   ELSE
      RETURN NULL;
   END IF;
END;
```

18.1.1.4 Converting a module

The **module** procedure allows you to convert all the lines of code in a specified program unit with one procedure call. The header for **module** is as follows:

```
PROCEDURE module
   (module_in IN VARCHAR2,
    cor_in IN VARCHAR2 := c_usecor,
    last_module_in IN BOOLEAN := TRUE);
```

The three arguments of **module** are explained below:

module_in
: The name of the module. You can pass in an nonqualified name, such as **total_sales**. You can also pass in a string in the format *type:name*. Valid values for type are listed in the next section.

cor_in
: Use this argument to pass in the constant **PLVcase.c_usecor** if you want PLVcase to automatically attach CREATE OR REPLACE syntax in the converted source code. You do this if you are converting source code that will be recompiled back into the database using SQL*Plus.

last_module_in
: Pass in TRUE if this is the last module to be converted. The default is TRUE, since module is intended to convert a single program unit at a time. When **last_module_in** is TRUE, the PLVio−directed source is closed and all changes are saved. This argument is needed to allow **PLVcase.modules** to convert multiple program units.

The PLVcase package uses the PLVobj interface to the ALL_OBJECTS data dictionary view to identify the program or programs indicated by the user. PLVobj is flexible in interpreting your input. The following variations are allowed: *name*, *schema.name*, *type:name*, and *type:schema.name*.

PLVcase.module relies on the PLVio to determine the location or target repository for the converted source code. If you have previously executed this statement:

```
PLVio.settrg (PLV.pstab);
```

then the converted code will be placed in **PLVio.target_table**. If, on the other hand, you execute this statement:

```
PLVio.settrg (PLV.stdout);
```

then you see the converted code scrolled onto your screen when the conversion process is complete.

You can also convert the case of more than one module at a time by calling the **modules** procedure, whose header is:

```
PROCEDURE modules (module_spec_in IN VARCHAR2 := NULL);
```

The following examples using **Plvcase.module** and **PLVcase.modules** give you a sense of how you can apply this functionality in your own environment.

1. Convert a stored function named **total_sales** without CREATE OR REPLACE syntax as follows:

   ```
   PLVcase.module ('f:total_sales', PLVcase.c_nousecor);
   ```

2. Convert the case of (a) a single function and (b) all the package specifications in the current schema:

   ```
   SQL> exec PLVcase.module ('total_sales');
   SQL> exec PLVcase.modules ('s:%');
   ```

In the call to **PLVcase.modules** I specify "s" or "specification" for the module type and "%" or "any and all" for the names of the modules.

18.1.2 A Script to Convert Programs

Most of the time when you use PLVcase on your source code, you will not simply execute a single call to a PLVcase procedure. PLVcase is too tightly integrated into and dependent on other packages in PL/Vision. For example, you first need to initialize the PLVio package so that PLVcase can find the original source code and properly write out the converted code.

I wrote the **setcase.sql** SQL*Plus script, shown in Example 18.1, to make it easier for developers to use PLVcase properly. This program prompts you for the module or modules you wish to convert. It then assumes that you want to read the source code from the ALL_SOURCE data dictionary −− and calls **PLVio.asrc** to "make it so." It also assumes that you want the converted source code simply CREATEd OR REPLACEd back into the database. It accomplishes this with the following call:

```
PLVio.settrg (PLV.dbtab);
```

This setting means that the converted code will be written to the **PLV_source** table. The **setcase** script then uses a SELECT on this table to write the new version of the source code to a command file called **setcase.cmd**. As a final step, **setcase.sql** executes the **setcase.cmd** file in SQL*Plus and the code is reinserted into the database.

Example 18.1: The setcase Script

```
SET ServerOutput ON
SET FEEDBACK OFF
SET VERIFY OFF
DELETE FROM PLV_source;
DECLARE
   modname VARCHAR2(100) := UPPER ('&1');
   modname2 VARCHAR2(100);
   modtype VARCHAR2(100);
   modschema VARCHAR2(100);
   modstring VARCHAR2(100);
   delim_loc INTEGER;
```

```
BEGIN
   IF INSTR (modname, '%') > 0 OR INSTR (modname, ':') = 0
   THEN
      /* Doing >1 module. */
      p.l ('=========================');
      p.l ('PL/Vision Case Conversion');
      p.l ('=========================');
      p.l ('Converting ' || modname || '...');
      PLVio.asrc;
      PLVio.settrg (PLV.dbtab);
      PLVcase.modules (modname);
   ELSE
      modname2 := modname;
      PLVobj.convobj (modname, modtype, modschema);
      modstring := modtype || ' ' || modname;
      p.l ('=========================');
      p.l ('PL/Vision Case Conversion');
      p.l ('=========================');
      p.l ('Converting ' || modstring || '..');
      PLVio.asrc;
      PLVio.settrg (PLV.dbtab);
      PLVcase.module (modname2);
   END IF;
END;
/
prompt Generating program creation script...
set pagesize 0
set linesize 120
set termout off
column text format a120
spool setcase.cmd
SELECT text
  FROM PLV_source
 ORDER BY
DECODE (type,
        'PACKAGE',    1, 'PACKAGE BODY', 2,
        'PROCEDURE', 3, 'FUNCTION',     4,
        5),
   name, line
/
spool off
set pagesize 25
set termout on
set feedback on
start setcase.cmd
```

See the sections on the PLVobj and PLVio packages for more information about how you can alter the source and target repositories for the source code converted by PLVcase. You could, for instance, convert program units stored in Oracle Forms tables in the database.

PREVIOUS | HOME | NEXT
17.5 Implementing PLVhlp | BOOK INDEX | 18.2 PLVcat: Cataloguing PL/SQL Source Code

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

18.2 PLVcat: Cataloguing PL/SQL Source Code

The PLVcat package gives you a way to parse and store (in a table−based catalogue) information about the contents of PL/SQL package specifications. You will glean information from this catalogue that is currently unavailable from the Oracle Server data dictionary.

Before describing the different areas of functionality of PLVcat, let's look at the problem PLVcat is intended to solve.

18.2.1 Knowing What You've Got

One of the tremendous advantages of storing PL/SQL programs in the database is that they can be executed by anyone granted authority to those objects. You don't have to know where the source code resides on disk; link lists and paths will not badger you in the world of PL/SQL. Yet you do face other challenges when seeking to leverage stored code −− especially on an enterprise−wide basis. These obstacles include:

- *Knowing what is available.* How do you know what programs are stored where, what they are supposed to do, and how you are supposed to use them?
- *Knowing where and how programs are being used.* How do you measure the impact of changing a particular program? It is often very useful to be able to answer questions like which programs use this function? How, then, is the function used?

At the time I'm writing this book, I think that it is fair to say that the potential and functionality of PL/SQL have outstripped the features of development environments using PL/SQL. Developers using PL/SQL often work within a light fog, stumbling about for information on the programs they can use. As a result, code reuse remains a lofty objective, an agreed−upon principle rarely put into practice.

Oracle provides some ability to get answers to your questions about stored PL/SQL It provides a set of data dictionary views that you can access using the SQL language. These views include:

USER_SOURCE
The source code of your stored programs. (There is also an ALL_SOURCE view, which contains the source code of all programs to which you have access.)

USER_DEPENDENCIES
Information about dependencies between all kinds of objects stored in the database.

Having this kind of information in views is a wonderful feature of an active data dictionary. But you have to ask yourself two questions: how easy is it to get the data and how good is the data? Sure, you can use SQL to retrieve the information from the views, but that can get very time−consuming and cumbersome. Suppose you want to see a list of all of the programs defined in a package specification. You *could* view that specification using the **PLVvu.code** procedure. That can at times be an overwhelming volume of information −− and it doesn't lend itself easily to generating a report showing this information in an outline view.

In the case of USER_DEPENDENCIES, the problem is not access to data; the problem is with a lack of data. This view shows you which package another object relies on, but it does not "pierce the veil" of the package to show you which element *inside* that package is the cause of the dependency. In other words, if my **calc_totals** program makes the following call:

```
recalc.full_analysis;
```

then the USER_DEPENDENCIES view shows that **calc_totals** is dependent on **recalc**. It will not, however, inform you that **full_analysis** is the program of the **recalc** package that is called by **calc_totals**.

PL/Vision fixes these shortcomings with a set of programs that parses the contents of your PL/SQL code and then stores the results of that process in database tables. You can then write simple SQL statements against these tables to generate reports that provide a much greater granularity of detail about your PL/SQL programs.

These elements of the PLVcat package are explained in later sections.

18.2.2 The PLVcat Database Tables

The PLVcat programs generate information about your PL/SQL code and then deposit that information in one of two database tables: **PLVctlg** or **PLVrfrnc**. These tables are created when you install PL/Vision. The **PLVctlg** table contains the catalogue of the contents of packages (those elements defined in the specification). The **PLVrfrnc** table contains the references or dependencies generated by calls to the **ref** programs. These tables and how to interpret their contents are explained below.

18.2.2.1 PLVctlg table

The structure of the **PLVctlg** table is:

```
CREATE TABLE PLVctlg
   (owner VARCHAR2(100),
    name1 VARCHAR2(100), /* Package name */
    name2 VARCHAR2(100), /* Element name */
    type VARCHAR2(100),  /* Same as in user_objects */
    iname VARCHAR2(100), /* Name of object inside */
    itype VARCHAR2(100), /* Type of object inside */
    idatatype VARCHAR2(100),
    overload INTEGER)
```

The **owner**, **name1**, **name2**, and **type** columns define the program unit for which elements have been catalogued. The **name2** column is always NULL in this version of PLVcat, since PL/Vision currently catalogues only package specifications.

The "inside" columns (**iname**, **itype**, and *idatatype*) indicate the element found in the program unit. The **idatatype** column is non−NULL if the element is a function or TYPE statement. The **overload** column contains the number of overloadings of a particular procedure or function name. All values are stored in uppercase.

Examples of how this table is filled from calling PLVcat modules are shown later in this chapter.

18.2.2.2 PLVrfrnc table

The **`PLVrfrnc`** table contains information about the references made to external elements from within a PL/SQL program unit. The structure of this table is:

```
CREATE TABLE PLVrfrnc
   (owner VARCHAR2(100),
    name1 VARCHAR2(100),
    name2 VARCHAR2(100),
    type VARCHAR2(100),
    reftype VARCHAR2 (100), /* Type of reference */
    rowner VARCHAR2(100), /* Leave null if not specified */
    rname1 VARCHAR2(100), /* Package name or stand alone */
    rname2 VARCHAR2(100)  /* Null if not in package. */
   )
```

The **`owner`**, **`name1`**, **`name2`**, and **`type`** columns define the program unit for which references have been analyzed. The **`reftype`**, **`rowner`**, **`rname1`**, and **`rname2`** columns define the object that is referenced inside the program unit. All values are stored in uppercase.

Examples of how this table is filled from calling PLVcat modules are shown later in this chapter.

18.2.3 Building a Catalogue

You can build a catalogue of your PL/SQL source code with the **`module`** and **`modules`** procedures. The **`module`** procedure catalogues a single program unit, while the **`modules`** procedure can handle wildcarded program names and automatically catalogue multiple program units, including all the stored code in a schema.

18.2.3.1 Cataloguing a single module

To build a catalogue of a single PL/SQL program, you call the **`module`** procedure, whose header is:

```
PROCEDURE module (module_in IN VARCHAR2);
```

You provide the name of the program you want to catalogue (currently only package specifications are supported; any types provided to the left of the **`:`** are ignored). The **`module`** program automatically sets the PLVio source repository to the ALL_SOURCE data dictionary view. It parses the source code using the PLVprsps package, searching for the definitions of any of the following PL/SQL code elements:

- Procedure header
- Function header
- Cursor header
- TYPE statement

`PLVcat.module` does not, in other words, currently catalogue variables, constants, exceptions, or other program elements that might appear in a PL/SQL package.

Since the **`module`** procedure works only with package specifications at this time, you do not have to tell PLVcat the type of object you want to catalogue when you call **`PLVcat.module`**. You simply provide the name of the package and it automatically scans the specification. As an example, to catalogue the PLVio package I would execute the following command in SQL*Plus:

```
SQL> exec PLVcat.module ('PLVio');
```

When control is returned back to the SQL*Plus prompt, the rows will have been written to **`PLVctlg`** and will be available for reporting and analysis. If your package is large, it may take a minute or two to complete the catalogue. String parsing and manipulation in the PL/SQL language is not known to be lightning fast.

18.2.3.2 Cataloguing multiple modules

The **`PLVcat.module`** procedure can only process a single package at a time; you cannot pass in wildcarded package names for multiple−program cataloguing in one call. The **`PLVcat.modules`** procedure offers this capability; its header is:

```
PROCEDURE modules (module_in IN VARCHAR2);
```

You can use **`modules`** to catalogue all the packages in your schema with this call:

```
SQL> exec PLVcat.modules ('%');
```

Or you can be more selective. The following call to **`modules`** will catalogue all packages in the PL/Vision library:

```
SQL> exec PLVcat.modules ('PLV%');
```

The case you use to specify the package names is not significant. All program names are stored in the data dictionary in uppercase. (All right, so if you surround your program name in double quotes you can actually create programs with names in mixed case in the data dictionary; if you do this, you deserve all the ensuing complexities!)

18.2.3.3 Examining the catalogue

To see how the **`PLVctlg`** table is populated by calls to **`PLVcat.module`** and **`PLVcat.modules`**, consider the following simplified version of the PLVtmr package specification:

```
PACKAGE PLVtmr
IS
   FUNCTION elapsed RETURN NUMBER;
   PROCEDURE show_elapsed;
END PLVtmr;
/
```

After cataloguing this package with **`PLVcat.module`**, I will have two rows in the **`PLVctlg`** table as follows:

Owner	Name1	Type	Iname	Itype	Idatatype	Overloading
	PLV	PLVtmr PACKAGE	elapsed	FUNCTION	NUMBER	1
	PLV	PLVtmr PACKAGE	show_elapsed	PROCEDURE	NULL	1

To obtain a list of all elements in the PLVtmr package, therefore, I could execute a SQL statement like this:

```
SELECT iname
  FROM PLVctlg
 WHERE name1 = 'PLVTMR';
```

To obtain a list of all functions catalogued for the PLV user account, I could execute a SQL statement like this:

```
SELECT iname
  FROM PLVctlg
 WHERE owner = 'PLV'
   AND itype = 'FUNCTION';
```

The script named **`inctlg.sql`** contains a more complex and useful SQL statement for viewing the contents of the catalogue. The code for this script is:

```
TTITLE 'Elements Catalogued in &1'
SET VERIFY OFF
SET PAGESIZE 66
SET LINESIZE 60
COLUMN element FORMAT A60
SELECT DECODE (idatatype, NULL, NULL, idatatype || ' ') ||
       itype || ' ' || owner || '.' ||
       name1 || '.' || iname  || ' ' ||
       DECODE (overload, 1, NULL,
               '(' || TO_CHAR(overload) || ')') element
  FROM PLVctlg
 WHERE name1 like UPPER ('&1')
 ORDER BY owner, type, name1, itype, iname;
```

This script accepts as a single parameter the name of the package whose catalogue you wish to view. Executing this script for the **`p`** package provides the following output:

```
SQL> @inctlg p
Sat Jun 01                                        page    1
               Elements Catalogued in p
ELEMENT
------------------------------------------------------------
VARCHAR2 FUNCTION PLV.P.LINESEP
VARCHAR2 FUNCTION PLV.P.PREFIX
PROCEDURE PLV.P.L (7)
PROCEDURE PLV.P.SET_LINESEP
PROCEDURE PLV.P.SET_PREFIX
PROCECURE PLV.P.TURN_OFF
PROCECURE PLV.P.TURN_ON
```

Notice that I am informed that the **`p.l`** procedure is overloaded seven times.

18.2.3.4 Cataloguing PL/Vision

You can generate the catalogue for PL/Vision packages by executing the **`plvcat.sql`** script, located in the *plvision\use* subdirectory.

The code for **`plvcat.sql`** is simply:

```
BEGIN
   FOR objind IN 1 .. PLV.numobjects
   LOOP
      PLVcat.module (PLV.objects(objind));
   END LOOP;
END;
/
```

This script takes advantage of the list of PL/Vision objects that are stored in the **`PLV.objects`** PL/SQL table. This PL/SQL table is created and assigned values in the initialization section of the PLV package. The **`plvcat.sql`** script generates 396 rows in the **`PLVctlg`** table.

18.2.4 Identifying References and Dependencies

The other major area of functionality in PLVcat is to identify the references made within a program unit to external program elements. Such a reference implies a dependency; this information can be very useful in maintaining code, analyzing reuse and impact, and so on. The Oracle7 Server does maintain some dependency information, but it is only the minimum data required by the database to validate the status of compiled code. I cannot, for example, find out from the USER_DEPENDENCIES data dictionary view how many programs use the PLVdyn.ddl procedure. The most I can determine is the set of programs that use something in PLVdyn −− and this is not enough to support adequately an enterprise−wide deployment of PL/SQL applications.

The PLVcat package offers three programs to generate dependency information right down to the name of the package element that was referenced. It even lets you catalogue references to builtin functions like SUBSTR and all the builtin package programs. The three procedures that perform this task are:

```
PROCEDURE refnonkw (module_in IN VARCHAR2);
PROCEDURE refbi (module_in IN VARCHAR2);
PROCEDURE refall (module_in IN VARCHAR2);
```

In all three cases, you pass in the name of the individual module for which you want references generated.

refnonkw
Identifies references to all non−keyword identifiers (application−specific elements).

refbi
Identifies references to all kinds of builtins.

refall
Identifies references to both non−keyword identifiers and builtins by calling **refnonkw** and **refbi**. Note that in the current implementation of **refall**, two passes are made against the specified module to parse the code.

The results of these scans are deposited in the **PLVrfrnc** table.

18.2.4.1 Examining the references

Let's look at an example of how this table is populated from the source code. Consider the **testcase** package:

```
PACKAGE BODY testcase
IS
   PROCEDURE save (string_in IN VARCHAR2)
   IS
      n INTEGER := DBMS_SQL.OPEN_CURSOR;
   BEGIN
      UPDATE PLV_output SET program = string_in;
      IF SQL%ROWCOUNT = 0
      THEN
         INSERT INTO PLV_output VALUES (string_in)
      END IF;
      PLVcmt.perform_commit;
   END;
END testcase;
```

After cataloguing all builtins in this package with the **PLVcat.refbi**, I will have four rows in the **PLVrfrnc** table as follows:

Owner	Name1	Name2	Type	Rowner	Rname1	Rname2
PLV	testcase	NULL	PACKAGE BODY	SYS	DBMS_SQL	OPEN_CURSOR
PLV	testcase	NULL	PACKAGE BODY	SYS	INSERT	NULL
PLV	testcase	NULL	PACKAGE BODY	SYS	ROWCOUNT	NULL
PLV	testcase	NULL	PACKAGE BODY	SYS	UPDATE	NULL

After extracting all non−keywords in this package with the **`PLVcat.refnonkw`**, I will have a single row in the **`PLVrfrnc`** table as follows:

Owner	Name1	Name2	Type	Rowner	Rname1	Rname2
PLV	testcase	NULL	PACKAGE BODY	PLV	PLVCMT	PERFORM_COMMIT

Notice that PLVcat does not currently store references to non−PL/SQL objects, such as the **`PLV_output`** table. The reason is that it uses DBMS_UTILITY.NAME_RESOLVE to locate the code and this builtin does not work with non−PL/SQL objects.

To see a list of all program units that call the SUBSTR builtin function, you could execute this query:

```
SELECT owner || '.' || name1 program
  FROM PLVrfrnc
 WHERE rname1 = 'SUBSTR';
```

To see a list of all program units that call the **`open_and_parse`** function of the PLVdyn package, you could execute this query:

```
SELECT owner || '.' || name1 program
  FROM PLVrfrnc
 WHERE rname1 = 'PLVdyn'
   AND rname2 = 'OPEN_AND_PARSE';
```

The script named **`inctlg.sql`** contains a more complex and useful SQL statement for viewing the contents of the catalogue. The code for this script is:

```
TTITLE 'Elements Referenced by &1'
SET VERIFY OFF
SET PAGESIZE 66
SET LINESIZE 60
COLUMN element FORMAT A60
SELECT owner || '.' ||
       name1 || ' CONTAINS ' || rname1  ||
       DECODE (rname2, NULL, NULL, '.' || rname2) element
  FROM PLVrfrnc
 WHERE name1 like UPPER ('&1')
 ORDER BY owner, type, name1, rname1, rname2;
```

This script accepts as a single parameter the name of the program whose references you wish to view. Executing this script for the **`testcase`** package body provides the following output:

```
SQL> start inrfrnc testcase
Sat Jun 01                                         page    1
              Elements Referenced by testcase
ELEMENT
------------------------------------------------------------
PLV.testcase CONTAINS DBMS_SQL.OPEN_CURSOR
PLV.testcase CONTAINS INSERT
PLV.testcase CONTAINS INSTR
PLV.testcase CONTAINS ROWCOUNT
PLV.testcase CONTAINS UPDATE



PLV.testcase CONTAINS PLVcmt.PERFORM_COMMIT
```

Special Notes on PLVcat

Here are some factors to consider when working with PLVcat:

- The PLVcat package supports only the ALL_SOURCE data dictionary view as a source repository for the PL/SQL source code. Future versions of PL/Vision Professional will support reading from operating system files.
- The **`build`** program performs a commit after completing its inserts into the **`PLVctlg`** table. The **`refnonkw`** and **`refbi`** programs perform a commit after completing their inserts into the **`PLVrfrnc`** table. These commits are executed through the PLVcmt package, so you can turn off commit processing by executing the **`PLVcmt.turn_off`** toggle (see Chapter 20, *PLVcmt and PLVrb: Commit and Rollback Processing*).

PREVIOUS | HOME | NEXT

18.1 PLVcase: Converting the Case of PL/SQL Programs | BOOK INDEX | V. Plug−and−Play Packages

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8*i* Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

19. PLVdyn and PLVfk: Dynamic SQL and PL/SQL

Contents:

This chapter describes several packages that provide a programmatic interface to the builtin DBMS_SQL package. These packages are the first in the "plug−and−play" category. Before plunging into a description of the specifics of the packages, I want to explain what I mean here by "plug−and−play" in PL/SQL code.

19.1 About Plug−and−Play

Plug−and−play packages allow you to replace whole sections of code with programs from PL/Vision packages. You essentially "plug−in" PL/SQL code and immediately gain benefits in your application. The best example of a PL/Vision plug−and−play component is the PLVexc (PL/Vision EXCeption handling) package. It provides very high−level exception−handling programs so that individual developers can simply call a procedure that describes the desired action, such as "record and continue," and PLVexc figures out what to do. It makes use of PLVlog to automatically write errors to the log of your choice (database table, PL/SQL table, etc.).

To give you a sense of plug−and−play in PL/SQL code, consider the following exception section. It has two different exception handlers: one for NO_DATA_FOUND and one for all other exceptions. When NO_DATA_FOUND is raised, I request that PLVexc display a message to the user, record the error, and then stop the program. When any other error occurs, I request that PLVexc record the error and then continue processing.

```
EXCEPTION
   WHEN NO_DATA_FOUND
   THEN
      PLVexc.recNstop ('Company has not been located.');
   WHEN OTHERS
   THEN
      PLVexc.recNgo;
END;
```

So PLVexc does a lot of work for me. What's new about that? You build a module encapsulating multiple actions and then use it over and over again. That's the central concept of modularization and black−boxing of logic. Why give it a fancy name like "plug−and−play"? Maybe it's just a difference of semantics. But maybe there's more to it.

The package structure of PL/SQL offers new opportunities when it comes to modularizing code. You can think of a package as nothing more than a list of programs, a convenient way to collect together related modules. With this perspective, you will not break new ground with packages. If, on the other hand, you look upon the package as a self−contained environment or product or object, with its own internal data structures, its own set of rules, you will find that you can construct a whole −− the package −− that is considerably greater than the sum of its parts (the individual elements defined in the package).

The PLVexc package certainly hides a lot of implementational complexity from its users. The real power of PLVexc is, however, reflected not so much in what or how much it hides. Rather, its strength resides more in what it lets you accomplish in your own programs −− and how you go about doing it.

For detailed information about the PLVexc package, see Chapter 22, *Exception Handling*.

PREVIOUS | HOME | NEXT
V. Plug−and−Play Packages | BOOK INDEX | 19.2 Declarative Programming in PL/SQL

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8*i* Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

PREVIOUS
Chapter 19
PLVdyn and PLVfk:
Dynamic SQL and PL/SQL
NEXT

19.2 Declarative Programming in PL/SQL

When you call a PLVexc exception handler, you shift from a procedural to a declarative style of programming in PL/SQL. To understand what I mean, think about the SQL language. You do not write programs in SQL. Instead, you use a declarative syntax to describe the set of data you want to see or modify. The underlying SQL engine then figures out the best way to accomplish the task. This declarative or "set at a time" mode of processing is a critical element of the power of the SQL language.

PL/SQL is very different from SQL. It is a procedural programming language. When you want to get something done in PL/SQL, you write a program. This gives you a tremendous amount of control, but it definitely cuts into your productivity −− and offers many over−ripe opportunities for introducing bugs into your code. Wouldn't it be wonderful to combine the control of the procedural language with the high−level abstraction and productivity of a declarative syntax? Well, we are not talking idle chatter here. We are talking packages.

Constructed properly, the PL/SQL package offers the opportunity for developers to write code in a declarative style. Take a look again at the exception section that utilizes PLVexc. To write this code, a user of PLVexc only had to know what kind of action she wanted to perform. Ignore the error? Record and continue? Record and halt? Just describe the action desired, pass the necessary data, and then let the underlying engine (the PLVexc package) figure out how to implement your request. Declarative coding in PL/SQL! Keep these concepts in mind as you read the chapters in Part V.

PREVIOUS
19.1 About Plug−and−Play
HOME
BOOK INDEX
NEXT
19.3 The Dynamic Packages of PL/Vision

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

19.3 The Dynamic Packages of PL/Vision

The builtin DBMS_SQL package supports all four methods of dynamic SQL, as well as dynamic PL/SQL code execution. It is a very powerful and useful tool that can in many ways transform the way in which you build applications, especially generic, reusable utilities, with PL/SQL.

There is, however, a tiny, little problem with DBMS_SQL: it is just too darn complicated. It consists of more than a dozen procedures and functions. These programs need to be used in a very particular sequence. Furthermore, depending on the method of dynamic SQL you wish to implement, you will use different combinations of those builtins. Finally, it just comes down to an awful lot of typing and know−how, even if you want to do something relatively simple. (See the sidebar for a general description of the flow of program calls for dynamic SQL. See Chapter 15, *PLVvu: Viewing Source Code and Compile Errors*, for a more complete description of DBMS_SQL.)

The result of this complexity is that relatively few developers take full advantage of all that DBMS_SQL has to offer. And since many of the actions required for dynamic SQL are the same regardless of the SQL statement, those individuals will be writing the same code over again.

What is wrong with this picture? Code redundancy is a maintenance nightmare. Requiring all developers to know the picayune details of technology like dynamic SQL is a productivity nightmare. Getting all of these versions of dynamic SQL to work is a code quality nightmare. Hey! Working with PL/SQL should not resemble a Freddy Krueger sequel. There's got to be something we can do here.

A DBMS_SQL Recap

The builtin DBMS_SQL package allows you to dynamically construct and execute SQL and PL/SQL statements. You get full programmatic control −− and with it comes full *responsibility*. With DBMS_SQL, nothing is taken for granted. You must specify each and every operation on the SQL statement, usually with a wide variety of procedure calls, from the SQL statement itself down to the values of bind variables and the data types of columns in SELECT statements.

To execute dynamic SQL (and PL/SQL) with the DBMS_SQL you must follow this general flow:

Open a cursor. When you open a cursor, you ask the RDBMs to set aside and maintain a valid cursor structure for your use with future DBMS_SQL calls. The RDBMs returns an INTEGER "handle" to this cursor. You will use this handle in all future calls to DBMS_SQL modules for this dynamic SQL statement. Note that this cursor is completely distinct from static PL/SQL cursors (whether implicit or explicit).

Parse the SQL statement. Before you can specify bind variable values and column structures for the SQL statement, it must be parsed by the RDBMs. This parse phase verifies that the SQL statement is properly constructed. It then associates the SQL statement with your cursor handle. Note that when you parse a DDL statement it is also executed immediately. Upon successful completion of the DDL parse, the RDBMs also issues an implicit commit. This behavior is consistent with that of SQL*Plus.

Bind all host variables. If the SQL statement contains references to host PL/SQL variables, you include placeholders to those variables in the SQL statement by prefacing their names with a colon, as in **`:salary`**. You must then bind the actual value for that variable into the SQL statement.

Define the columns in SELECT statements. Each column in the list of the SELECT must be defined. This define phase sets up a correspondence between the expressions in the list of the SQL statement and local PL/SQL variables that receive the values when a row is fetched (see COLUMN_VALUE). This step is necessary only for SELECT statements and is roughly equivalent to the INTO clause of an implicit SELECT statement in PL/SQL.

Execute the SQL statement. Execute the specified cursor, that is, its associated SQL statement. If the SQL statement is an INSERT, UPDATE, or DELETE, the EXECUTE command returns the numbers of rows processed. Otherwise you should ignore that return value.

Fetch rows from the dynamic SQL query. If you execute a SQL statement, you must then fetch the rows from the cursor, as you would with a PL/SQL cursor. When you fetch, however, you do not fetch directly into local PL/SQL variables.

Retrieve values from the execution of the dynamic SQL. If the SQL statement is a query, you will retrieve values from the SELECT expression list using COLUMN_VALUE. If you have executed a PL/SQL block, you will use VARIABLE_VALUE to retrieve any bind variables included in the PL/SQL code.

Close the cursor. Just as with normal PL/SQL cursors, you should always clean up by closing the cursor when you are done. This releases the memory associated with the cursor.

The answer is simple, at least in concept: build a package. And that is what I did. In fact, I built three packages for dynamic SQL that make it easier to use the builtin DBMS_SQL package:

PLVdyn
: Gives a thorough layer of code built around the DBMS_SQL builtin package.

PLVdyn1
: Supports single bind variable dynamic SQL.

PLVfk
: Offers a generic utility to perform foreign key lookups for any table.

PLVdyn and PLVfk are covered in this chapter; PLVdyn1, which works in similar fashion to PLVdyn, is described on the companion disk.

PREVIOUS | HOME | NEXT
19.2 Declarative Programming in PL/SQL | BOOK INDEX | 19.4 PLVdyn: A Code Layer over DBMS_SQL

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

19.4 PLVdyn: A Code Layer over DBMS_SQL

The PLVdyn (PL/Vision DYNamic SQL) package offers an easy−to−use, programmatic interface, or API, to the DBMS_SQL builtin package. It combines many basic operations in DBMS_SQL into functions and procedures, freeing up developers from having to know about many of the intricacies of using the builtin package.

Operations include:

- Execute a Data Definition Language (DDL) statement with one line of code
- Execute a PL/SQL block with one line of code
- Delete from or truncate the specified table
- Drop one or more objects from the data dictionary
- Execute an INSERT...SELECT FROM statement with an absolute minimum of SQL coding
- View the contents of the specified table
- Toggle on/off tracing that displays the dynamic SQL statement being parsed

The following sections show how to use each of the different elements of the PLVdyn package.

19.4.1 DDL Operations

The PLVdyn package offers several programmatic interfaces to DDL or Data Definition Language commands. Before DBMS_SQL was available, you could not execute any DDL statements (such as CREATE TABLE, CREATE INDEX, etc.) within PL/SQL. Now DBMS_SQL lets you execute anything you want −− as long as you have the appropriate authority.

DDL statements in SQL are handled differently from DML (Data Manipulation Language) statements such as UPDATE, INSERT, and so on. There is no need to execute a DDL statement. The simple act of parsing DDL automatically executes it and performs a COMMIT. This is true in SQL*Plus and it is true in PL/SQL programs. If you execute a DDL statement in your PL/SQL program, it commits all outstanding changes in your session. This may or may not be acceptable, so factor it into your use of the following PLVdyn programs.

> ***NOTE:*** To dynamically execute DDL from within PL/Vision, the account in which PLVdyn is installed must have the appropriate privilege granted to it explicitly. If you rely solely on role−based privileges, you receive an **`ORA−01031 error: insufficient privileges`**. For example, if you want to create tables from within PLVdyn, you need to have CREATE TABLE privilege granted to the PLVdyn account.

19.4.1.1 Generic DDL interface

First of all, PLVdyn offers a single, completely generic procedure to execute a DDL statement. Its header follows:

```
PROCEDURE ddl (string_in IN VARCHAR2);
```

You pass the string to **`PLVdyn.ddl`** and it takes care of the details (the implementation of which is discussed in the section called "Bundling Common Operations").

I can use the **`ddl`** procedure to perform any kind of DDL, as the following examples illustrate.

1. Create an index on the **`emp`** table.

   ```
   PLVdyn.ddl ('CREATE INDEX empsal ON emp (sal)');
   ```

2. Create or replace a procedure called **`temp`**.

   ```
   PLVdyn.ddl
      ('CREATE OR REPLACE PROCEDURE temp ' ||
       'IS BEGIN NULL; END;');
   ```

PLVdyn offers a number of other, more specialized programs to execute various kinds of DDL. These are described in the following sections.

19.4.1.2 Dropping and truncating objects with PLVdyn

PLVdyn offers two separate "cleanup" programs: **`drop_object`** and **`truncate`**. The **`drop_object`** procedure provides a powerful, flexible interface for dropping one or many objects in your schema. The header for **`drop_object`** is:

```
PROCEDURE drop_object
   (type_in IN VARCHAR2,
    name_in IN VARCHAR2,
    schema_in IN VARCHAR2 := USER);
```

The **`truncate`** command truncates either tables or clusters and has the same interface as **`drop_object`**:

```
PROCEDURE truncate
   (type_in IN VARCHAR2,
    name_in IN VARCHAR2,
    schema_in IN VARCHAR2 := USER);
```

The rest of this section describes the behavior and flexibility of **`drop_object`**. The same information applies to **`truncate`**.

You provide the type of object to drop, the name of the object, and the schema, (if you do want to drop objects in another schema). So instead of typing statements like this in SQL*Plus:

```
SQL> drop table emp;
```

you can now use PLVdyn and enter this instead:

```
SQL> exec PLVdyn.drop_object ('table', 'emp');
```

If I were an aggressive and desperate salesperson, I would try to convince you that my way (with PLVdyn) is better than the "native" DROP TABLE statement. That is, however, totally foolish. This is a case where the simple DDL statement is much more straightforward. So why did I bother writing the **`drop_object`** procedure? Because when I wrote it, I didn't plan simply to match the capabilities of a single DDL statement. Instead, I examined the way in which DBAs often need to drop and manipulate objects in a schema and I discovered a way to leverage PL/SQL to provide added value when it came to dropping objects.

19.4.1.3 Adding value with PL/SQL

In many of the accounts in which I have worked, an application abbreviation is prefixed onto the names of all objects (tables, views, programs, synonyms, etc.) for the application. The inventory system would use the **`INV`** abbreviation, for instance, to segregate by name all related objects. The main inventory table is named **`INV_master`**, the line items table named **`INV_item`**, and the package to maintain the invoices named **`INV_maint`**.

A common action taken in such environments is to drop all objects for a particular application or to clean out all of the stored procedures or tables or views. I may want to clear the inventory application from my test schema so that I can move the next release over from development. Without dynamic SQL in PL/SQL, you would have to use SQL to generate SQL, reading rows from the USER_OBJECTS table to create a series of DROP statements, and then execute those statements in SQL*Plus.

With DBMS_SQL and the PLVdyn package, you no longer have to take such a convoluted path to get the job done. Both the type and name arguments of the **`drop_object`** procedure can be wildcarded, and this gives you a tremendous amount of flexibility. I can drop all the objects with the inventory prefix as follows:

```
SQL> exec PLVdyn.drop_object ('%', 'INV%');
```

I can also request that all tables in the SCOTT schema be dropped with this command (it will, of course, only work if the owner of PLVdyn has the authority to drop objects in the SCOTT schema):

```
SQL> exec PLVdyn.drop_object ('table', '%', 'scott');
```

You can provide the same kinds of wildcarded arguments to **`truncate`**. If you specify "%" for the object type, for instance, **`truncate`** automatically applies the truncate command only to objects of type TABLE or CLUSTER.

19.4.1.4 Implementing multiobject actions

The implementation of **`drop_object`** and **`truncate`** is interesting; both of these programs simply call a private module called **`multiobj`**, which stands for "multiple objects." This procedure applies the specified command to all the objects in the ALL_OBJECTS view that match the type and name provided.

The **`multiobj`** procedure itself is a combination of static and dynamic SQL, as shown below:

```
PROCEDURE multiobj
   (action_in IN VARCHAR2,
    type_in IN VARCHAR2,
    name_in IN VARCHAR2,
    schema_in IN VARCHAR2 := USER)
IS
   /* The static cursor retrieving all matching objects */
   CURSOR obj_cur IS
      SELECT object_name, object_type
```

```
           FROM all_objects
          WHERE object_name LIKE UPPER (name_in)
            AND object_type LIKE UPPER (type_in)
               AND (UPPER (action_in) != c_truncate OR
                 (UPPER (action_in) = c_truncate AND
                  object_type IN ('TABLE', 'CLUSTER')))
            AND owner = UPPER (schema_in);

   BEGIN
      /* For each matching object ... */
      FOR obj_rec IN obj_cur
      LOOP
         /* Open and parse the drop statement. */
         ddl
            (action_in || ' ' ||
             obj_rec.object_type || ' ' ||
             UPPER (schema_in) || '.' ||
             obj_rec.object_name);
      END LOOP;
   END;
```

The first argument to **`multiobj`** is the action desired. The rest of the arguments specify one or more objects upon which the action is to be applied.

The static cursor fetches all records from the ALL_OBJECTS data dictionary view that match the criteria. The dynamic cursor is defined and executed inside the generic **`PLVdyn.ddl`** procedure.

Why, my readers may be asking, did I not use the PLVobj package to fetch from the ALL_OBJECTS view? The whole point of that package was to allow me to avoid making direct references to that view; instead I could rely on a programmatic interface and very high−level operators. Using **`PLVobj.loopexec`**, I could theoretically implement **`multiobj`** with a single statement, in which I execute **`PLVdyn.ddl`** as a dynamically constructed PL/SQL program.

Believe me, I really *wanted* to use PLVobj in this program. The problem I encountered is that PLVobj is not sufficiently flexible. As you can see in the declaration section of **`multiobj`**, my cursor against ALL_OBJECTS is somewhat specialized. I have very particular logic inserted that automatically filters out objects that are not appropriate for TRUNCATE operations. There was no way for me to incorporate this logic into PLVobj as it currently exists. I do plan, however, that a future version of PLVobj will utilize dynamic SQL and then allow me to modify the WHERE clause (and even which view it works against: USER_OBJECTS, ALL_OBJECTS, or DBA_OBJECTS).

In the meantime, I write my own, somewhat redundant cursor against ALL_OBJECTS and then construct the appropriate DDL statement based on the incoming action and values from the row fetched from ALL_OBJECTS.

19.4.1.5 Generating sequence numbers

The **`nextseq`** function returns the *n*th next value from the specified Oracle sequence. Its header is as follows:

```
   FUNCTION nextseq
      (seq_in IN VARCHAR2, increment_in IN INTEGER := 1)
      RETURN INTEGER;
```

The default value for the increment is 1, so by default you get the immediate next value from the sequence with a call like this:

```
   :emp.empno := PLVdyn.nextseq ('emp_seq');
```

You can also use **nextseq** to move your sequence forward by an arbitrary number of values. This is often necessary when the sequence has somehow gotten out of sync with the data. To move the **emp_seq** sequence ahead by 1000 values, simply execute this statement:

```
SQL> exec PLVdyn.nextseq.('emp_seq', 1000);
```

Why did I bother building **nextseq** ? The Oracle database offers a very powerful method for generating unique sequential numbers: the sequence generator. This database object guarantees uniqueness of values and comes in very handy when you need to get the next primary key value for INSERTs. One drawback of the sequence generator is that you can only obtain the next value in the sequence by referencing the sequence object from within a SQL statement.

The following SELECT statement, for example, is like the one often used within PRE−INSERT triggers in Oracle Forms applications:

```
SELECT emp_seq.NEXTVAL INTO :emp.empno FROM dual;
```

In other words, you must make this artificial query from the **dual** table to obtain the sequence number to then use inside the Oracle Forms application.

And suppose that you want to move the sequence forward by 1000 (this kind of requirement arises when, for one reason or another, the sequence has gotten out of sync with the table's primary key). You would then have to write and execute a FOR along these lines:

```
DECLARE
   dummy INTEGER;
BEGIN
   FOR val IN 1 .. 1000
   LOOP
      SELECT emp_seq.NEXTVAL INTO dummy FROM dual;
   END LOOP;
END;
/
```

I don't know about you, but the year is 1996 and I just don't think I should have to execute queries against **dual** to get things done. That makes me feel like a dummy! I also believe that Oracle Corporation will come to its senses eventually and allow you to obtain sequence numbers without going to the SQL layer. In the meantime, however, PLVdyn offers a programmatic interface to any sequence so that you can at least hide the fact that you are using **dual** to get your sequence number. You even get to hide the specific syntax of sequence generation (the NEXTVAL keyword, for instance), which will make it easier for anyone from novices to developers to utilize this technology.

19.4.1.6 What about the overhead?

Of course, since you are executing dynamic SQL, it takes more time to generate the sequence with **nextseq** compared with a static generation. My tests showed that, on average, the static generation approach (using a direct call to the sequence NEXTVAL through a dummy SQL query) took .55 seconds for 100 increments of the sequence. The dynamic approach using **PLVdyn.nextseq**, on the other hand, required 1.54 seconds for 100 increments of the sequence. There are two ways of looking at these results:

1. *Cup half empty:* The dynamic sequence generation is three times slower than static! How awful!

2. *Cup half full:* Dynamic generation of a single sequence value took only .0154 seconds on average. Unless I am working in a high−transaction, subsecond kind of environment, this is a totally acceptable level of performance.

Is your cup half empty or half full? It depends on your specific application situation. As a general rule, you should carefully evaluate your use of dynamic SQL to make sure that you can afford the overhead. When your application can absorb the extra CPU cycles (and your users can tolerate the difference in response time), a program like **PLVdyn.nextseq** offers many advantages.

19.4.1.7 Compiling source code with PLVdyn

When you CREATE OR REPLACE a PL/SQL program unit in SQL*Plus, you are executing a DDL command. You can, consequently, issue that same command using DBMS_SQL −− the difference is that you can construct, create, and compile the PL/SQL program dynamically. PLVdyn offers two versions of the **compile** procedure precisely to allow you to take this action. The headers of the **compile** programs are as shown:

```
PROCEDURE compile
   (stg_in IN VARCHAR2,
    show_err_in IN VARCHAR2 := PLV.noshow);
PROCEDURE compile
   (table_in IN PLVtab.vc2000_table,
    lines_in IN INTEGER,
    show_err_in IN VARCHAR2 := PLV.noshow);
```

The second version of **compile** assumes that the program definition is in a PL/SQL table in which each row starting from 1 and going to **lines_in** rows contains a sequential line of code. This procedure simply concatenates all of the lines together and passes them to the first version of **compile**.

The string version of **cor** takes two arguments: **stg_in**, a string of up to 32,767 characters that contains the definition of the program, and **show_err_in**, which indicates whether you want to call **PLVvu.err** after compilation to check for compile errors. The program definition should start with the program unit type (PROCEDURE, FUNCTION, PACKAGE, or PACKAGE BODY). Do not append a CREATE OR REPLACE to the string; **compile** does this automatically. You also should not include a final / after the END; statement. This syntax is necessary only when executing the CREATE OR REPLACE directly in SQL*Plus. You can, on the other hand, include newline characters in your string.

Here is an example of using **PLVdyn.compile** to create a very simple procedure with an error. I also request that PLVdyn show me the compile errors.

```
SQL> exec PLVdyn.compile('procedure temp is begin nul; end;',PLV.show);
--------------------------------------------------------------------
PL/Vision Error Listing for PROCEDURE TEMP
--------------------------------------------------------------------
Line#  Source
--------------------------------------------------------------------
    1 procedure temp is begin nul; end;
ERR                            *
    PLS−00313: 'NUL' not declared in this scope
```

How would you use **PLVdyn.compile**? You might try building yourself a Windows−based frontend for PL/SQL development. For example, find a really good programming editor that has a macro language and the ability to execute dynamic link libraries (DLLs). Create a DLL that accepts a string and executes **PLVdyn.compile**. Tie this in to the editor and you are on your way to dramatically improving your PL/SQL development environment.

> ***NOTE:*** To dynamically compile and create stored PL/SQL code from within PL/Vision, the account in which PLVdyn is installed must be directly granted CREATE PROCEDURE authority. If you rely solely on role−based privileges, you receive an **ORA−01031 error: insufficient privileges**.

You might also use **PLVdyn.compile** to temporarily create program units for use in your application, and then drop them at the end of your session. You could, for example, create a package dynamically that would define some global data structures whose names are established dynamically. They could only be referenced through dynamic PL/SQL as well, but PLVdyn does make this possible.

19.4.1.8 Implementing the compile procedure

The implementation of the string−based **compile** procedure is a good example of how I am able to leverage existing elements of PL/Vision to easily extend the functionality of my library. The body of **compile** is shown below:

```
PROCEDURE compile
   (stg_in IN VARCHAR2,
    show_err_in IN VARCHAR2 := PLV.noshow)
IS
   v_name1 PLV.plsql_identifier%TYPE;
   v_name2 PLV.plsql_identifier%TYPE;
BEGIN
   ddl ('CREATE OR REPLACE ' || stg_in);
   IF show_err_in = PLV.show
   THEN
      v_name1 := PLVprs.nth_atomic (stg_in, 1, PLVprs.c_word);
      v_name2 := PLVprs.nth_atomic (stg_in, 2, PLVprs.c_word);
      IF UPPER(v_name1||' '||v_name2) = 'PACKAGE BODY'
      THEN
         v_name1 := v_name1 || ' ' || v_name2;
         v_name2 :=
            PLVprs.nth_atomic (stg_in, 3, PLVprs.c_word);
      END IF;
      PLVvu.err (v_name1 || ':' || v_name2);
   END IF;
END;
```

Notice, first of all, that it relies on the **ddl** procedure to execute the program. Then **compile** checks to see if you wanted to view compile errors. If so, it uses the **nth_atomic** function of PLVprs to extract the first two words from the program definition. If a PACKAGE BODY, it then retrieves the third word, which is the name of the package. Finally, it calls **PLVvu.err** to display any errors.

PREVIOUS | HOME | NEXT
19.3 The Dynamic Packages of PL/Vision | BOOK INDEX | 19.5 DML Operations

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

19.5 DML Operations

Although it is possible to execute DML statements with static strings in PL/SQL, the dynamic nature of DBMS_SQL does offer many new opportunities. The PLVdyn packages offers a programmatic interface for several common DML operations to make it easier for you to take advantage of this technology.

PLVdyn offers three different DML operations: **`dml_insert_select`**, **`dml_delete`**, and **`dml_update`**. There are many other possibilities for dynamic DML in PLVdyn; I encourage you to experiment with your own extensions to this package.

> ***NOTE:*** You cannot include bind variables for any of the DML programs in PLVdyn. Generally, PLVdyn does not support bind variables. The consequence of this is that the WHERE clauses may not contain any colons unless they are embedded in a literal string. The PLVdyn1 package (described on the companion disk) does allow you to specify single bind variable values for a variety of DML statements.

19.5.1 Performing an INSERT−SELECT FROM

You can execute an INSERT−SELECT FROM statement using the **`dml_insert_select`**. The header for this procedure is:

```
PROCEDURE dml_insert_select
   (table_in IN VARCHAR2, select_in IN VARCHAR2);
```

In the first argument you provide the name of the table that is going to receive the rows from the SELECT statement. The second argument is the SELECT statement itself. Here is an example of using this procedure to copy current invoices to a history table.

```
PLVdyn.dml_insert_select
   ('inv_history',
    'SELECT * FROM inv_current' ||
    ' WHERE inv_date < ' || TO_CHAR (v_enddate));
```

19.5.2 Executing dynamic DELETEs

The **`dml_delete`** procedure allows you to delete from any table. The header is:

```
PROCEDURE dml_delete
   (table_in IN VARCHAR2, where_in IN VARCHAR2 := NULL);
```

The first argument is the table from which rows are to be deleted. The second argument is an optional WHERE clause, which restricts the number of rows deleted. As with static SQL, the simplest form of a call to **`dml_delete`** (a NULL WHERE clause) results in the largest number of rows deleted.

The following call to **`dml_delete`** removes all employees in department 10 from the **`emp`** table:

```
PLVdyn.dml_delete ('emp', 'deptno = 10');
```

Clearly, this syntax offers very little in the way of productivity enhancement over simply executing this SQL statement:

```
SQL> delete from emp where deptno=10;
```

The big news about PLV**dyn.dml_delete** is that you can execute it from within a PL/SQL environment.

19.5.3 Executing dynamic UPDATEs

The **dml_update** procedure is overloaded to allow you to easily perform updates of single columns of date, number or string datatypes. Here is the overloaded header for **dml_update**:

```
PROCEDURE dml_update
   (table_in IN VARCHAR2,
    column_in IN VARCHAR2,
    value_in IN VARCHAR2|DATE|NUMBER,
    where_in IN VARCHAR2 := NULL);
```

The only optional argument is the WHERE clause. If you do not supply a WHERE clause, the requested UPDATE will be performed for *all* rows in the table.

The following examples demonstrate the different ways to use **dml_update**.

1. Set the salary of all employees to $100,000.

   ```
   PLVdyn.dml_update ('emp', 'sal', 100000);
   ```

2. Set the last updated time stamp to SYSDATE for all invoices with creation dates in the last three months.

   ```
   PLVdyn.dml_update
      ('invoice', 'update_ts', SYSDATE,
       'create_ts >= ADD_MONTHS(SYSDATE,-3)');
   ```

19.5.4 Checking feedback on DML

Each of the PLVdyn DML programs calls the **PLVdyn.execute** procedure, which, in turn, calls the DBMS_SQL.EXECUTE builtin to actually execute the SQL statement. DBMS_SQL.EXECUTE is a function that returns an integer, telling you the number of rows affected by the SQL statement.

PLVdyn saves and hides this feedback value to make it easier to execute your SQL. You can, however, check the result of your latest dynamic execution by calling the **dml_result** function. In the example following, I delete rows from the **emp** table and then check to see how many rows were actually deleted.

```
BEGIN
   PLVdyn.dml_delete ('emp');
   IF PLVdyn.dml_result > 100
   THEN
      /*
      || Too many deletes. There is some kind
      || of data problem. Issue a rollback.
      */
      PLVrb.do_rollback;
   END IF;
```

19.5.5 Executing Dynamic PL/SQL Code

PLVdyn makes it easy for you to execute dynamically constructed PL/SQL statements with the **`plsql`** procedure. The header for this program is:

```
PROCEDURE plsql (string_in IN VARCHAR2);
```

You construct a PL/SQL statement or block of statements and then pass that string to **`PLVdyn.plsql`**. It then executes that code. By using **`PLVdyn.plsql`**, you do not have to code the separate open, parse, and execute steps that are required with dynamic PL/SQL. In addition, this procedure relieves you of the burden of dealing with the following rules about dynamic PL/SQL:

1. The statement must end in a semicolon.
2. The statement must be a valid PL/SQL block. It must begin, in other words, with either the DECLARE or BEGIN keywords, and end with the END; statement.

You can provide to **`PLVdyn.plsql`** a string that meets these requirements, or you can ignore those requirements and **`plsql`** takes care of things for you. As a result, any of the following calls to **`PLVdyn.plsql`** properly executes the **`calc_totals`** procedure:

```
SQL> exec PLVdyn.plsql ('calc_totals');
SQL> exec PLVdyn.plsql ('calc_totals;');
SQL> exec PLVdyn.plsql ('BEGIN calc_totals; END;');
```

The **`plsql`** procedure accomplishes this by stripping off any trailing semicolons and wrapping your string inside a BEGIN−END block, regardless of whether or not you have already done so.

19.5.5.1 Implementation of PLVdyn.plsql

The implementation of **`plsql`**, shown below, is a good example of how you can build additional smarts into your software so that it is easier and more intuitive to use. It also demonstrates how I combine other, bundled operations to construct higher−level programs (these bundled operations are explored in a later section).

```
PROCEDURE plsql (string_in IN VARCHAR2)
IS
   cur INTEGER :=
      open_and_parse (plsql_block (string_in));
BEGIN
   execute_and_close (cur);
END;
```

The **`plsql_block`** function is designed specifically to return a valid PL/SQL block as follows:

```
FUNCTION plsql_block (string_in IN VARCHAR2) RETURN VARCHAR2
IS
BEGIN
   RETURN 'BEGIN ' || RTRIM (string_in, ';') || '; END;';
END;
```

> ***NOTE:*** The **`plsql`** procedure assumes that you do not have any bind variables. If a colon appears in the PL/SQL code and it is not within single quotes (part, that is, of a literal string), **`PLVdyn.plsql`** will not be able to execute your statement successfully.

19.5.5.2 Scope of a dynamic PL/SQL block

Precisely what kind of PL/SQL code can you execute with DBMS_SQL? The answer is not as simple as it might seem at first glance. The rule you must follow is this:

> *A dynamically−constructed PL/SQL block can only execute programs and reference data structures which are defined in the specification of a package.*

Consider, for example, the following simple script:

```
<<dynamic>>
DECLARE
   n NUMBER;
BEGIN
   PLVdyn.plsql ('n := 5');
END;
/
```

All I am doing is assigning a value of 5 to the local variable n. This string is executed within its own BEGIN−END block, that would appear to be a nested block within the anonymous block named "dynamic" with the label. Yet when I execute this script I receive the following error:

```
PLS−00302: component 'N' must be declared
ORA−06512: at "SYS.DBMS_SYS_SQL", line 239
```

The PL/SQL engine is unable to resolve the reference to the variable named n. I get the same error even if I qualify the variable name with its block name:

```
<<dynamic>>
DECLARE
   n NUMBER;
BEGIN
   /* Also causes a PLS−00302 error! */
   PLVdyn.plsql ('dynamic.n := 5');
END;
/
```

Yet if instead of modifying the value of n, I modify the **`PLV.plsql_identifier`** variable as shown below, I am able to execute the dynamic assignment successfully.

```
<<dynamic>>
DECLARE
   n NUMBER;
BEGIN
   PLVdyn.plsql ('PLV.plsql_identifier := ''5''');
END;
/
```

What's the difference between these two pieces of data? The variable n is defined locally in the anonymous PL/SQL block. The **`plsql_identifier`** variable is a public global defined in the PLV package. This distinction makes all the difference with dynamic PL/SQL.

It turns out that a dynamically constructed and executed PL/SQL block is not treated as a *nested* block. Instead, it is handled like a procedure or function called from within the current block. So any variables local to the current or enclosing blocks are not recognized in the dynamic PL/SQL block. You can only make references to globally defined programs and data structures. These PL/SQL elements include standalone functions and procedures and any element defined in the specification of a package. That is why my reference to the **`plsql_identifier`** variable of the PLV package passed without error. It is defined in the package specification and is globally available.

Fortunately, the dynamic block is executed within the context of the calling block. If you have an exception section within the calling block, it traps exceptions raised in the dynamic block.

19.5.5.3 The overhead of dynamic PL/SQL

As soon as you move to dynamic processing, you can expect that there will be some overhead associated with the extra work performed (construct the string, parse the string, and −− with dynamic PL/SQL −− compile the anonymous block). Dynamic PL/SQL is really neat stuff −− but is it practical? What is the performance penalty when executing, for example, the **PLVdyn.plsql** procedure?

I came up with an answer by using the PLVtmr package (see Chapter 14, *PLVtmr: Analyzing Program Performance*). The script below first determines how much time it takes to do, well, nothing with a call to the NULL statement. This provides a baseline for static code execution (and you can't get much more base than that). I then execute the NULL statement dynamically, with a call to the **PLVdyn.plsql** procedure. The single substitution parameter, **&1**, allows me to specify the number of iterations for the test.

```
BEGIN
   PLVtmr.capture;
   FOR rep IN 1 .. &1
   LOOP
      NULL;
   END LOOP;
   PLVtmr.show_elapsed ('static');

   PLVtmr.capture;
   FOR rep IN 1 .. &1
   LOOP
      PLVdyn.plsql ('NULL');
   END LOOP;
   PLVtmr.show_elapsed ('dynamic');
END;
/
```

I then executed this script in SQL*Plus several times to make sure my results were consistent:

```
SQL> @temp 100
static Elapsed: 0 seconds.
dynamic Elapsed: 1.38 seconds.
SQL> @temp 1000
static Elapsed: .11 seconds.
Dynamic Elapsed: 13.57 seconds.
SQL> set verify off
SQL> @temp 1000
static Elapsed: .16 seconds.
dynamic Elapsed: 13.41 seconds.
```

I conclude, therefore, that the overhead of constructing, compiling, and executing a dynamic block of PL/SQL code is at least (and approximately) .0133 seconds. I say "at least" because if your PL/SQL block is bigger (consisting of more than, say, a single statement or call to a stored procedure), your overhead increases. It looks to me that for most situations the additional processing time required for dynamic PL/SQL should not deter you from using this technique.

19.5.6 Bundled and Passthrough Elements

While PLVdyn does offer a number of useful, high−level operations to perform dynamic SQL, there is no way that I can build enough of these kinds of programs to handle all dynamic SQL needs. I can, however, still add value for a developer who has very unique requirements: I can bundle together common, often−repeated steps into single lines of code. The developer still has to write his or her own full−fledged dynamic SQL program, but can rely on these lower−level "prebuilts" (as opposed to "builtins") to improve productivity and code quality.

The PLVdyn package offers these bundled elements:

open_and_parse
Performs the open and parse phases.

execute_and_close
Executes and closes the cursor.

parse_delete
Constructs and parses a DELETE statement from its components.

PLVdyn adds value in one other way: with passthrough programs. These passthroughs do not (in normal operation) necessarily do anything but execute their corresponding DBMS_SQL program. Since they are a layer of code around the builtins, however, the passthroughs provide an opportunity to add functionality in other ways.

The passthroughs of PLVdyn are **execute** and **execute_and_fetch**; they are discussed after the bundled elements below.

19.5.6.1 Open and parse

The **open_and_parse** function calls DBMS_SQL.OPEN_CURSOR and DBMS_SQL.PARSE. These are both mandatory steps for *any* method of dynamic SQL you choose to implement with DBMS_SQL. So rather than have to call these two separate programs over and over again, you can simply call the single function. In addition, the function hides the need to pass the database version (in the form of a constant such DBMS_SQL.V7) when you call PARSE. The header for **open_and_parse** is:

```
FUNCTION open_and_parse
   (string_in IN VARCHAR2 SEF,
    mode_in IN INTEGER := DBMS_SQL.NATIVE)
RETURN INTEGER
```

With this bundled operation, you can open and parse a SQL statement with the following single line of code:

```
cur := PLVdyn.open_and_parse (string_in);
```

Without **open_and_parse**, you type the following lines of code:

```
cur := DBMS_SQL.OPEN_CURSOR;
DBMS_SQL.PARSE (cur, string_in, DBMS_SQL.V7);
```

PLVdyn programs themselves make use of **open_and_parse**. Any time you parse a SQL statement through PLVdyn, you are, in fact, executing **open_and_parse**. This bundled program therefore acts as a kind of gateway through which all dynamic SQL must pass. I took advantage of this fact when I constructed a trace or window for PLVdyn (covered later in this chapter).

19.5.6.2 Execute and close

In many (but not all) situations, you want to execute your cursor and then immediately close it. If you are executing an INSERT, UPDATE, or DELETE, for example, you do not have to fetch rows from the SQL statement. After executing, you are ready to close.

The header for **execute_and_close** is as follows:

```
PROCEDURE execute_and_close (cur_inout IN OUT INTEGER)
```

In other words, you pass in the cursor and PLVdyn does the rest for you. Using this bundled operation, you can execute and close with this single line:

```
PLVdyn.execute_and_close (cur);
```

Otherwise, you have to code these lines (and declare a variable to hold the feedback from the EXECUTE statement):

```
fdbk := DBMS_SQL.EXECUTE (cur);
DBMS_SQL.CLOSE (cur);
```

With PLVdyn, you don't have to deal with the EXECUTE feedback unless you want to. If you need to know the results of the operation, you can access it through the **dml_result** or **fdbk** functions. These programs return the feedback from the most recent execution of **execute_and_close**.

By calling PLV**dyn.execute_and_close** instead of DBMS_SQL.EXECUTE, you can also take advantage of the execute toggle in PLVdyn, which is covered below.

19.5.6.3 Parse for delete

The last bundled operation is **parse_delete**. It is a bit more complex than the other operations. It is used by the **dml_delete** procedure to construct the SQL DELETE and then parse it, as shown below:

```
PROCEDURE dml_delete
   (table_in IN VARCHAR2, where_in IN VARCHAR2 := NULL)
IS
   cur INTEGER;
BEGIN
   parse_delete (table_in, where_in, cur);
   execute_and_close (cur);
END;
```

You might ask why I built a separate **parse_delete** procedure for this single call in **dml_delete** (it is not used anywhere else in PLVdyn). That is a reasonable question to ask. After all, one should be careful not to over−modularize. You can argue that it never hurts to break out functionality into separate programs. Then it is at least potentially reusable. You could, however, spend (waste?) many hours constructing individual modules that are never actually reused. Find the right balance.

In the case of **parse_delete**, it only looks as if the procedure is used once. In fact, **parse_delete** is also used in a second dynamic SQL package, PLVdyn1 (the code for which is found at the end of this chapter). This package contains a series of overloaded delete programs that take a single bind variable as input. For example, I could delete all employees where the **hiredate** is more than 100 years in the past with this call:

```
PLVdyn1.dml_delete
  ('emp',
   ADD_MONTHS (SYSDATE, −1200),
   'hiredate < :indate');
```

So I do, in fact, need a consolidated **parse_delete** function to handle common operations across at least four different delete programs. Now that I have justified my development process to my readers, let's take a look at the specification for **parse_delete**:

```
PROCEDURE parse_delete
   (table_in IN VARCHAR2,
    where_in IN VARCHAR2,
    cur_out OUT INTEGER);
```

You pass a table and the WHERE clause and receive a cursor handle in return. You can then use this cursor handle in a call to **`execute_and_close`**. The implementation of **`parse_delete`** is shown below. It constructs the DELETE string based on the inputs and then calls the prebuilt **`open_and_parse`** (notice that it does not need the "PLVdyn" prefix on the call to **`open_and_parse`**, since it is coded *within* the package).

```
PROCEDURE parse_delete
  (table_in IN VARCHAR2,
   where_in IN VARCHAR2,
   cur_out OUT INTEGER)
IS
   delete_stg PLV.dbmax_varchar2%TYPE := 'DELETE FROM ' || table_in;
   cur INTEGER;
BEGIN
   IF where_in IS NOT NULL
   THEN
      delete_stg := delete_stg || ' WHERE ' || where_in;
   END IF;
   cur_out := open_and_parse (delete_stg);
END;
```

These low−level, bundled operations are very handy when constructing more complex, full−featured dynamic SQL programs. Consider the **`ddl`** procedure described earlier in Section 19.5, "DML Operations". The implementation of **`PLVdyn.ddl`** is shown below:

```
PROCEDURE ddl (string_in IN VARCHAR2)
IS
   cur INTEGER;
BEGIN
   IF NOT executing
   THEN
      p.l ('PLVdyn: No parse/execute of DDL.');
      display_dynamic_sql (string_in);
   ELSE
      cur := open_and_parse (string_in);
   END IF;
END;
```

The executing function and the **`display_dynamic_sql`** support the execution toggle and execution trace features of PLVdyn. Disregarding those lines for the time being, the entire body of the **`ddl`** procedure is simply:

```
cur := open_and_parse (string_in);
```

Since it is a DDL statement, the parse automatically executes *and* commits, so this is all the code that is needed. The bundled operations (**`open_and_parse`**, **`execute_and_close`**, and so on) are called throughout the body of the PLVdyn package to minimize code volume and maximize code reuse.

19.5.6.4 Passthrough elements

PLVdyn offers two execute−related programs that are nothing more than passthroughs to the corresponding DBMS_SQL element. The headers for these passthroughs are:

```
PROCEDURE execute (cur_in IN INTEGER);
PROCEDURE execute_and_fetch
   (cur_in IN INTEGER, match_in IN BOOLEAN := FALSE);
```

`PLVdyn.execute` calls the DBMS_SQL.EXECUTE builtin, while **`PLVdyn.execute_and_fetch`** calls the DBMS_SQL.EXECUTE_AND_FETCH builtin.

Why did I bother building these procedures and why should you bother to use them? There are two basic reasons:

Ease of use. The DBMS_SQL versions of these programs are functions. They return the number of rows affected by the SQL statement executed. This is important information, but it is not always needed. With PLVdyn passthroughs, you can simply call the procedure to perform the execute. If you then need to see the feedback value, you can call the **`PLVdyn.fdbk`** function.

2. *PLVdyn has placed a layer of code between your program and the native builtin function.* Once this layer is in place, PLVdyn can add functionality around the DBMS_SQL execute operation. The **`execsql`** toggle, for example, allows you to turn on or off the actual execution of your dynamic SQL −− without having to change your application's code. PLVdyn could also enhance the trace to show you when code is executed.

19.5.7 Displaying a Table

PLVdyn offers the **`disptab`** procedure to display the contents of any database table from within the PL/SQL environment. This procedure is both a useful utility (one that is used by PLVlog, as is shown below) and an excellent demonstration of how to build programs that support Method 4 dynamic SQL with the DBMS_SQL package.

The header of **`disptab`** follows:

```
PROCEDURE disptab
  (table_in IN VARCHAR2,
   string_length_in IN INTEGER := 20,
   where_in IN VARCHAR2 := NULL,
   date_format_in IN VARCHAR2 := PLV.datemask);
```

The four arguments to **`disptab`** are as follows:

`table_in`
: The name of the table to be displayed. This is the only required argument.

`string_length_in`
: The maximum number of characters to be used in displaying string columns. I do not even attempt to do the kind of string wrapping performed in SQL*Plus. Instead, I use SUBSTR to truncate the values.

`where_in`
: An optional WHERE clause to restrict the rows displayed. If not specified, all rows are retrieved. You can also use this argument to pass in ORDER BY and HAVING clauses, since they follow immediately after the WHERE clause.

`date_format_in`
: The format to be used to display dates. The default is the default date mask for PL/Vision, maintained in the PLV package.

Here are some examples of output from **`disptab`**:

1. Display the full contents of the employee table:

```
SQL> execute PLVdyn.disptab ('emp');

-----------------------------------------------------------------------
                           Contents of emp
-----------------------------------------------------------------------
EMPNO ENAME      JOB        MGR  HIREDATE        SAL     COMM    DEPTNO
```

```
-----------------------------------------------------------------------
7839  KING        PRESIDENT      11/17/81 120000 5000           10
7698  BLAKE       MANAGER   7839 05/01/81 120000 2850           30
7782  CLARK       MANAGER   7839 06/09/81 120000 2450           10
7566  JONES       MANAGER   7839 04/02/81 120000 2975           20
7654  MARTIN      SALESMAN  7698 09/28/81 120000 1250    1400    30
7499  ALLEN       SALESMAN  7698 02/20/81 120000 1600    300     30
7844  TURNER      SALESMAN  7698 09/08/81 120000 1500    0       30
7900  JAMES       CLERK     7698 12/03/81 120000 950            30
7521  WARD        SALESMAN  7698 02/22/81 120000 1250    500     30
7902  FORD        ANALYST   7566 12/03/81 120000 3000           20
7369  SMITH       CLERK     7902 12/17/80 120000 800            20
7788  SCOTT       ANALYST   7566 12/09/82 120000 3000           20
7876  ADAMS       CLERK     7788 01/12/83 120000 1100           20
7934  MILLER      CLERK     7782 01/23/82 120000 1300           10
```

2. Display only those rows of the **emp** table in department 10, ordering the rows by salary in increasing order. I also request that a maximum of 20 characters be used to display string information.

```
SQL> execute PLVdyn.disptab ('emp', 20, 'deptno = 10 order by sal');
-----------------------------------------------------------------------
                           Contents of emp
-----------------------------------------------------------------------
EMPNO ENAME      JOB       MGR  HIREDATE        SAL     COMM    DEPTNO
-----------------------------------------------------------------------
7934  MILLER      CLERK     7782 01/23/82 120000 1300           10
7782  CLARK       MANAGER   7839 06/09/81 120000 2450           10
7839  KING        PRESIDENT      11/17/81 120000 5000           10
```

3. I use the **disptab** procedure inside the PLVlog package to display the contents of the log table when the user has requested logging to a database table. The PLVlog display procedure is shown below. If the log type is set to database table, **disptab** is called. Otherwise the PLVtab package is used to display the contents of the PL/SQL table containing the log.

```
PROCEDURE display IS
BEGIN
   IF log_type = PLV.dbtab
   THEN
      /* Use generic table display program? */
      PLVdyn.disptab (log_struc.table_name, 30);
   ELSIF log_type = PLV.pstab

   THEN
      PLVtab.display
         (log_pstable,

          pstab_count,
          'PL/Vision Log');
   END IF;
END;
```

Notice that the user does not have to provide any information about the structure of the table. The **disptab** procedure gets that information itself −− and this is precisely the aspect of **disptab** that makes it a Method 4 dynamic SQL example and very useful.

19.5.8 Controlling Execution of Dynamic SQL

You can never have enough flexibility. That's my motto. I built this package to execute my dynamic SQL and then I actually encountered circumstances in which I wanted to run all of my code, but preferred to not actually *execute* the dynamic SQL. So I added yet another toggle to the PLVdyn package that would give me just this capability.

To turn on execution of dynamic SQL with PLVdyn, you call the **execsql** procedure. Here is an example of calling **execsql** from within a SQL*Plus session:

```
SQL> exec PLVdyn.execsql
```

To turn off execute of dynamic SQL in PLVdyn, call **noexecsql** as shown below:

```
SQL> exec PLVdyn.noexecsql
```

The executing function returns TRUE if PLVdyn is currently executing the SQL statements passed to it. You can call this function; PLVdyn calls the function itself to decide if it should call DBMS_SQL.EXECUTE, as you can see in the implementation of **PLVdyn.execute**:

```
PROCEDURE execute (cur_in IN INTEGER) IS
BEGIN
   IF executing
   THEN
      dml_feedback := DBMS_SQL.EXECUTE (cur_in);
   ELSE
      p.l ('PLVdyn: Execution disabled...');
   END IF;
END;
```

19.5.9 Tracing Use of Dynamic SQL

With PLVdyn, you construct a SQL statement and pass that string to a PLVdyn program for parsing or execution. It's a big time saver, but it also implies a loss of some control. You trust PLVdyn to do the right thing −− and it does. The question remains, however: what is *your* code passing to PLVdyn?

The code used to construct the dynamic SQL statement is often complicated (you can see this in some of my high−level operators). As I began to use PLVdyn, I often found that I wanted to see the SQL statement that PLVdyn was executing. I needed to verify that my calling program had put the SQL together properly.

There were two ways I could display my SQL statement:

1. Put calls to DBMS_OUTPUT *before* each of my calls to PLVdyn modules. In this way, I do not have to change PLVdyn. (It is not, after all, the fault of PLVdyn −− or its author! −− that I wasn't sure what my code was doing.)

2. Put the trace *inside* PLVdyn. This approach would require changes to the base package. On the other hand, it would be done once and then be available for all users of PLVdyn.

To me, the second approach was clearly preferable. If I were going to put this trace inside the package, I needed to make sure that the trace displayed only my SQL statement when I wanted to see it. When the package was used in a production environment, I couldn't have extraneous messages showing up on the user's screen.

I therefore implemented my trace with the following procedures and functions:

```
PROCEDURE showsql (start_with_in IN VARCHAR2 := NULL);
PROCEDURE noshowsql;
FUNCTION showing RETURN BOOLEAN;
```

The **showsql** procedure turns on the trace. If you execute **PLVdyn.noshowsql**, you turn off the trace. The **showing** function tells you the current state of the toggle.

When you turn on the PLVdyn trace, you can specify a "start with" string. If this string is not NULL, then that serves as the starting point for display of the dynamic SQL. You can use that option to skip over portions of SQL in which you have no interest. This feature was more useful and necessary in the early days of the PLVdyn trace when I had not incorporated the PLVprs string−wrapping functionality.

19.5.9.1 Using the dynamic SQL trace

In the following script, I turn on the trace and then execute a request to open and parse an INSERT statement.

```
SET SERVEROUTPUT ON
execute PLVdyn.showsql;

DECLARE
   cur INTEGER;
BEGIN
   cur := PLVdyn.open_and_parse
     ('insert into emp (empno, deptno) values (1505, 100)');
END;
/
```

I then receive this output:

```
Dynamic SQL: insert into emp (empno, deptno) values (1505, 100)
```

The trace comes in especially handy when you encounter errors in your dynamic SQL execution. Suppose you make extensive use of PLVdyn in your application. You start testing the code and have tossed in your face the following exception:

```
ORA−06510: PL/SQL: unhandled user−defined exception
```

You can turn on logging or displaying of exceptions if you are using PLVexc. Or you can simply issue this command:

```
PLVdyn.showsql;
```

and then re−execute your application. Suddenly, every time you parse SQL through PLVdyn, you see the SQL statement appear on the screen. When the problematic SQL fails, you will be able to identify the text and, from that, the part of your code that constructs that dynamic SQL string. Moving from that point to a solution is usually fairly straightforward.

The trace facility of PLVdyn illustrates some important principles of both generic package structure and high−quality reusable code. First, the public−private nature of the package allows me to construct a window into PLVdyn. This window offers a very controlled glimpse into the interior of the package. I let developers view the dynamic SQL string, but they can't look at or do anything else. This level of control allows Oracle to give us all of those wonderful builtin packages like DBMS_SQL and DBMS_PIPE. And it lets developers provide reusable PL/SQL components to other developers without fearing corruption of internal data structures.

See Chapter 2, *Best Practices for Packages*, for more information on windows into packages.

19.5.10 Executing PLVdyn in Different Schemas

When you execute stored code, you run it under the authority of the owner of that code. This has some startling implications for a package that frontends the DBMS_SQL package. Suppose that you install PLVdyn along with all of the other PL/Vision packages in a single account and then grant execute authority on PLVdyn to PUBLIC. This way all users can more easily take advantage of dynamic SQL in PL/SQL. Let's look at the implications of such an architecture.

Suppose that PL/Vision (and PLVdyn) is owned by the PLVISION account and that SCOTT is another account in the database instance. A DBA logs in to SCOTT to perform some database tuning. It seems that the company has added many employees over the years and the **emp** table now has 6,000,000 rows. An index is needed on the **ename** table to improve query performance.

So the DBA logs into the new Oracle Forms frontend she built to make her life easier. It uses PLVdyn to perform a variety of administrative tasks. Through a fill−in−the−form process, she ends up executing a CREATE INDEX command as follows:

```
PLVdyn.ddl ('create index empname_idx on emp(ename)');
```

So PLVdyn does its thing and the Oracle Forms application notifies the DBA that the job is done. The DBA is impressed at how rapidly the index was built and notifies the application development team that all is better now. Fifteen minutes of quiet pass for the DBA before she gets an angry call from a developer:

"The performance hasn't changed one bit!" he says angrily. "The screens still work just as slowly as before when I try to search for an employee by name."

19.5.10.1 Checking the indexes

The DBA is bewildered and quickly runs the following script to examine the indexes on the **emp** table:

```
REM vuindex.sql
SELECT i.index_name, i.tablespace_name,
       uniqueness u, column_name, column_position pos
  FROM all_indexes i, all_ind_columns c
 WHERE i.index_name = c.index_name
   AND i.table_name = upper ('&1');
SQL> start vuindex emp
INDEX_NAME          TABLESPACE_NAME      U         COLUMN_NAME          POS
------------------- -------------------- --------- -------------------- ---
EMP_PRIMARY_KEY     USER_DATA            UNIQUE    EMPNO                  1
```

There is no **empname_idx** index! What has gone wrong? On a hunch, the DBA connects to the PLVISION account, re−executes the same script, and sees these results:

```
INDEX_NAME          TABLESPACE_NAME      U         COLUMN_NAME          POS
------------------- -------------------- --------- -------------------- ---
EMPNAME_IDX         USER_DATA            NONUNIQUE ENAME                  1
EMP_PRIMARY_KEY     USER_DATA            UNIQUE    EMPNO                  1
```

The index was created in the PLVISION schema! Why did this happen?

When the DBA executed the index creation statement from within a call to **PLVdyn.ddl**, the DDL statement was processed as though it were being executed by PLVISION, not by SCOTT. If the DBA had wanted to create an index in her own schema, she should have entered the following command:

```
PLVdyn.ddl ('create index SCOTT.ename_idx on SCOTT.emp(ename)');
```

19.5.10.2 Avoiding schema confusion for DDL

It is very difficult to make sure that all developers who use PLVdyn remember this unusual twist. As a result, you may want to treat PLVdyn differently in terms of how you make it available to developers.

Ideally, one could enhance a package like PLVdyn to make sure that the correct schema (that returned by a call to USER) is applied to all DDL. As far as I can tell, however, that is a near−impossible task. How can a PL/SQL program parse the SQL statement and figure out where a schema must be added?

The alternative is ugly, but practical. Instead of creating the package in one account and granting execute authority to all others, consider creating the PLVdyn package directly in the accounts of each of the users of the package. This creates something of a maintenance headache, but it also guarantees that any SQL and PL/SQL developers run dynamically run under their own schema.

A final approach might be to strictly limit who has access to PLVdyn; perhaps it can only be executed through an interface that guarantees that schemas are applied correctly in the SQL before it is passed to PLVdyn for parsing.

Special Notes on PLVdyn

Here are some factors to consider when working with PLVdyn:

- The PLVdyn package does not provide support for bind variables. The PLVdyn1 package offers a programmatic interface to a variety of dynamic SQL operations that have a single bind variable. This package is provided on the companion disk, and you will find documentation for it there. You will be able to see how to use its programs simply by examining the specification and body of PLVdyn1.

- The **`disptab`** procedure has lots of room for fine−tuning and enhancement. It would be nice, for example, to specify a maximum length for numeric values, alternate titles for columns, and a restricted set of columns. Future versions of PL/Vision Professional may incorporate these ideas. I strongly recommend, however, that in the meantime you make a copy of **`disptab`** and play around with it yourself.

PREVIOUS | HOME | NEXT
19.4 PLVdyn: A Code Layer over DBMS_SQL | BOOK INDEX | 19.6 PLVfk: Generic Foreign Key Lookups

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

19.6 PLVfk: Generic Foreign Key Lookups

The PLVfk (PL/Vision Foreign Key) package offers a high−level, easy−to−use interface to looking up foreign key information in your tables.

In any well−normalized relational database, you have many foreign keys. For example, a customer database has companies, contacts within companies, types of companies, types of contacts, etc. The contact table has a foreign key to the company table. The company table has a foreign key to the company type table, and so on. Normalization is a great thing for avoidance of data redundancy, but it can create anguish for developers.

The reality of foreign keys is most succinctly expressed in the following maxim (first encountered in a scribble in the stall of an Oracle headquarters bathroom!):

Where you have foreign keys, you also have foreign key lookups.

In other words, whenever I query/display a contact record, I need to look up the name of the company, the description of the contact type, and so on in order to translate my keys (usually sequentially generated numbers or cryptic string codes) into information users can understand. Furthermore, whenever a user enters a name or description, I will want to look up the corresponding key to make sure the entry is valid.

Foreign key lookups have historically required an awful lot of custom programming. In fact, let's spend a few paragraphs exploring what you would normally have to do to handle foreign keys. Then I will show you how you can use dynamic SQL to create a generic function to handle almost all foreign key lookups.

19.6.1 Traditional Foreign Key Management

We'll stick to the company−contact model. The tables I will use are as follows:

```
CREATE TABLE company
   (company_id NUMBER,
    company_nm VARCHAR2(100),
    company_type_id NUMBER);

CREATE TABLE contact
   (contact_id NUMBER,
    company_id NUMBER,
    contact_ nm VARCHAR2(100),
    contact_type_id NUMBER);

CREATE TABLE company_type
   (company_type_id NUMBER, company_type_nm VARCHAR2(100);

CREATE TABLE contact_type
   (comntact_type_id NUMBER, contact_type_nm VARCHAR2(100);
```

The traditional (pre−DBMS_SQL) approach to foreign key lookups entails building a function for each separate entity that serves as a foreign key in a table. This function would take the foreign key and return the name or description. The specifications for such functions would look like:

```
FUNCTION contact_name (contact_id_in IN contact.contact_id%TYPE)
   RETURN VARCHAR2;

FUNCTION company_name (company_id_in IN company.company_id%TYPE)
   RETURN VARCHAR2;

FUNCTION company_type (company_type_id_in IN company.company_type_id%TYPE)
   RETURN VARCHAR2;
```

and so on, for as many foreign keys as you've got. And every time a new foreign key is added to the mix, you must write a new function. In addition, you would have a set of functions that take a name and return an ID. Lots of different program elements.

19.6.1.1 Typical foreign key lookup code

Here is an example of what only one of these functions would contain:

```
FUNCTION contact_name (contact_id_in IN contact.contact_id%TYPE)
   RETURN VARCHAR2
IS
   CURSOR con_cur (id_in IN NUMBER)
   IS
      SELECT contact_nm
        FROM contact
       WHERE contact_id = id_in;
   con_rec con_cur%ROWTYPE;

   return_value contact.name%TYPE := NULL;
BEGIN
   OPEN con_cur (contact_id_in);
   FETCH con_cur INTO con_rec;
   IF con_cur%FOUND
   THEN
      return_value := con_rec.contact_nm;
   END IF;
   CLOSE con_cur;
   RETURN return_value;
END contact_name;
```

Not a terribly complicated function, but when you repeat those steps over and over again, you end up with a significant volume of code to construct, debug, and maintain.

19.6.1.2 A better mousetrap

Wouldn't it be just fabulous if you could construct a single, generic function using dynamic SQL that would work for *all* foreign keys? The PLVfk package offers that single function. Actually, it provides two different general−purpose functions: one that accepts a key or ID and returns the associated name (the **`name`** procedure), and another that accepts a name and returns the associated key (the **`id`** procedure).

By using PLVfk, you can avoid writing functions like the **`contact_name`** program shown above. Instead, you simply execute this kind of command:

```
v_cname := PLVfk.name
              (v_contact_id, 'contact', 'contact_id', 'contact_nm');
```

and maybe even nothing more than this:

```
v_cname := PLVfk.name (v_contact_id, 'contact');
```

The elements of the PLVfk package are explained in the following sections. To get more information about the implementation behind PLVfk, you can review the code on the companion disk or read *Chapter 15* of

Oracle PL/SQL Programming.

19.6.2 Configuring the PLVfk Package

I have found that in many Oracle shops there are clear, consistent guidelines for naming tables and their columns. If you work in this kind of environment, you can leverage the predictability of these conventions into a user interface that both reflects and takes advantage of these standards.

In the PLVfk package, this means that you can inform PLVfk through the "set" programs how the column names for primary keys and their descriptors are constructed in relation to the entity or table name. You can tell it, for example, that the string "_ID" is always attached to the table name as a suffix to form the primary key column name. Or you can tell it that the descriptor column is always formed as the string "NAME$" attached as a prefix to the table name.

Once you have informed PLVfk of your standards, you do not have to constantly type in the names of your columns. Instead, you just pass in the table name and let PLVfk do the rest. And if you run into exceptions to your rule, you always can override the default conventions with full names or alternative conventions. All of these variations are explored in the following sections.

PLVfk provides three different "set" programs to provide override values to default elements of the PLVfk configuration: **`set_id_default`**, **`set_nm_default`**, and **`set_vclen`**.

19.6.2.1 Setting column name defaults

The **`set_id_default`** procedure sets the default string to be used as a suffix or prefix to the specified table name. The other two set programs determine how PLVfk constructs column names to be used in the dynamic SQL that retrieves the requested data. The header for **`set_id_default`** is:

```
PROCEDURE set_id_default
   (string_in IN VARCHAR2 := c_no_change,
    type_in IN VARCHAR2 := c_no_change);
```

where **`string_in`** is the string to be used as suffix or prefix and **`type_in`** is the type of concatenation action. The constant, **`c_no_change`**, can be used to indicate that you do *not* want to change one of these settings. The following examples illustrate the different ways to call **`set_id_default`**.

1. Change the prefix from the default of **`_ID`** to **`_KEY`**. Don't provide any value for the type, since the default is prefix and you are sure that hasn't been modified.

   ```
   SQL> exec PLVfk.set_id_default ('_KEY');
   ```

2. Rather than use prefixes, my site uses standard suffixes of **`ID_`** and **`NM_`**. So in my call to **`set_id_default`**, I only change the value of the type and leave the string itself the same (since they match the default, starting values).

   ```
   SQL> exec PLVfk.set_id_default (type_in=>PLVfk.c_suffix);
   ```

   The **`set_nm_default`** procedure performs the same kind of action, except that it applies to the name column and not the ID column. The header for this procedure is:

   ```
   PROCEDURE set_nm_default
      (string_in IN VARCHAR2 := c_no_change,
       type_in IN VARCHAR2 := c_no_change);
   ```

See the examples for **`set_id_default`** to get an idea of what you can do with **`set_nm_default`**.

19.6.2.2 Setting the default string length

The **`set_vclen`** procedure can be used to set the maximum size of a value returned by DBMS_SQL with a call to COLUMN_VALUE. The header for **`set_vclen`** is:

```
PROCEDURE set_vclen (length_in IN INTEGER);
```

The default value for this maximum length is 100. The following call to **`set_vclen`** notifies PLVfk that it may encounter descriptors of up to 255 bytes in length:

```
SQL> exec PLVfk.set_vclen (255);
```

Once you have set the PLVfk package to reflect as closely as possible your server environment and standards, you can use the id and name functions of PLVfk to perform effortless foreign key lookups.

19.6.3 Looking Up the Name

You can use the **`name`** function to retrieve the name for a specific key. The header for **`name`** is:

```
FUNCTION name
   (id_in IN INTEGER,
    table_in IN VARCHAR2,
    id_col_in IN VARCHAR2 := c_no_change,
    nm_col_in IN VARCHAR2 := c_no_change,
    max_length_in IN INTEGER := c_int_no_change,
    where_clause_in IN VARCHAR2 := NULL)
RETURN VARCHAR2;
```

where **`id_in`** is the specific ID value used in the lookup and **`table_in`** is the name of the table to be searched. These are the only required arguments. The next four arguments allow you to tweak the SQL statement constructed by PLVfk, as described below:

`id_col_in`
: The name of the ID column in the table. If you do not supply a value, the default column suffix or prefix is applied to the table name. If you do supply a value and it starts or ends with an underscore, that string is used as a suffix or prefix. Otherwise, the string you supply is used as the column name itself.

`nm_col_in`
: The name of the name or descriptor column in the table. If you do not supply a value, the default column suffix or prefix is applied to the table name. If you do supply a value and it starts or ends with an underscore, that string is used as a suffix or prefix. Otherwise, the string you supply is used as the column name itself.

`max_length_in`
: The maximum length of the string value returned by the query. If no value is provided, the current value (initially 100 and set by the **`set_vclen`** procedure) is used in the call to the DBMS_SQL.DEFINE_COLUMN builtin.

`where_clause_in`
: Optional WHERE clause to attach to the SELECT statement. You can use this argument to add additional AND clauses, such as a restriction to look only at active records.

The following series of examples demonstrate how to use the **`name`** function.

1.

Assume that table contact has **`contact_id`** and **`contact_nm`** columns.

```
v_name := PLVfk.name (v_contact_id, 'contact');
```

2.
Assume that **`contact_type`** table has **`contact_type_id`** and **`contact_type_nm`** columns.

```
v__type_ds := PLVfk.name (v_type_id, 'contact_type');
```

Of course, in the real world, conventions do not hold up so consistently. In fact, I have found that database administrators and data analysts often treat an entity like contact, with its contact ID number and contact name, differently from the way they would a contact type, with its type code and description. The columns for the contact type table are more likely to be: **`contact_typ_cd`** and **`contact_typ_ds`**. Fortunately, **`PLVfk.name`** still handles this situation as shown in number 3.

3.
Use alternative suffixes for a code table.

```
v_type_ds := PLVfk.name (v_type_id, 'contact_type', '_cd', '_ds');
```

4.
Only retrieve the description of the call type if that record is still flagged as active. Notice that I must stick several single quotes together to get the right number of quotes in the evaluated argument passed to **`name`**.

```
v_contact_type_ds :=
  PLVfk.name
    (v_contact_type_id,
     'contact_type', '_cd', '_ds', 25,
     'AND row_active_flag = ''Y''');
```

In examples 3 and 4, I could avoid specifying the column suffixes by making a prior call to **`set_id_default`** and **`set_nm_default`**, as shown below:

```
PLVfk.set_id_default ('_cd');
PLVfk.set_nm_default ('_ds');
```

and now my calls to PLVfk.name are made simpler:

```
v_type_ds := PLVfk.name (v_type_id, 'contact_type');

v_contact_type_ds :=
  PLVfk.name
    (v_contact_type_id,
     'contact_type', 25, 'AND row_active_flag = ''Y''');
```

5.
Retrieve the name of the store that is kept in the record for the current year. I use named notation to skip over all intermediate arguments for which I want to use the default values and specify my WHERE clause. Notice that I must use two contiguous single quotes inside my string so that it evaluates to a valid string for a SQL statement.

```
/* Only the record for this year should be used */
year_number := TO_CHAR (SYSDATE, 'YYYY');
/* Pass check for year to WHERE clause. */
v_description :=
  name
    (v_store_id, 'store_history',
     where_clause_in =>
        'AND TO_CHAR (eff_date, ''YYYY'') = ''' ||
                    year_number || '''');
```

The following table shows how various arguments for the ID and name column strings are converted into the column names concatenated into the dynamic SQL SELECT statement.

Formal Parameter	Argument Supplied to PLVfk.name	Converted Value
ID column	**c_no_change** or simply skipped	**contact_id**
Name column	**c_no_change** or simply skipped	**contact_nm**
ID column	**contact_number**	**contact_number**
Name column	**contact_name**	**contact_name**
ID column	_#	contact_#
Name column	_fullname	**contact_fullname**

19.6.4 Looking up the ID

Sometimes you want to get the key or ID number from a name. In this case, you use the **id** function, whose header is shown below:

```
FUNCTION id
   (nm_in IN VARCHAR2,
    table_in IN VARCHAR2,
    id_col_in IN VARCHAR2 := c_no_change,
    nm_col_in IN VARCHAR2 := c_no_change,
    max_length_in IN INTEGER := c_int_no_change,
    where_clause_in IN VARCHAR2 := NULL)
RETURN INTEGER;
```

where **nm_in** is the specific name used in the lookup and **table_in** is the name of the table to be searched. These are the only required arguments. The next four arguments allow you to tweak the SQL statement constructed by PLVfk, as described below:

id_col_in
: The name of the ID column in the table. If you do not supply a value, the default column suffix or prefix is applied to the table name. If you do supply a value and it starts or ends with an underscore, that string is used as a suffix or prefix. Otherwise, the string you supply is used as the column name itself.

nm_col_in
: The name of the name or descriptor column in the table. If you do not supply a value, the default column suffix or prefix is applied to the table name. If you do supply a value and it starts or ends with an underscore, that string is used as a suffix or prefix. Otherwise, the string you supply is used as the column name itself.

max_length_in
: The maximum length of the string value returned by the query. If no value is provided, the current value (initially 100 and set by the **set_vclen** procedure) is used in the call to the DBMS_SQL.DEFINE_COLUMN builtin.

where_clause_in
: Optional WHERE clause to attach to the SELECT statement. You can use this argument to add additional AND clauses, such as a restriction to only look at active records.

As you can see, the **id** function is very similar to the **name** function; they are mirrors of each other. Rather than repeat another set of examples, see those supplied for the **name** function to get a feeling for how to use the **`PLVfk.id`** function.

Special Notes on PLVfk

Here are some factors to consider when working with PLVfk:

- You cannot call either of the PLVfk functions from within a SQL statement. Yes, it is a PL/SQL function and you can call PL/SQL functions from within SELECTs, INSERTs, and so on. You cannot, on the other hand, call a PL/SQL function which, in turn, executes any programs from a builtin package. Since PLVfk relies heavily on DBMS_SQL, it is not callable from within SQL. I received more messages (complaints?) about this problem (usually phrased more like "your code doesn't work!") from my first book than on any other topic.

- David Thompson, ace technical reviewer of this book and author of PLVddd, offered an interesting idea for enhancing PLVfk further. Why depend on all those complicated column name prefixes and suffixes to determine the primary key? Why not read the data dictionary for this information instead? You could even store all primary key information in a PL/SQL table to avoid repetitive data dictionary access. An excellent idea to explore, although I would be concerned about adding more overhead to PLVfk processing.

PREVIOUS | HOME | NEXT
19.5 DML Operations | BOOK INDEX | 20. PLVcmt and PLVrb: Commit and Rollback Processing

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

20. PLVcmt and PLVrb: Commit and Rollback Processing

Contents:

PL/Vision provides two packages that help you manage transaction processing in PL/SQL applications: PLVcmt and PLVrb. These packages provide a layer of code around the transaction−oriented builtins of the PL/SQL language. If you make full use of PLVcmt and PLVrb, you will no longer need to make direct calls to COMMIT and ROLLBACK. Instead, you will call the corresponding programs in these two packages; by doing so, you will greatly increase your flexibility, improve your ability to test your code, and reduce overall code volume.

20.1 PLVcmt: Enhancing Commit Processing

The PLVcmt (PL/Vision CoMmiT) package encapsulates logic and complexity for dealing with commit processing. For example, you can use PLVcmt to rapidly define scripts that execute commits every 1,000 transactions. You can replace any direct calls to COMMIT with a call to PLV`cmt.perform_commit` and thereby give yourself additional testing and debugging flexibility. By providing a programmatic interface to commit activity in PL/SQL, PL/Vision gives you the ability to change transaction−level behavior at runtime. You can also reduce the volume of code you write to perform commits in your applications.

20.1.1 Who Needs PLVcmt?

In my first book on PL/SQL, I used more that 900 pages to talk about almost every aspect of the PL/SQL language. But notice that word: almost. I did not, in fact, cover two very important commands in PL/SQL: COMMIT and ROLLBACK. Why didn't I discuss these commands? For two reasons: oversight and rationalization. The oversight was due to the fact that I had up to that time rarely performed commits in my PL/SQL programs (they were usually a part of Oracle Forms applications or were developer utilities).

When I did issue a commit, I didn't pay much attention. There just wasn't much to it. And that is where the rationalization part of the explanation comes in. Even when I did realize that COMMIT and ROLLBACK were missing from my book (fairly late in the game, but in time to include them), I said to myself: they are so simple and easy to use. I don't really need to write about that aspect of PL/SQL. Everybody knows about them from the SQL language anyway.

Since those days, I have had good reason to take a second, longer look at the (deceptively) simple COMMIT statement and its use in PL/SQL programs. I found from work at an account in early 1996 that there can be much more to committing than meets the eye. I found, in particular, that by managing your commit processing from within a package you can greatly improve your flexibility in testing. You will also gain an additional, welcome level of control in your batch processing applications. In fact, coming out of my experience I would make the following recommendation:

> ***NOTE:*** You should never make a direct call to COMMIT, ROLLBACK, or SAVEPOINT in your code. By doing so, you hard−code irreversible operations into your PL/SQL programs and limit your flexibility.

At this point, you must surely consider me a package fanatic. What is the big deal about the COMMIT statement? Why would you possibly want to go to the trouble of building a package just to do a commit? Well, certainly on the logical level, a COMMIT is a very big deal. It completes a transaction by saving information to the database. When you are writing applications to manage data in your Oracle7 Server, the commit is a central and critical step. It certainly is easy enough to do a commit. You just type the following statement in your program:

```
COMMIT;
```

No, executing a commit is easy. Determining when and how often to do the commit can be more challenging. Managing rollbacks is, furthermore, even more interesting. I found that a package gave me the flexibility I needed to meet their requirements. The PLVcmt package arose from this application's challenges.

20.1.1.1 Commit processing challenges

My customer, which I'll refer to as Bigdata Inc., needed to perform a complex data translation from one Oracle database instance to another. Approximately 20 million records in two tables were involved. It wasn't one of those all−or−nothing situations. If we could manage to get through a million records before some failure occurred, that was fine. All we had to do was come up with a mechanism for keeping track of which records had already been processed, so we didn't do them again. We used a "transfer indicator," which also led to a distributed transaction.

I've got to be honest with you: I have not spent many hours of my career (prior to Bigdata) working with this kind of volume of data. It sure puts a different spin on everything you do. When a simple SELECT could take hours to complete, you get very careful about the queries you execute. You no longer make casual statements (and take casual actions) like: "Let's just run this and see how it works. If it fails, we'll try it again." Three days later (or two weeks later), the job might crash from a "snapshot too old" error and you are back at square one −− if you didn't take precautions.

In fact, I quickly became intimate with a range of Oracle errors that earlier had been fairly academic to me: the −015NN series. Errors like:

```
ORA−01555 snapshot too old (rollback segment too small)
ORA−01562 failed to extend rollback segment
```

became a regular and unwelcome part of my life. Sure, we had big rollback segments, but one of our tables took up 2 gigabytes of storage in the database. We simply couldn't execute queries across the entire table at the same time that updates were taking place. I learned to "chunk down" by primary key ranges the rows I processed in each pass of my program. And I discovered the need to get flexible when it came to commits.

20.1.1.2 Committing every ? records

When I first started with the account, we agreed that committing every 10,000 records would be great. This is the kind of code we wrote to handle the situation:

```
commit_counter := 0
FOR original_rec IN original_cur
LOOP
   translate_data (original_rec);
   IF commit_counter >= 10000
   THEN
      COMMIT;
      commit_counter := 0;
   ELSE
      commit_counter := commit_counter + 1;
   END IF;
END LOOP;
COMMIT;
```

Of course, there were a number of different programs and each had this logic, along with a declaration of **`commit_counter`**, in each program. We soon found, however, that 10,000 was simply too high a number. We blew out rollback segments on a maddeningly occasional, but unpredictable basis. So we decided to change the number to 1,000 and off I went to each of the different programs removing that troublesome zero.

I felt dumb doing this, but of course we faced looming deadlines and had no time to reflect. The next complication I ran into was the need to run my script in "test mode:" perform the data translation for one or several records and then examine the accuracy of the data. In this situation, I found that I would rather not commit at all. Just run the program, use queries to examine the changes, and then issue a ROLLBACK. To do this, I went back into my program and commented−out the entire IF statement having to do with commits and keeping counts.

```
commit_counter := 0
FOR original_rec IN original_cur
LOOP
   translate_data (original_rec);
   /*
   IF commit_counter >= 1000
   THEN
      COMMIT;
      commit_counter := 0;
   ELSE
      commit_counter := commit_counter + 1;
   END IF;
   */
END LOOP;
COMMIT;
```

Once I got through several debug−test cycles, I reactivated my commit logic by removing the comment markers and recompiling.

20.1.1.3 What committed when?

It was then time to run the process for the full sweep of the data (in its manageable chunks). So I started the program and went home for the weekend. Saturday and Sunday were very pleasant, but I came back in on Monday and found that the job has stopped on Sunday afternoon. I had a heck of a time figuring out why it had stopped and how far it had gotten. I realized that it would have been very useful to have a log of each commit performed by the program. So I changed my basic loop (shown previously) as follows:

```
commit_counter := 0
FOR original_rec IN original_cur
LOOP
   translate_data (original_rec);
   IF commit_counter >= 1000
   THEN
      COMMIT;
      DBMS_OUTPUT.PUT_LINE
        ('Commit at' ||
         original_rec.keyvalue);
      commit_counter := 0;
   ELSE
      commit_counter :=
          commit_counter + 1;
   END IF;
END LOOP;
COMMIT;
```

Now, every time a commit occurred, the primary key value would be displayed. I set up this job to run and after an hour or two it died −− this time because my program's output had exceeded the DBMS_OUTPUT default buffer of 2K! This was getting very frustrating. Maybe I should expand the size of the buffer. Maybe I should be writing the commit log out to a table. Or maybe it was just time for a break.

With a moment to reflect, I saw the insanity of my way. Here I was putting out little fires, patching up this hole, then that hole in my logic. In the process, the code I had written to perform commits was actually getting more complicated than the actual application logic −− and, again, it was repeated in several different programs.

20.1.1.4 Getting back on track

Time out! I declared to myself and the rest of the technical team. I had committed several grievous errors, any one of which should have raised a red flag:

1. I repeated the same code in multiple programs (declaration of **`commit_counter`**, IF statement, etc.). This should always be avoided by consolidating repetitive code behind a procedural interface.

2. I edited my code in order to move from "test mode" to "production" status −− I inserted and then removed the comment markers. You want to avoid whenever possible these kinds of last−minute, "no problem" edits of code. Any time you change your code, you really should retest. Do you want to introduce another round of testing right after you thought you *finished* all your testing?

No, it was time to go back to square one, do some top−down design, and do it right...the second time around.

The first thing that caught my eye is that I could simply be smarter about how to determine when to perform my commit. Rather than use an independent counter, I could take advantage of the %ROWCOUNT cursor attribute to figure out how many rows I had fetched. Combined with use of the MOD function, I could change my loop to the following, more concise implementation:

```
FOR original_rec IN original_cur
LOOP
   translate_data (original_rec);
   IF MOD (original_cur%ROWCOUNT, 1000) = 0
   THEN
      COMMIT;
   END IF;
END LOOP;
COMMIT;
```

In this approach, whenever the number of rows fetched is a multiple of 1000, the MOD function returns 0 and COMMIT is executed. No local variable counter to declare and maintain −− when working within a cursor loop anyway. This was a satisfying discovery, but it didn't address some of my other concerns: turning off commits for test purposes, changing the number to use in the call to MOD, and so on. No, I decided to press on...and come up with a package−based solution, which turned into PLVcmt.

20.1.1.5 The impact of PLVcmt

Here is what my data translation loop looks like when I use a package−based approach:

```
PLVcmt.init_counter;
FOR original_rec IN original_cur
LOOP
   translate_data (original_rec);
   PLVcmt.increment_and_commit;
END LOOP;
PLVcmt.perform_commit;
```

In other words:

1. I initialize my commit package values.

2.

I commit based on the counter inside the loop.

3. Then after the loop terminates, I perform a final commit.

Notice that the **`commit_counter`** variable has disappeared. I don't want to deal with that. Also gone is the code to display the commit action and the IF statement. Nor can you find a call to COMMIT. It's all tucked away somewhere else. Ah! A sigh of relief. And −− here is where it gets really dreamy −− if I want to run the program and *not* perform any commits, I wish to be able to simply call another PLVcmt program to tell it not to commit, like this:

```
execute PLVcmt.turn_off;
```

Without making any changes to *my* program, the behavior of the PLVcmt package would change. Now that would be a wondrous thing, would it not? Let's see how we might go about building such a package, because at least in this case, my fantasies can be transformed fully into reality.

Now that you've seen the inspiration behind PLVcmt and how useful it can be, it's time for a formal introduction. These following sections explain these features of the PLVcmt package:

- Using a package−based substitute for COMMIT
- Performing incremental commits
- Controlling commit processing
- Logging commit activity

20.1.2 The COMMIT Substitute

PLVcmt offers two programs that can perform commits for you: **`perform_commit`** and **`increment_and_commit`**. The **`perform_commit`** program is a direct substitution for COMMIT. The **`increment_and_commit`** program is used in conjunction with loops in situations where you want to commit every *n* transactions.

The header for **`perform_commit`** is:

```
PROCEDURE perform_commit (context_in IN VARCHAR2 := NULL);
```

The single argument to **`perform_commit`**, **`context_in`**, is an optional string that you want to associate with this commit point. This string is then logged with the PLVlog package when a commit is performed through PLVcmt *and* the user has requested that commits be shown.

A direct substitution for a call to COMMIT is this statement:

```
PLVcmt.perform_commit;
```

The following call to this procedure associates the commit point with a calculation of net sales for the current year.

```
PLVcmt.perform_commit ('Net sales ' || TO_CHAR (v_curr_year));
```

This string is ignored unless you have executed the PLVcmt.log command to turn on logging of commits.

20.1.3 Performing Incremental Commits

When you use PLVcmt, it is very easy to write code that handles the following kind of requirement: "commit every 100 records." With PLVcmt, you don't have to declare a local counter, increment the counter, or call COMMIT. Instead, you simply make calls to the appropriate PLVcmt programs and concentrate on writing your transaction logic.

Three PLVcmt programs implement incremental commits: the **`init_counter`**, **`commit_after`**, and **`increment_and_commit`** procedures.

20.1.3.1 Setting the commit point

The first step in using PLVcmt to perform incremental commits is to tell the package how often you want a commit to occur. You do this with the **`commit_after`** procedure, whose header is shown below:

```
PROCEDURE commit_after (count_in IN INTEGER);
```

where **`count_in`** is the number of transactions you want to occur before a commit takes place. The default, initial value of the count is 1, which means that every time you call **`PLVcmt.increment_and_commit`**, a COMMIT is executed (unless you have turned off commit processing, which is discussed in the next section).

In the following call to **`commit_after`**, I request that a commit occur every 100 transactions.

```
PLVcmt.commit_after (100);
```

In this next call to **`commit_after`**, I set the commit point to 0.

```
PLVcmt.commit_after (0);
```

This effectively turns off the execution of a COMMIT from within the **`increment_and_commit`** program. With the "commit after" set to zero, a COMMIT occurs only when **`PLVcmt.perform_commit`** is called.

> ***NOTE:*** The commit point established by the **`commit_after`** procedure does not in any way affect the behavior of the **`perform_commit`** procedure.

20.1.3.2 Initializing the counter

You have called **`PLVcmt.commit_after`** to tell PLVcmt that you want to commit every n records. Before you start running your code, you should initialize the PLVcmt counter to make sure that *n* transactions occur before a commit.

The header for **`init_counter`** is:

```
PROCEDURE init_counter;
```

When called, this program sets the internal PLVcmt counter to 0. The only way to modify this counter is with a call to **`init_counter`** or to **`increment_and_commit`**.

20.1.3.3 Increment and commit

When you want to commit every *n* records, you can simply insert a call to the **`increment_and_commit`** procedure in your code. The header is:

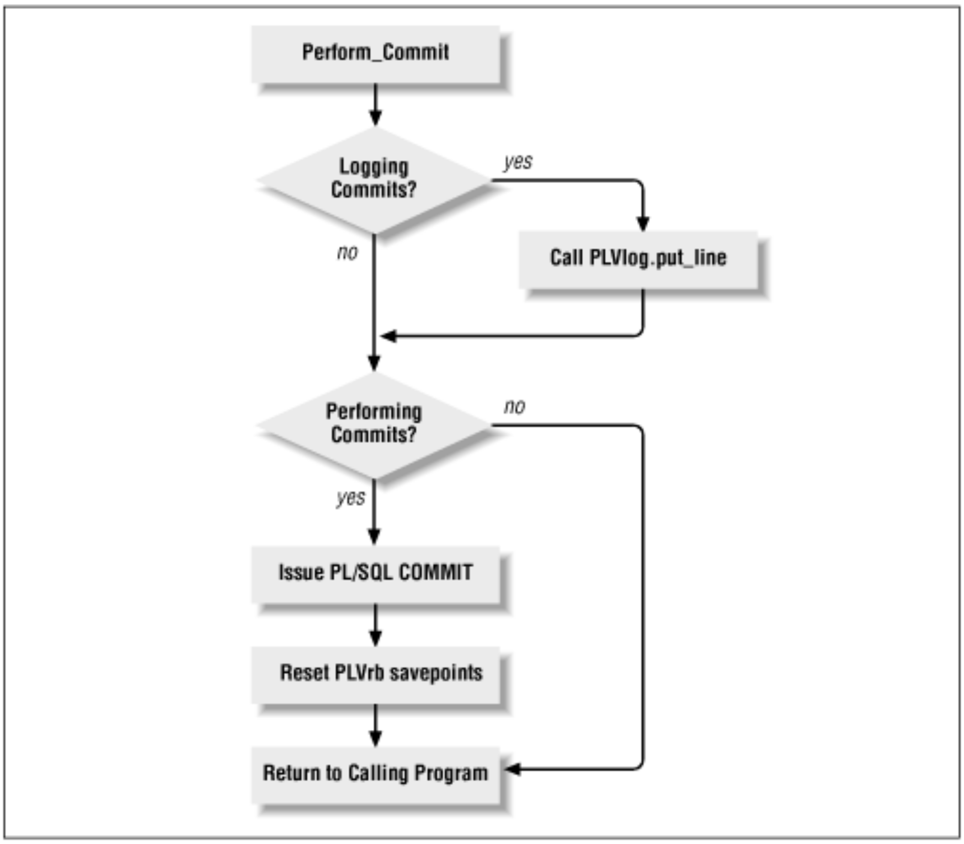
```
PROCEDURE increment_and_commit (context_in IN VARCHAR2 := NULL);
```

This program always increments the PLVcmt counter. If the counter exceeds the commit−after value set with a call to **`PLVcmt.commit_after`**, then the **`perform_commit`** procedure is called. Immediately after that, PLVcmt calls its own **`init_counter`** to reset the counter to 0.

20.1.4 Controlling Commit Processing

One of the big advantages to using PLVcmt instead of issuing direct calls to COMMIT is that you have placed a layer of code between your application and COMMIT. This layer gives you (through PL/Vision) the ability to modify commit processing behavior *without changing your application code.* This is very important because it allows you to stabilize your code, but still change the way it works for purposes of testing and debugging. Figure 20.1 shows this layer of code.

Figure 20.1: Code layer around COMMIT



By using PLVcmt you can, in fact, actually disable COMMITs in your application. I have found this feature useful when I am working with test data. My code changes the data, but then I have to change it *back* for the next test. If, however, I call PLVcmt commit programs instead of issuing direct calls to COMMIT, I can simply tell PLVcmt to not commit for the test run. I can then run my code, examine the results within my current session, and perform a rollback. No recovery scripts are necessary.

PLVcmt offers a standard PL/Vision toggle to control commit processing. This triumvirate of programs is:

```
PROCEDURE turn_on;
PROCEDURE turn_off;
FUNCTION committing RETURN BOOLEAN;
```

One procedure to turn on commit processing, another to turn it off, and a final function to indicate the current state of affairs (for completeness *and* politeness). All the two procedures do is set the value of a private Boolean variable, but by correctly applying that Boolean inside an IF statement in PLVcmt, the package's user gets to fine−tune the package's behavior.

20.1.5 Logging Commits

The commit action in an application is a critical step. It is, for one thing, irreversible. Once you commit, you cannot uncommit. It is often very useful to know when commits have taken place and the action that was taken around that commit point. I have found this to be most important when I am executing long−running processes with incremental commits. How far along am I in processing my ten million transactions? The PLVcmt logging facility gives you access to this information.

Whenever you call **`PLVcmt.perform_commit`** and **`PLVcmt.increment_and_commit`**, you can supply a string or context for that action. This string is ignored unless logging is turned on. If logging is enabled, PLVcmt calls the PLVlog facility to log your message. You can, within PLVlog, send this information to a database table, PL/SQL table, operating system file (with Release 2.3 of PL/SQL), or standard output (your screen).

PLVcmt offers a standard PL/Vision toggle to control commit processing. This triumvirate of programs is:

```
PROCEDURE log;
PROCEDURE nolog;
FUNCTION logging RETURN BOOLEAN;
```

> ***NOTE:*** You do not have to turn on logging in PLVlog for the PLVcmt log to function properly. It will automatically turn on logging in PLVlog in order to write its commit−related information, and then reset the PLVlog status to its prior state.

20.1.6 Using PLVcmt

The following several examples show how to use these different elements of PLVcmt. First, we'll recast the previous anonymous block as a procedure so that it can be called from within a SQL*Plus session:

```
PROCEDURE translate_all
IS
   CURSOR original_cur IS SELECT ...;
BEGIN
   PLVcmt.init_counter;
   FOR original_rec IN original_cur
   LOOP
      translate_data (original_rec);
      PLVcmt.increment_and_commit
         (original_rec.keyvalue);
   END LOOP;
   PLVcmt.perform_commit;
END translate_all;
```

Again, notice that by using PLVcmt I do not have to declare or manage a counter. By default, this program commits the data translation for each record fetched by the original data cursor and then commits on the way out. Whenever I increment inside the loop, I also pass the key value to PLVcmt for display −− when so specified. Now I can call this program in conjunction with other PLVcmt modules.

In the following SQL*Plus session, I specify that a commit should take place every 1,000 records:

```
SQL> exec PLVcmt.commit_after (1000)
SQL> exec translate_all
```

In this next SQL*Plus session, I request a display of the commits as they occur and then sit back and watch the results:

```
SQL> exec PLVcmt.commit_after (1000)
SQL> exec PLVcmt.log
SQL> exec translate_all
commit ON IL123457
commit ON KY000566
commit ON NY121249
```

The reason PLVcmt indicates that commit is on is that I could run this same session, but avoid doing any commits. However, I might still want to see the output to verify correctness. Here we go:

```
SQL> exec PLVcmt.turn_off
SQL> exec PLVcmt.log
SQL> exec translate_all
commit OFF IL123457
commit OFF KY000566
commit OFF NY121249
```

Notice that I can achieve this change in behavior of my application without making any changes whatsoever to the application code itself. All the logic and complexity is hidden behind the interface of the package.

Since the writing of my first book, I have fallen in with love (sure, go ahead, laugh at me!) with these kinds of toggles and tightly controlled windows into packages. I can write my basic, useful package and then pack into it all kinds of flexibility, controlled by the user of the package. This flexibility makes my package more useful in a variety of circumstances. This improved usability increases the reusability of my own code and the code of others who have begun to use PL/Vision.

Special Notes on PLVcmt

One idea for an enhancement: Allow the developer to specify a commit not simply when a counter is reached, but also when a Boolean expression evaluates to TRUE. For example, you might want to commit when the balance in an account exceeds $1000. To do this, you might consider overloading the **`increment_and_commit`** procedure to accept a Boolean parameter -- or create an additional program entirely.

PREVIOUS 19.6 PLVfk: Generic Foreign Key Lookups | HOME BOOK INDEX | NEXT 20.2 PLVrb: Performing Rollbacks

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

20.2 PLVrb: Performing Rollbacks

The PLVrb PL/Vision RollBack package provides a programmatic interface to rollback activity in PL/SQL. With PLVrb, you no longer issue explicit ROLLBACK and SAVEPOINT commands. Instead, you call the appropriate PLVrb module. This layer of code gives you the ability to change transaction−level behavior at runtime. With PLVrb, you can even execute soft−coded savepoints, a feat usually considered impossible in PL/SQL.

These different elements of the PLVrb package are explained in the following sections.

20.2.1 Controlling Rollbacks

One of the big advantages to using PLVrb instead of direct calls to ROLLBACK is that you have placed a layer of code between your application and the ROLLBACK. This layer gives you (through PL/Vision) the ability to modify rollback processing behavior *without changing your application code.* This is very important because it allows you to stabilize your code, but still change the way it works for purposes of testing and debugging.

PLVrb offers a standard PL/Vision toggle to control rollback processing. This triumvirate of programs is:

```
      PROCEDURE turn_on;
      PROCEDURE turn_off;
      FUNCTION rolling_back RETURN BOOLEAN;
```

All that the first two procedures do is set the value of a private Boolean variable, but by correctly applying that Boolean inside an IF statement in PLVrb, the package's user gets to fine−tune the package's behavior.

20.2.2 Logging Rollbacks

The rollback action in an application is a critical step. It is, for one thing, irreversible. Once you rollback, you cannot un−rollback. It is often very useful to know when rollbacks have taken place and the action that was taken around that rollback point. I have found this to be most important when I am executing long−running processes. Have I issued any rollbacks? If so, what was the cause? The PLVrb logging facility gives me the answers to these types of questions.

Whenever you call **`PLVrb.perform_rollback`**, **`PLVrb.rb_to_last`**, and **`PLVrb.rollback_to`**, you can supply a string or context for that action. This string is ignored unless logging is turned on. If logging is enabled, PLVrb calls the PLVlog facility to log your message. You can, within PLVlog, send this information to a database table, PL/SQL table, operating system file (with Release 2.3 of PL/SQL), or standard output (your screen).

PLVrb offers a standard PL/Vision toggle to control the logging of rollback processing. This triumvirate of programs is:

```
PROCEDURE log;
PROCEDURE nolog;
FUNCTION logging RETURN BOOLEAN;
```

NOTE: You do not have to turn on logging in PLVlog for the PLVrb log to function properly. It automatically turns on logging in PLVlog in order to write its rollback−related information, and then reset the PLVlog status to its prior state.

20.2.3 Setting Savepoints

When you set a savepoint in PL/SQL, you give yourself a spot in your code to which you can rollback your changes. This is useful when you need to discard some, but not all, of your uncommitted changes. The usual situation you face with PL/SQL is that you must hard code the names of savepoints in your code. A savepoint is, in fact, an undeclared identifier. You don't (and cannot) declare a savepoint in your declaration section, as you would with an exception. Instead, you simply provide an identifier after the keyword SAVEPOINT in your code and that savepoint is established. Then, when you issue a ROLLBACK, you must also hard code that same identifier value in the ROLLBACK.

In the following block of code, I set a savepoint and then in the exception section rollback to that same savepoint.

```
BEGIN
   SAVEPOINT start_trans;
   INSERT INTO emp ...;
   DELETE FROM emp_history ... ;
EXCEPTION
   WHEN OTHERS
   THEN
      ROLLBACK TO start_trans;
END;
```

I cannot, on the other hand, write code like this:

```
PACKAGE empsav
IS
   insert_point VARCHAR2(10) := insert ;
   delete_point VARCHAR2(10) := delete ;
END;

BEGIN
   SAVEPOINT empsav.insert_point;
   INSERT INTO emp ... ;
   SAVEPOINT empsav.delete_point;
   DELETE FROM emp_history ... ;
EXCEPTION
   WHEN DUP_VAL_ON_INDEX
   THEN
      ROLLBACK TO empsav.insert_point;

   WHEN empsav.still_active
   THEN
      ROLLBACK TO empsav.delete_point;
END;
```

PL/SQL will not, absolutely not, evaluate the packaged contents into a literal and then use that literal to direct rollback activity. Instead, the code will fail to compile as shown:

```
SAVEPOINT empsav.insert_point;
                *
ERROR at line 2:
ORA−06550: line 2, column 20:
```

```
PLS−00103: Encountered the symbol "." when expecting one of the
   following:
```

One consequence of this hard−coding is that you must know the name of the savepoint at compile time, not at runtime. In most situations, this might be fine. In other programs, this can be a significant obstacle. PLVlog, for example, offers a generic logging mechanism. When logging to a database table, you must often perform a ROLLBACK before an INSERT to the log table and then follow up with the setting of a SAVEPOINT. And it really needs to do these steps for a dynamically determined savepoint.

This dynamic setting of savepoints (and rolling back to those savepoints) is provided by the PLVrb package. To set a savepoint whose name is determined at runtime, call the **`set_savepoint`** procedure. The header for this program is:

```
   PROCEDURE set_savepoint (sp_in IN VARCHAR2);
```

where **`sp_in`** is the savepoint name. The **`sp_in`** argument must be a valid PL/SQL identifier (starts with a letter, is composed of letters, digits, $, #, and the underscore character, and must be no longer than 30 characters in length).

Every time you set a savepoint, the procedure takes the following actions:

1. Set the savepoint with a call to **`PLVdyn.plsql`**. The **`set_savepoint`** programs constructs the SAVEPOINT command and executes it dynamically.

2. Sets the last savepoint value to the provided savepoint.

3. Pushes the savepoint onto the stack of savepoints maintained by PLVrb using the PLVstk package.

Instead of issuing statements like this:

```
SAVEPOINT start_trans;
```

you can now pass the name of your savepoint to PLVrb for setting:

```
PLVrb.set_savepoint ('start_trans');
```

or:

```
PLVrb.set_savepoint (v_starttrans_sp);
```

20.2.4 Performing Rollbacks

PLVrb offers three programs to perform rollbacks: **`perform_rollback`**, **`rollback_to`**, and **`rb_to_last`**. The headers for these programs are:

```
PROCEDURE perform_rollback
   (context_in IN VARCHAR2 := NULL);

PROCEDURE rollback_to
   (sp_in IN VARCHAR2,
    context_in IN VARCHAR2 := NULL);

PROCEDURE rb_to_last
   (context_in IN VARCHAR2 := NULL);
```

In all three procedures, the **`context_in`** argument is a string that will be logged using PLVlog if you have called **`PLVrb.log`** to turn on logging.

When you call **`perform_rollback`**, a full, unqualified rollback is performed; no savepoint is used, and all uncommitted changes are rolled back.

When you call **`rollback_to`**, PLVrb issues a ROLLBACK to the specified savepoint. Besides issuing the savepoint, PLVrb also removes from the savepoint stack any savepoints that came after the savepoint you specified. If the savepoint argument is not in the PLVrb savepoint stack, the stack is emptied.

When you call **`rb_to_lst`**, PLVrb issues a ROLLBACK to the savepoint specified in the most recent call to **`set_savepoint`**.

Special Notes on PLVrb

If you want to use PLVrb, you should also use PLVcmt. This commit package maintains the savepoint stack of PLVrb. If you use PLVrb to set savepoints and then issue explicit COMMITS in your programs, the savepoint stack will be out of sync with your transaction.

PREVIOUS	HOME	NEXT
20.1 PLVcmt: Enhancing Commit Processing	BOOK INDEX	21. PLVlog and PLVtrc: Logging and Tracing

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

21. PLVlog and PLVtrc: Logging and Tracing

Contents:

PL/Vision provides two packages to provide execution tracing and logging from within PL/SQL programs.

- Use the PLVlog package to write information to a log.
- Use the PLVtrc package to trace the execution of your PL/SQL programs.

Both of these packages are used by other PL/Vision packages, PLVlog in particular. For example, when you handle an exception with a PLVexc handler like **`rec_continue`**, it, in turn, calls PLVlog to log the error.

21.1 PLVlog: Logging Activity in PL/SQL Programs

The PLVlog (PL/Vision LOGging) package offers a powerful, generic logging mechanism for PL/SQL−based applications. The need to log activity arises in a number of different settings, including logging errors, tracing execution, and auditing activity. Rather than build this functionality over and over again, you can use PLVlog to handle many different circumstances.

The central features of PLVlog include:

- Writing of log information to one of several different repositories, including a database table, a PL/SQL table, or an operating system file (for PL/SQL Release 2.3 and above).
- Simultaneous, optional display of logged information.
- Toggle processing of log activity. You can decide to turn off logging at any time without having to change your own application code.
- High−level programs to manage the log. You can display contents of the log with a single program call, and transfer the log contents from a PL/SQL table to a database table.
- Automated rollback and savepoint activity. The package provides the ability to preserve database−logged information even if the surrounding transaction is rolled back.

The following sections show how to use each of the different elements of PLVlog.

21.1.1 Controlling the Log Action

PLVlog writes information to the log only when it is turned on. The package provides a standard PL/Vision toggle to control the action of the logging mechanism. The programs comprising this toggle are:

```
PROCEDURE turn_on;
PROCEDURE turn_off;
FUNCTION logging RETURN BOOLEAN;
```

One procedure to turn on logging, another to turn it off, and a final function to indicate the current state of affairs (for completeness and politeness). All the two procedures do is set the value of a private Boolean variable, but by correctly applying that Boolean inside an IF statement in PLVlog, the package's user gets to fine−tune the package's behavior.

Why would you turn off logging? You might be executing a production program for a one−time batch of millions of records. If you use the normal logging built into your program, you end up with millions of lines in the log. You can't afford the performance or disk overhead, and you don't really care about the log information for this run. So instead of modifying your program (supposed it is called **`analyze_sales`**), you simply turn off logging as shown below:

```
SQL> exec PLVlog.turn_off;
SQL> exec analyze_sales;
```

21.1.2 Writing to the Log

You use the **`put_line`** procedure to write a line to the log. There are two overloaded versions of **`put_line`**; the headers are shown below:

```
PROCEDURE put_line
   (context_in IN VARCHAR2,
    code_in IN INTEGER,
    string_in IN VARCHAR2 := NULL,
    create_by_in IN VARCHAR2 := USER,
    rb_to_in IN VARCHAR2 := c_default,
    override_in IN BOOLEAN := FALSE);

PROCEDURE put_line
   (string_in IN VARCHAR2,
    rb_to_in IN VARCHAR2 := c_default,
    override_in IN BOOLEAN := FALSE);
```

The second version of **`put_line`** −− the one with only three arguments −− simply calls the first version with null values for the other arguments. It is provided for convenience, when you simply want to log a string of text and not bother with all the other values. The full set of arguments to **`put_line`** is explained in the following:

`context_in`
: The context from which the log was called, which usually means the program unit in which **`put_line`** was called. You must supply a context.

code_in
: A numeric code to be stored with the text message. This would usually be the error number, but it could be anything. You must supply an integer code.

string_in
: The text to be stored in the log. It can be up to 2000 bytes in length.

create_by_in
: The name of the user or account that created the line in the log. The default is provided by the builtin USER function.

rb_to_in
: The name of the savepoint to which you want a rollback to occur. The default value is the currently defined default savepoint in PLVlog. This argument is used when you have called **`do_rollback`**;

see Section 21.1.5, "Rolling Back with PLVlog" for more information on this argument.

override_in
: If you pass TRUE for this argument, then the string is written to the log even if logging is otherwise turned off.

If you have turned off logging with a call to PLV**log.turn_off**, a call to **put_line** will not add any information to the log (unless you override that state of affairs).

PLVlog.put_line is called from several different PL/Vision packages; I'll use those programs as examples so you can see how and why **put_line** has so many arguments.

21.1.2.1 Using put_line

1. In the PLVcmt package, you can ask that commits be recorded to the log by calling the PLVcmt.log procedure. When logging is turned on, the **put_line** program is called at commit time, as shown below:

   ```
   IF logging
   THEN
      PLVlog.put_line
         ('PLVcmt', 0,
          v_message || ' ' || context_in,
          PLVlog.c_noaction,
          TRUE);
   END IF;
   ```

   I provide the context, a generic PLVcmt, an INTEGER code of 0 to indicate success, and the message constructed from various components. Since I am committing, I do not want any rollback activity to occur, so I request "no action" for the savepoint. Finally, I want to record this information even if the user has turned off logging, so I pass TRUE to override the current setting.

2. The PLVexc exception−handling package uses PLVlog to record errors in one's application. In the code shown below, **PLVlog.put_line** is only called when the user has requested that an error be recorded. So I again override the current log setting to make sure that this information is written to the log. I also pass in the context, error code and error message associated with the exception. Finally, I do not provide a value for the savepoint argument; I simply rely on the default handling. If the user wants a special rollback action, she provides direction with calls to the appropriate PLVlog programs directly.

   ```
   IF recording_exception (handle_action_in)
   THEN
      PLVlog.put_line
         (context_in, err_code_in, string_in,
          override_in => TRUE);
   END IF;
   ```

3. Sometimes I simply want to take advantage of the PLVlog architecture to allow me to write a string out to a log without having to fuss with all those arguments. In the following code fragment, I request that output be directed to a database table and then execute a series of procedures, documenting their completion with calls to **PLVlog.put_line**:

   ```
   BEGIN
      PLVlog.to_dbtab;
   ```

```
      calculate_gross_sales (1995, sales$);
      PLVlog.put_line ('gross sales', sales$);

      calculate_office_expenses (1995, offexp$);
      PLVlog.put_line ('office expenses', offexp$);

      calculate_empl_comp(1995, emplcomp$);
      PLVlog.put_line ('employee compensation', emplcomp$);

      PLVcmt.perform_commit;
   END;
```

You can use all the arguments or only the minimum. Either way, the **`put_line`** logging mechanism should come in very handy. You could even use PLVlog as a lazy way to write information to a PL/SQL table!

21.1.3 Selecting the Log Type

The PLVlog package was designed to offer maximum flexibility for logging. The usual logging mechanism requires that you write a record to a database table. This is certainly supported by PLVlog, but you can also select a different type of log, as appropriate to your needs. The log types supported by PLVlog are: database table, PL/SQL table, operating system file, and standard output (your monitor). These different types and how to work with them are explained in the following sections. PLVlog does provide a single program, **`sendto`**, that you can use to set the log type. The header for **`sendto`** is:

```
PROCEDURE sendto (type_in IN VARCHAR2, file_in IN VARCHAR2 := NULL);
```

where **`type_in`** is the type of log and **`file_in`** is the name of the operating system file (relevant only if you are setting the log type to **`PLV.file`**). The **`type_in`** must be one of the repository constants defined in the PLV package: **`PLV.dbtab`**, **`PLV.pstab`**, **`PLV.file`**, or **`PLV.stdout`**.

In addition to this generic type−setting procedure, PLVlog offers a procedure that specifically sets the log type. So to set the log type to a database table, you can enter either of these commands:

```
SQL> exec PLVlog.sendto (PLV.dbtab)
```

or:

```
SQL> exec PLVlog.to_dbtab
```

To set the log type to a PL/SQL table, you can enter either of these commands:

```
SQL> exec PLVlog.sendto (PLV.pstab)
```

or:

```
SQL> exec PLVlog.to_pstab
```

To set the log type to the **`mylog.txt`** operating system file, you can enter either of these commands:

```
SQL> exec PLVlog.sendto (PLV.file, 'mylog.txt')
```

or:

```
SQL> exec PLVlog.to_file ('mylog.txt')
```

Finally, to set the log type to standard output, you can enter either of these commands:

```
SQL> exec PLVlog.sendto (PLV.stdout)
```

or:

```
SQL> exec PLVlog.to_stdout
```

When you are working with database or PL/SQL table logs, there are some other programs in PLVlog that you may find useful, as I describe below.

21.1.3.1 Defining the database log table

If you do decide to write the log to a database table, you can either use the default log table or specify a different table. When PLVlog performs an insert to the log, it uses dynamic SQL (PLVdyn), so it can construct the INSERT statement from your inputs. This approach allows you to use PLVlog for different purposes. You can even use PLVlog within the same application and Oracle connection to write to different logs!

PL/Vision provides a default log table, whose structure is shown below:

```
CREATE TABLE PLV_log
   (context VARCHAR2(100),
    code INTEGER,
    text VARCHAR2(2000),
    create_ts DATE,
    create_by VARCHAR2(100));
```

where **`context`** is the context in which the log entry was made. This might be a program name or a section within a program or an action. The **`code`** column is a numeric code. This could be an error code or a number in use in the application. The **`text`** column contains a (possibly) long line of text. The **`create_ts`** and **`create_by`** columns provide a user and date/time audit for the creation of the log entry.

PLVlog provides the **`set_dbtab`** procedure to change the name of the table and/or the names of each of the columns in the log table. The header for **`set_dbtab`** is:

```
PROCEDURE set_dbtab
   (table_in IN VARCHAR2 := 'PLV_log',
    context_col_in IN VARCHAR2 := 'context',
    code_col_in IN VARCHAR2 := 'code',
    text_col_in IN VARCHAR2 := 'text',
    create_ts_col_in IN VARCHAR2 := 'create_ts',
    create_by_col_in IN VARCHAR2 := 'create_by');
```

As you can see, **`set_dbtab`** provides you with the opportunity to override the default table and column names. Here are two examples of applying **`set_dbtab`** to different application circumstances:

1. Change the name of the table used for logging; the names of the columns remain the default.

   ```
   PLVlog.set_dbtab ('pnl_log');
   ```

2. Change the name of the table and its columns used for logging.

   ```
   PLVlog.set_dbtab
     ('pnl_log', 'progname', 'err_code',
      'err_text', 'inserted_on', 'inserted_by');
   ```

You can use **`set_dbtab`** to change some or all of these names, but remember that the table you select for logging must have at least these five columns with the same datatypes as the default table.

> ***NOTE:*** If you use a database table for your log, remember that you must also execute a commit in your session to save the log. In addition, you often need to issue rollbacks so that transactions in error are not saved along with the log.

21.1.3.2 Closing the file

If you are writing to an operating system file, you need to close that file before you can see log information written to that repository. To close the PLVlog file, use the **`fclose`** procedure:

```
PROCEDURE fclose;
```

You do not have to specify the file name; that information has been stored inside the package.

21.1.4 Managing a PL/SQL Table Log

PLVlog lets you avoid the hassles of writing to a database log during execution of your application by writing instead to a PL/SQL table. This data structure is memory−resident and owned by the session, so you don't have to deal with commits and rollbacks. Just run your application and then either display the log or copy the contents of the log to a database table with the **`ps2db`** procedure (explained below).

To set the log repository to the PL/SQL table, execute either:

```
PLVlog.to_pstab;
```

or:

```
PLVlog.sendto (PLV.pstab);
```

Then, whenever a line is written to the log, it is deposited in the next available row in the PL/SQL table. The log always starts at row 1 and moves sequentially forward.

The PL/SQL table that stores the log information is a table of 2,000−byte strings, defined as follows:

```
pstablog PLVtab.vcmax_table;
```

When a line is written to the table it is formatted as shown below:

```
context_in || c_delim ||
TO_CHAR (code_in) || c_delim ||
string_in || c_delim ||
TO_CHAR (SYSDATE, PLV.datemask) || c_delim ||
create_by_in || c_delim
```

where **`c_delim`** is defined to be CHR(8) (see the explanation of the **`put_line`** procedure for more information about the individual variables in this string of concatenations). This character shows up in Windows as a black box and is a useful delimiter.

21.1.4.1 Counting and clearing the PL/SQL table

When you are using the PL/SQL table for logging, PLVlog provides two other programs with which to manage the table: **`clear_pstab`** and **`pstab_count`**. If you want to make sure that the PL/SQL table−based log is empty, call the **`clear_pstab`** procedure as follows:

```
PLVlog.clear_pstab;
```

If you want to find out the number of rows currently in the log, you can call **`pstab_count`** as shown in the following IF statement (and used in the FOR loop of the **`showlog.sql`** script above):

```
IF PLVlog.pstab_count > 0
THEN
   PLVlog.display;
END IF;
```

You cannot directly access the PL/SQL table log, since the table itself is defined inside the body of the package and is, therefore, private. On the other hand, you can transfer the PL/SQL table to a database table (with the **`ps2db`** procedure). You can also **`get_line`** to retrieve a specific row from the table, as discussed in the next section.

21.1.4.2 Retrieving log data from PL/SQL table

You can unpack a log string in the PL/SQL table into individual variables with the **`get_line`** procedure, whose header is shown below:

```
PROCEDURE get_line
   (row_in IN INTEGER,
    context_out OUT VARCHAR2,
    code_out OUT INTEGER,
    string_out OUT VARCHAR2,
    create_by_out OUT VARCHAR2,
    create_ts_out OUT DATE);
```

You supply the row number of the PL/SQL table in which you are interested, and **`get_line`** returns the different elements of the logged message. This program is used by the **`ps2db`** procedure, and you can use it as well to parse the log information from the PL/SQL table. The following script (stored in the **`showlog.sql`** file on the companion disk) displays all the rows in the log that were entered today for the context CALC_TOTALS:

```
DECLARE
   v_context PLV_log.context%TYPE;
   v_code PLV_log.code%TYPE;
   v_text PLV_log.text%TYPE;
   v_create_by PLV_log.create_by%TYPE;
   v_create_ts PLV_log.create_ts%TYPE;

BEGIN
   /* pstab_count is explained below. */
   FOR row_ind IN 1 .. PLVlog.pstab_count
   LOOP
      PLVlog.get_line
         (row_ind,
          v_context, v_code, v_text,
          v_create_by, v_create_ts);

      IF v_create_ts BETWEEN TRUNC (SYSDATE)
                         AND TRUNC (SYSDATE+1) AND
         v_context = 'CALC_TOTALS'
      THEN
         p.l (v_text);
      END IF;
   END LOOP;
END;
/
```

NOTE: If you are on the lookout for handy, concise string manipulation routines, you might want to examine the body of **`get_line`**. It contains a local procedure called **`get_colval`**, which allows me to cleanly separate out the different elements of my delimited string. It makes use of the **`betwnstr`** function of the PLV package, which in and of itself is a useful extra tool.

21.1.4.3 Transferring a PL/SQL table log to a database table

PLVlog provides the **ps2db** procedure to transfer the contents of the PL/SQL table log to the currently defined database log table. The header for this procedure is:

```
PROCEDURE ps2db;
```

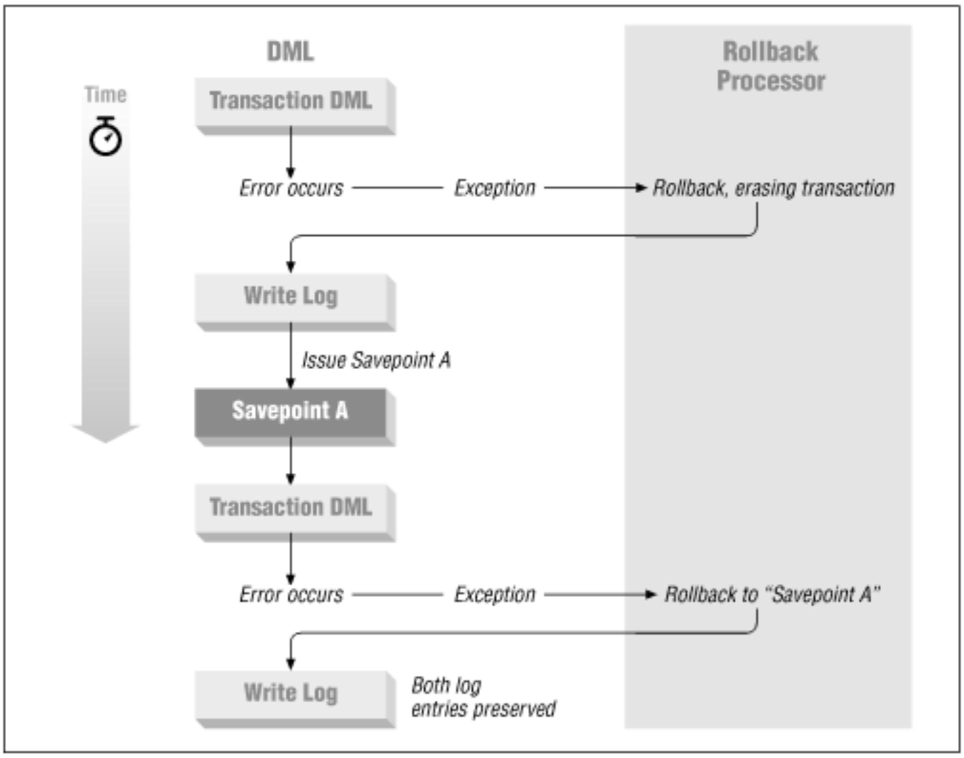
Why would you bother moving the contents of the log from a PL/SQL table to a database table? You might have relied on the PL/SQL table log during the execution of your application so that you didn't have to worry about commit and rollback processing. Now the test session is done and you want to move your log information to "persistent" data. The PL/SQL table goes away when your session ends. If you move the data to a database table, you can examine the contents at your leisure and with the flexibility offered by the SQL language.

21.1.5 Rolling Back with PLVlog

One of the problems with writing log information to a database table is that the information is only available once the new rows are committed to the database. This can be a problem because PLVlog is often employed to track errors −− and in many of these situations, the current transaction has failed or otherwise requires a rollback.

Figure 21.1 illustrates the complexity of maintaining transaction integrity while also preserving records written to a database table log. Transaction A raises an exception. A consequence of the exception is a rollback of the entire transaction. PLVlog is then called to put a line in the log. The application then moves on to the next transaction. If transaction B also raises an exception and issues a rollback, the record written to the log is also erased. So it is necessary to issue a savepoint immediately after the write to the database log. The final complication, however, is that when the rollback of transaction B occurs, it must roll back to the post−log savepoint.

Figure 21.1: Preserving log entries while rolling back transactions



This scenario gives rise to a number of interesting challenges for a generic logging package like PLVlog: how can it avoid saving in−error transactions while saving log information? How can the post−log transaction rollback know about the savepoint? If known, how can this dynamic rollback to a variable savepoint be accomplished? And how can Plvlog make this process as transparent as possible to users of the logging mechanism?

PLVlog offers an architecture to answer these questions. It is complicated and sometimes hard to follow, but it does seem to get the job done.

At a high level, I want to perform steps like this:

```
Rollback−Transaction
Insert−Line−in−Log
Set−Savepoint−to−Preserve−Insert
```

Yet when you examine "Rollback−Transaction" more closely, a question arises: Do I want to perform an unqualified rollback to reject all uncommitted changes, or do I want to roll back to a particular savepoint?

PL/Vision could require that the individual developers who use PLVlog simply handle all of these rollback and savepoint issues. But that approach places a high burden on my users and makes it much less likely that PLVlog will be used to its full potential −− a drawback I take kind of personally. So, instead, PLVlog offers several different toggles, settings, and commands you can use to direct PLVlog to take precisely the right actions in the right order and at the right time.

First, let's go over the way PLVlog performs rollbacks and savepoints, then examine how you can control this aspect of the package's behavior.

21.1.5.1 Toggling rollback activity

If your log repository is an operating system file or PL/SQL table, the concept of a rollback simply doesn't apply. You only have to worry about this complication when you are logging directly to a database table −− which is, after all, a pretty common activity. Suppose, then, that you are using a database table log. PLVlog will still not perform any rollbacks or issue any savepoints unless you explicitly turn on this behavior.

PLVlog offers a standard PL/Vision toggle. This triumvirate of programs is:

```
PROCEDURE do_rollback;
PROCEDURE nodo_rollback;
FUNCTION rolling_back RETURN BOOLEAN;
```

One procedure to turn on rollback and savepoint activity, another to turn it off (the default), and a final function to indicate the current state of affairs. All the two procedures do is set the value of a private Boolean variable, but by correctly applying that Boolean inside an IF statement in PLVlog, the package's user gets to fine−tune the package's behavior.

The discussion in the following sections assumes that at some point in your session before you tried to log activity to the database, you issued this command in SQL*Plus:

```
SQL> exec PLVlog.do_rollback
```

or directly executed this command inside another PL/SQL program like this:

```
BEGIN
   PLVlog.do_rollback;
   transfer_data;
END;
```

21.1.5.2 How PLVlog manages the transaction

Assuming that you have directed PLVlog to help you manage your transaction automatically when information is sent to the database log, let's take a look at how PLVlog handles the complexities.

The best way to explain this process is to show you the relevant part of the body of the **put_line** program. The section below is executed only when logging is turned on or overridden in that call to **put_line**. As you can see, the discussion of rollbacks and this section of code is relevant only when the log type is a database table.

```
1   IF log_type = PLV.dbtab
2   THEN
3      IF rolling_back
4      THEN
5         do_rb (v_message);
6      END IF;
7
8      put_to_db
9         (context_in, code_in, string_in, create_by_in,
10         SYSDATE);
11
12     IF rolling_back
13     THEN
14        PLVrb.set_savepoint (v_savepoint);
15     END IF;
```

Lines 8 through 10 contain the call to a private module, **put_to_db**, that performs the actual INSERT into the specified table. This INSERT is sandwiched between two rollback−related activities. Lines 3 and 12 apply the rollback toggle discussed in the previous section; if you have not requested **PLVlog.do_rollback**, then **rolling_back** returns FALSE and the code on lines 5 and 14 are not executed.

Suppose, however, that you have executed **do_rollback**. Then before a line of data is inserted into your log table, **put_line** executes the **do_rb** program, which is a local module in **put_line** that figures out exactly which type of rollback to execute. This topic is covered in Section 21.1.5.6, "Performing different kinds of rollbacks". After the information is inserted into the log, PLVlog calls the **PLVrb.set_savepoint** to set a savepoint using dynamic PL/SQL. The savepoint used (**v_savepoint**) is the current PLVlog savepoint and is discussed in the next section.

By setting a savepoint immediately after the insert, PLVlog gives you a way to roll back all changes except for the last write to the log table. This savepoint is either a savepoint declared by PLVlog or a savepoint you have defined for PLVlog to use, as is explained below.

21.1.5.3 Types of rollback activity

PLVlog provides a set of constants, shown in the following table, that you can use to modify the way rollbacks occur in the **put_line** procedure.

Constant Name	Value	Description
`c_noaction`	`*NO ROLLBACK*`	Do not perform any kind of rollback.
`c_none`	`*FULL*`	Perform a full rollback (not back to a savepoint).
`c_default`	`*DEFAULT*`	Roll back to the default PLVlog savepoint.
`c_last`	`*PLVRB-LAST*`	Roll back to the last savepoint maintained by the PLVrb package.
`c_PLVlogsp`	`PLVlog_savepoint`	Roll back to the **PLVlog_savepoint**, the initial default savepoint

		for PLVlog.

You have probably noticed that the values for the first three "savepoints" are not valid savepoint names. Only the constant **c_PLVlogsp** can actually be used in a SAVEPOINT command. In fact, the other constants are only used by the **do_rb** program in **put_line** to determine which kind of ROLLBACK action to take. The **do_rb** procedure is explored in more detail in Section 21.1.5.6.

21.1.5.4 Setting the default "rollback to" behavior

The default "rollback to" savepoint of PLVlog is used when the user has turned on rollback activity and the user has not provided an alternative savepoint in the call to **put_line**. This savepoint is then used to determine the type of rollback to execute before inserting a line into the log.

The default savepoint is initialized to the **c_none** constant, which means that a full, unqualified rollback is executed.

To change the default "rollback to" savepoint, you can call any of the programs whose headers are shown below:

```
PROCEDURE rb_to (savepoint_in IN VARCHAR2 := c_none);
PROCEDURE rb_to_last;
PROCEDURE rb_to_default;
```

The procedures **rb_to_last** and **rb_to_default** are simply special cases of the **rb_to** procedure. Let's look at the impact of using these programs with the different savepoint constants and, of course, your own savepoint names.

1. Set the default savepoint to "no action:"

   ```
   PLVlog.rb_to (PLVlog.c_noaction);
   ```

   When you set the "rollback to" savepoint to "no action," no rollback occurs -- even if you have turned on rollbacks with a call to **do_rollback**. You will usually not pass this constant in to **rb_to**. Instead, you might do so in a call to **put_line** so that no rollbacks occur for that single insert.

2. Set the default savepoint to none:

   ```
   PLVlog.rb_to (PLVlog.c_none);
   ```

   or:

   ```
   PLVlog.rb_to;
   ```

   When you set the "rollback to" savepoint to none, you are requesting a full ROLLBACK, unqualified by any savepoint. This causes all uncommitted changes to be erased from your session.

3. Set the default savepoint to the default:

   ```
   PLVlog.rb_to (PLVlog.c_default);
   ```

   or:

   ```
   PLVlog.rb_to_default;
   ```

By setting the "rollback to" savepoint to the default, you actually set the default savepoint to the constant **c_PLVlogsp**. You will usually not pass this constant in to **rb_to**. Instead, you might do so in a call to **put_line** to ensure that the current default savepoint is used for rollback activity (this, in fact, is the default).

4.
   Set the default savepoint to the PLVlog initial value:

   ```
   PLVlog.rb_to (PLVlog.c_PLVlogsp);
   ```

   By setting the "rollback to" savepoint to this constant, you return the default savepoint back to its initial value. You might do this at the start of a process in order to reset the PLVlog package back to its original values.

5.
   Set the default savepoint to the last savepoint issued by PLVrb:

   ```
   PLVlog.rb_to (PLVlog.c_last);
   ```

   or:

   ```
   PLVlog.rb_to_last;
   ```

   By setting the "rollback to" savepoint to this constant, you are coordinating closely with the PLVrb package, which maintains a stack of savepoints. You are indicating that when a rollback occurs, it should only roll back to the last savepoint issued by PLVrb. You should only use this setting if you are rigorous about using the **PLVrb.set_savepoint** procedure whenever you issue a SAVEPOINT.

6.
   Set the default savepoint to a user−defined savepoint indicating the start of a new order transaction:

   ```
   PLVlog.rb_to ('new_order');
   ```

   By setting the savepoint to this string, you set the default behavior of PLVlog to rollback to this savepoint before an insert to the log table.

21.1.5.5 Specifying rollbacks when calling put_line

You can also use the rollback constants when specifying a value for the **rb_to_in** argument of **put_line**. This acts as an override to the default "rollback to" savepoint. Since we've already looked at how **put_line** works, let's now explore how to use the **rb_to_in** argument with these constants to change the rollback behavior for a specific call to **put_line**.

Here, again, is the header for **put_line** (the abbreviated version):

```
PROCEDURE put_line
   (string_in IN VARCHAR2,
    rb_to_in IN VARCHAR2 := c_default,
    override_in IN BOOLEAN := FALSE);
```

The default value for the "rollback to" argument is **c_default**, which means that PLVlog issues a rollback according to the current default setting (explained in the next section). The following examples show the alternatives to this behavior:

1.

Put a line in the log but do not perform any rollback activity, regardless of the value returned by **`PLVlog.rolling_back`**.

```
PLVlog.put_line (v_err_msg, PLVlog.c_noaction);
```

2. Put a line in the log and request a rollback to the last savepoint issued by PLVrb.

```
PLVlog.put_line (v_err_msg, PLVlog.c_last);
```

This setting ties in PLVlog as tightly as possible with the use of PLVrb to manage and issue savepoints in your application.

3. Put a line in the log and request a full, unqualified rollback.

```
PLVlog.put_line (v_err_msg, PLVlog.c_none);
```

The constant is called **`c_none`** because PLVlog does not roll back to any savepoint, instead it just issues a ROLLBACK command.

4. Put a line in the log, rolling back all changes made since the last SAVEPOINT TO the standard PLVlog savepoint.

```
PLVlog.put_line (v_err_msg, PLVlog.c_PLVlogsp);
```

This is not the same as rolling back to the default savepoint, because you may have changed the default with a call to one of the **`rb_to`** programs documented in the next section.

In all of the above examples, the rollback activity described applies only to that single call to **`PLVlog.put_line`**. Whenever you call **`put_line`** and do not provide a value for the **`rb_to_in`** argument, PLVlog relies on the default activity you have previously defined (explained below).

21.1.5.6 Performing different kinds of rollbacks

PLVlog takes a different rollback action depending on the default savepoint value. This logic is encapsulated in the local **`do_rb`** procedure inside **`put_line`**:

```
PROCEDURE do_rb (msg_in IN VARCHAR2)
IS
   v_sp PLV.plsql_identifier%TYPE := v_rb_to;
BEGIN
   IF rb_to_in != c_default
   THEN
      v_sp := rb_to_in;
   END IF;

   IF v_sp = c_noaction
   THEN
      NULL;

   ELSIF v_sp = c_none OR v_sp IS NULL
   THEN
      PLVrb.perform_rollback (msg_in);

   ELSIF v_sp = c_default
   THEN
      PLVrb.rollback_to (v_sp_PLVlog, msg_in);
   ELSIF v_sp = c_last
   THEN
      PLVrb.rb_to_last (msg_in);

   ELSE
      PLVrb.rollback_to (v_sp, msg_in);
   END IF;
END;
```

Allow me to translate:

- If you have specified (through the default savepoint value stored in **`v_savepoint`**) "no action," then don't do anything.
- If you set the default to none or if the current default savepoint itself is NULL, perform a full, unqualified rollback with a call to **`PLVrb.perform_rollback`**.
- If the current "rollback to" savepoint is the default, roll back to the initial default value for PLVlog: **`PLVlog_savepoint`**.
- If you have requested a rollback to the last savepoint issued by PLVrb, PLVlog calls the corresponding **`PLVrb.rb_to_last`** procedure to implement precisely that functionality.

Finally, for any other savepoint values, PLVlog requests that PLVrb roll back uncommitted changes to that savepoint string.

21.1.5.7 Setting the post−insert savepoint

After the insert to the database log has taken place, PLVlog issues a savepoint if **`PLVlog.rolling_back`** returns TRUE. The purpose of this savepoint is to preserve the new log record −− even if a rollback comes surging back from activity after the call to the **`PLVlog.put_line`** procedure.

The post−insert savepoint of PLVlog (referred to for the rest of this section simply as "the savepoint") is initialized to the constant **`c_PLVlogsp`**, the predefined savepoint of the package.

You can change this savepoint with a call to the **`set_sp`** procedure, whose header is:

```
PROCEDURE set_sp (savepoint_in IN VARCHAR2);
```

Savepoint Argument	Post−Insert Savepoint Set To
`c_none`	NULL; no savepoint is issued after the insert.
`c_noaction`	NULL; no savepoint is issued after the insert.
`c_last`	**`PLVrb.lastsp`**; this function returns the last savepoint issued by PLVrb.
`c_default`	**`c_PLVlogsp`**; the initial value and the default value for the PLVlog package.
others	The value provided in the argument. This string must be a valid savepoint, that is, a valid PL/SQL identifier −− no more than 30 characters starting with a letter.

21.1.6 Displaying the Log

PLVlog provides a single display procedure to display the contents of the log, whether the log is stored in the database or in a PL/SQL table. If the log is in a database table, PLVlog makes use of the **`PLVdyn.intab`** procedure. If the log is in a PL/SQL table, PLVlog uses the **`PLVtab.display`** procedure. Here is the header for the **`display`** program:

```
          PROCEDURE display (header_in IN VARCHAR2 := 'PL/Vision Log');
```

The header, which is optional, is used in the PL/SQL table display, but not for the display of the database table.

Special Notes on PLVlog

Here are some factors to consider when working with PLVlog:

- The maximum size of the text information in the log is 2,000 characters.
- At runtime you can decide on the name of the database log table and its columns, but that table must have columns for the context, code, text line, date, and user.
- You cannot write log information to an operating system file unless you are running Oracle7 Server Release 7.3 and the corresponding Release 2.3 of PL/SQL.
- One other possibility for log output is a DBMS_PIPE pipe. This has as the same advantages of a PL/SQL table (no need to commit) and, in addition, allows the log data to be shared across sessions.

Here is an example of using the **`display`** procedure: set the log repository to the PL/SQL table, execute the application, and then display the log.

```
SQL> execute PLVlog.to_pstab
SQL> start myapp
SQL> execute PLVlog.display
Contents of PL/Vision Log
proc1 ORA−01476: divisor is equal to zero 12/19/1995 11:41:30
proc2 ORA−06502: PL/SQL: num or val error 12/19/1995 11:41:30
```

You can also view the contents of the log with SQL if the log is a database table. In the following scenario, I set the log repository to the database table, execute the application, and then display the log.

```
SQL> execute PLVlog.to_dbtab
SQL> start myapp
SQL> start inlog
CONTEXT CODE    TEXT                          CREATE_TS
------- ----- ------------------------------ -----------------
proc1   −1476 ORA−01476: divisor is equal to 12/19/95 114301
                         zero
proc2   −6502 ORA−06502: PL/SQL: numeric or  12/19/95 114301
                         value error
```

where the **`inlog.sql`** script is as follows:

```
SET ARRAYSIZE 10
column code format 999999
column context format a12
column create_ts format a17
column text format a30
SELECT context, code, text,
       TO_CHAR (create_ts, 'mm/dd/yy hhmiss') create_ts
  FROM PLV_log
 ORDER BY create_ts;
```

Notice that this script uses the default logging table, **`PLV_log`**. You can, as noted above, override this default with your own table and column names.

PREVIOUS HOME NEXT
20.2 PLVrb: Performing Rollbacks BOOK INDEX 21.2 PLVtrc: Tracing Execution of PL/SQL Programs

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

21.2 PLVtrc: Tracing Execution of PL/SQL Programs

The PLVtrc (PL/Vision TRaCe) package offers a generic trace facility for PL/SQL applications. It is especially useful if you do not have access to a source debugger for PL/SQL. It is also used by PLVexc to record the currently executing PL/SQL program unit when an exception occurs.

The PLVtrc package offers several capabilities:

- Provides a flexible execution trace facility. It inserts trace points anywhere in your code; these fire and display (or write to a log) only if you explicitly turn on the trace. Thus, your trace can remain in place even in production code.
- Parses the string returned by a call to the DBMS_UTILITY.FORMAT_CALL_STACK function. You can specify retrieval of the *n*th module in the stack and use this information in your own auditing and trace activities.
- Is similar to the **p** package. PLVtrc offers a heavily overloaded interface to the PUT_LINE procedure of DBMS_OUTPUT. (In fact, PLVtrc calls the **p.l** procedure to generate its output.)

The elements provided by PLVtrc can be broken down into three areas: output control, module tracking, and activity trace. They are explored in the following sections.

21.2.1 Directing Output from PLVtrc

As with many other PL/Vision packages, PLVtrc allows you to programmatically control its behavior, in this case its output. PLVtrc offers three different types of output:

- Display the trace message to the screen. If not turned on, all calls to PLVtrc programs are ignored.
- Log the trace message to the PL/Vision log. If tracing is activated, this feature also writes a line out to the current PLVlog repository.
- Display the currently executing module based on the PL/SQL FORMAT_CALL_STACK function. If tracing is activated, this feature adds the current module to the display.

PLVtrc offers a separate toggle for each of these aspects of output. Here are the programs to turn the trace facility itself on and off:

```
PROCEDURE turn_on;
PROCEDURE turn_off;
FUNCTION tracing RETURN BOOLEAN;
```

To activate the trace facility, enter:

```
SQL> execute PLVtrc.turn_on;
```

To de−activate the trace facility enter:

```
SQL> execute PLVtrc.turn_off;
```

To control logging of trace messages using the PLVlog package, PLVtrc offers these programs:

```
PROCEDURE log;
PROCEDURE nolog;
FUNCTION logging RETURN BOOLEAN;
```

This command turns on logging of trace message:

```
SQL> execute PLVtrc.log;
```

To turn off logging, enter:

```
SQL> execute PLVtrc.nolog;
```

You can also request that when trace information is displayed the current module is included in the message. You control the inclusion of that data with this toggle:

```
PROCEDURE dispmod;
PROCEDURE nodispmod;
FUNCTION displaying_module RETURN BOOLEAN;
```

So to include the module in trace messages, enter:

```
SQL> execute PLVtrc.dispmod;
```

To ignore the module name enter:

```
SQL> execute PLVtrc.nodispmod;
```

To turn on all these options, you need to execute all three "on" procedures:

```
SQL> exec PLVtrc.dispmod;
SQL> exec PLVtrc.log;
SQL> exec PLVtrc.turn_on;
```

The order in which you call these toggle programs is not important. Just remember that you cannot write information to the log or display the current module unless the overall trace is turned on.

In all examples above I have shown the syntax for executing the programs from within SQL*Plus. You can also call these programs from within a PL/SQL program, in which case you would *not* use the execute command.

21.2.2 Accessing the PL/SQL Call Stack

The PLVtrc package provides you with the ability to access and parse the call stack maintained by the PL/SQL runtime engine. This call stack is available with a call to the DBMS_UTILITY.FORMAT_CALL_STACK function. Here is an example of the string returned by this function:

```
----- PL/SQL Call Stack -----
object      line    object
handle      number  name
88ce3f74         8  package STEVEN.DISPCSTK
88e49fc4         2  function STEVEN.COMPANY_TYPE
88e384c8         1  procedure STEVEN.CALC_PROFITS
88e3823c         1  procedure STEVEN.CALC_TOTALS
88e49390         1  procedure STEVEN.CALC_NET_WORTH
88e2bd20         1  anonymous block
```

The string actually contains many newline characters (you can find these by searching for CHR(10) with the INSTR function). It is designed for easy display, but not easy manipulation within a programmatic setting.

The PLVtrc package offers two programs to access this PL/SQL call stack:

ps_callstack
: Returns the same string returned by the FORMAT_CALL_STACK function. It is provided for consistency and to save you some typing.

ps_module
: Returns the *n*th module in the PL/SQL call stack, with the default being the "most recent module," (i.e., the program that was active before the **`PLVtrc.module`** function was called.) This, by the way, is the *second* module in the stack.

> ***NOTE:*** One big problem with FORMAT_CALL_STACK is that it will not provide the name of the current program within a package. If you are executing a standalone function or procedure, FORMAT_CALL_STACK shows you its name. If you are running a function within a package, however, it only shows you the package name. If your code design is package−driven, this fact renders the FORMAT_CALL_STACK function largely irrelevant. This shortcoming is the main reason that PLVtrc also maintains its own program call stack in a PL/Vision stack.

21.2.3 Tracing Code Execution

PLVtrc offers two programs to trace the execution of your code: **`action`** and **`show`**. The **`action`** program's header is:

```
PROCEDURE action
   (string_in IN VARCHAR2 := NULL,
    counter_in IN INTEGER := NULL,
    prefix_in IN VARCHAR2 := NULL);
```

You pass a string, a numeric counter or indicator, and another string that is used as a prefix on the trace message. The action procedure is used by **`startup`**, **`terminate`**, and the other activity trace module, **`show`**. It is, in other words, the lowest−level trace procedure.

The **`show`** procedure is heavily overloaded. Like the **`p.l`** procedure, the **`show`** program comes in many flavors of argument combinations, as shown in the list below. This is done to make it easy for you to pass different combinations of data for display without having to perform TO_CHAR conversions and concatenations.

The following datatype combinations are supported by **`PLVtrc.show`**:

Single−value	Double−value	Triple−value
string	string, date	string, number, number
date	string, number	string, number, date

number	string, boolean	string, number, boolean
boolean		

Here are the headers for the single−value **`show`** procedures:

```
PROCEDURE show (stg1 IN VARCHAR2);
PROCEDURE show (date1 IN DATE, mask_in IN VARCHAR2 := PLV.datemask);
PROCEDURE show (bool1 IN BOOLEAN);
PROCEDURE show (num1 IN NUMBER);
```

Here are the headers for the double−value **`show`** procedures:

```
PROCEDURE show (stg1 IN VARCHAR2, num1 IN NUMBER);
PROCEDURE show
   (stg1 IN VARCHAR2,
    date1 IN DATE, mask_in IN VARCHAR2 := PLV.datemask);
PROCEDURE show (stg1 IN VARCHAR2, bool1 IN BOOLEAN);
```

Here are the headers for the triple−value **`show`** procedures:

```
PROCEDURE show
   (stg1 IN VARCHAR2, num1 IN NUMBER, num2 IN NUMBER);
PROCEDURE show
   (stg1 IN VARCHAR2, num1 IN NUMBER,
    date1 IN DATE, mask_in IN VARCHAR2 := PLV.datemask);
PROCEDURE show
   (stg1 IN VARCHAR2, num1 IN NUMBER, bool1 IN BOOLEAN);
```

You can place calls to both **`action`** and **`show`** in your programs. No output is generated from these message lines until you turn on the trace. And since there is very little overhead involved in calling these programs, you can leave the trace in your code even when it goes into production status. When you have to debug the code, you simply call **`PLVtrc.turn_on`**, run the application, and you have a wealth of information available to you.

21.2.4 The PLVtrc Program Stack

PLVtrc offers two programs to build its own program execution stack: **`startup`** and **`terminate`**. You can also get information about the current and previous modules of the stack.

The PLVtrc call stack operations provide two key advantages over the builtin:

1. The call stack contains the names of the specific programs being executed (or whatever strings you pass to represent the names of programs).

2. You can call **`startup`** at any point, so you can give names in your call stack to anonymous blocks as well as named modules. Just don't forget to **`terminate`** if you run **`startup`**.

21.2.4.1 startup

You should call **`PLVtrc.startup`** as the first line in the body of your program. Its header is:

```
PROCEDURE startup
   (module_in IN VARCHAR2, string_in IN VARCHAR2 := NULL);
```

You provide the module name or abbreviation or whatever string you want to record as representing the program. You can also pass in a second string argument. This value is displayed or logged by PLVtrc, according to the toggle settings. This second argument allows you to pass variable data into the trace.

21.2.4.2 terminate

The **terminate** program performs a task opposite that of **startup**: it pops off the stack the most recently pushed module and sets the previous module variable. The header for **terminate** is:

```
PROCEDURE terminate (string_in IN VARCHAR2 := NULL);
```

As with **startup**, you can provide a string to be displayed or logged, depending on the status of the PLVtrc toggles. You should call **PLVtrc.terminate** as the last executable statement in your procedure, and immediately before your RETURN statement in a function.

You should also call **PLVtrc.terminate** in each of your exception handlers in a PL/SQL block where startup was called. Otherwise the enclosing module will not be popped off the stack when the block fails.

If you use one of the high−level handlers of PLVexc to handle your exception, however, you do not have to −− and should not −− call **terminate**. Those handlers do that for you.

21.2.4.3 Current module

Each time **PLVtrc.startup** is executed, it pushes the current module onto the PLVtrc execution stack and sets the current module to the first argument in the call to **startup**.

You can obtain the name of the current module in the PLVtrc environment by calling the **currmod** function, whose header is:

```
FUNCTION currmod RETURN VARCHAR2;
```

21.2.4.4 Previous module

To see the name of the previous module, you can call the **prevmod** function, whose header is:

```
FUNCTION prevmod RETURN VARCHAR2;
```

The PLVexc package makes use of this function so that it can record the program in which an exception was raised. You might not have too much use for **prevmod**.

See Section 21.2.5, "Using PLVtrc" to see how you put these pieces together.

21.2.4.5 Emptying the stack

The **clearecs** procedure empties the execution stack by recreating it. The header for this procedure is:

```
PROCEDURE clearecs;
```

You will want to use this program when you have finished running a test and you want to make sure that there aren't any extraneous module names left on the stack.

21.2.4.6 Displaying the stack

You can display the contents of the stack with a call to **showecs**; the header is:

```
PROCEDURE showecs;
```

This program, in turn, calls **`PLVlst.display`** to display the contents of the list, which comprise the underlying data structure for the stack (implemented, actually, with the PLVstk package −− an interesting exercise in code layering).

21.2.4.7 Retrieving stack contents

If you do not want to directly display the PLVtrc stack, you can extract it as a string in much the same format as that provided by the builtin FORMAT_CALL_STACK function with the **`ecs_string`** function. Its header is:

```
FUNCTION ecs_string RETURN VARCHAR2;
```

Each module name in the call stack is separated by a newline character.

You can also retrieve a single module from the stack with the **`module`** procedure. This program's header is:

```
FUNCTION module (pos_in IN INTEGER := c_top_pos)
   RETURN VARCHAR2;
```

where **`pos_in`** is the position in the stack in which you are interested. The current program is stored in the top−most position of 0 (actually not yet on the call stack) and is encapsulated in the package constant, **`c_top_pos`**. To obtain the name of the module that called the current program, you would pass in a position of 1.

21.2.5 Using PLVtrc

The following examples show you how to use the different elements of the PLVtrc package.

1. Use the **`startup`** and **`terminate`** procedures in my procedure to integrate it into the PLVtrc call stack. In addition, handle exceptions using the PLVexc component (which also performs a **`terminate`**).

   ```
   CREATE OR REPLACE PROCEDURE proc (val in number) IS
   BEGIN
      PLVtrc.startup ('proc');
      IF 1/val > 1 THEN NULL; END IF;
      PLVtrc.terminate;
   EXCEPTION
      WHEN OTHERS THEN PLVexc.rec_continue;
   END;
   /
   ```

2. Show the employee name and date before giving them a name.

   ```
   FOR emp_rec IN emp_cur
   LOOP
      PLVtrc.show (emp_rec.ename, emp_rec.hiredate);
      give_raise (emp_rec.empno);
   END LOOP;
   ```

   Remember: this call to **`show`** will not actually generate any output unless you turn on at least one of the trace features.

3. Use the **`startup`** and **`terminate`** procedures to track execution of a nested, anonymous block. Notice that in the exception section, I explicitly call terminate for the NO_DATA_FOUND exception.

For all other errors, I let the PLVexc package handle the **`PLVtrc.terminate`** and the error as well.

```
PROCEDURE annual_calcs (val in number)
IS
BEGIN
   PLVtrc.startup ('proc');

   calc_gross_revenue;
   BEGIN
      PLVtrc.startup ('analyze');
      calc_rev_distribution;
      PLVtrc.terminate;
   EXCEPTION
      WHEN OTHERS THEN PLVexc.halt;
   END;
   call_profits;
   PLVtrc.terminate;

EXCEPTION
   WHEN NO_DATA_FOUND
   THEN
      PLVtrc.terminate;
      p.l ('Invalid value: ' || TO_CHAR (val);

   WHEN OTHERS
   THEN
      /* This program calls PLVtrc.terminate */
      PLVexc.rec_continue;
END;
/
```

PREVIOUS 21.1 PLVlog: Logging Activity in PL/SQL Programs

HOME BOOK INDEX

NEXT 22. Exception Handling

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

22. Exception Handling

Contents:

The PLVexc (PL/Vision EXCeption handling) package provides a powerful, plug−and−play component to perform exception handling in your PL/SQL programs. It makes use of PLVlog to automatically write errors to the log of your choice (database table, PL/SQL table, etc.). It offers high−level exception handler programs so that individual developers can simply call a procedure that describes the desired action, such as "record and continue," and PLVexc figures out what to do.

This chapter first analyzes the need for exception handling and the traditional solutions in a PL/SQL environment. It then presents the elements of the PLVexc package, along with the information you need to apply this functionality in your own programs. Finally, the chapter explores the implementation of PLVexc in two phases.

22.1 The Challenge of Exception Handling

Do you test your own programs? If you do, I am willing to wager large sums of money that your users feel as if *they* are the ones doing the testing. Authors of a program cannot find all the bugs in their software. They have, in fact, an uncanny ability to unconsciously follow a path through their code that avoids all bugs and logical holes.Programmers cannot be responsible for testing their own code.

Tightly linked to proper testing is proper error handling. Even if a program does not have actual bugs, it does need to handle abnormal conditions gracefully. Yet exception handling is the last thing any of us wants to worry about when we build our PL/SQL programs. To recognize the need to write exception handlers is to acknowledge that things might go wrong. Who can afford the time and resources to focus on the negative? It is all we can do to get our programs to work properly under normal circumstances −− to conform, in other words, to the specifications. How many times do you hear others (never yourself, right?) say things like this: "After I get my program to work, I'll go back and put in the exception handlers." Of course, no one ever has the time to "go back."

Anticipating and handling the problems in one's program is crucial to writing a robust application. PL/SQL offers a very powerful architecture for dealing with abnormal conditions: the exception section and exception handlers. Yet this architecture can also be difficult to use, particularly in a manner that ensures consistent error handling across an entire application. This difficulty, combined with the importance of getting it right, makes exception handling an excellent candidate for the use of a plug−and−play component.

22.1.1 A Package−Based Solution

A plug−and−play component, based in PL/SQL package technology, can hide the complexities and difficulties of exception handling. It can provide a declarative interface to both responding to and logging problems encountered in an application. With this declarative approach, programmers can, in the exception section, state what they want the program to do and leave it to the underlying engine (the package) to figure out how to get the job done. This is very similar to the theory and practices of the SQL language.

To give you a taste of what is possible, the following exception section has two different exception handlers: one for NO_DATA_FOUND and one for all other exceptions. When NO_DATA_FOUND is raised, a message is displayed to the user, the error is recorded, and then the program is halted. When any other error occurs, that error is recorded and then processing continues.

```
EXCEPTION
   WHEN NO_DATA_FOUND
   THEN
      PLVexc.recNstop ('Company has not been located.');
   WHEN OTHERS
   THEN
      PLVexc.recNgo;
END;
```

Both of these exception handlers make use of the PLVexc package. This package automatically records the current program, user, error number, and message. It relies on the PLVlog package so that the recording can take place to a database table or to a PL/SQL table. It performs rollbacks to the last savepoint if requested so that the current transaction is "erased," but the write to the log table is preserved.

There is, in other words, a lot going on when a developer makes a simple, high−level call to a PLVexc error handling procedure. Yet all that activity is rendered invisible by the package. All the user of PLVexc has to know is what kind of action he wants to perform. Ignore the error? Record and continue? Record and halt? Just tell PLVexc what it is you want to do and let the component do the rest.

The PLVexc package is the best example in PL/Vision of a plug−and−play component, prebuilt code you can plug into your own programs, instantly improving the functionality, reliability, and flexibility of your own applications. In this chapter, I'll first show you how to use PLVexc. Then I'll present the implementation of PLVexc in two stages: the first version I built and the second, "final" version of the package you will find on the disk.

I show you the two stages of development and design of PLVexc so that you can follow the thought process I used to move from a specific problem to an increasingly generalized solution. Learning how to use PLVexc in your environment is important, but even more important and more difficult is to develop your own analytical and programming skills so that you can build your own plug−and−play package−based components.

22.1.2 Package−Based Exceptions

The PLVexc package provides several predefined exceptions for you to use and also to provide a model for adding additional exceptions in the same way to PLVexc. These exceptions are:

NO_SUCH_TABLE
ORA−00942: table or view does not exist

SNAPSHOT_TOO_OLD
ORA−01555: snapshot too old (rollback segment too small)

A third exception, **`process_halted`**, is used in the bailout feature and is discussed later in this chapter.

The code required to declare these exceptions is as follows:

```
no_such_table EXCEPTION;
PRAGMA EXCEPTION_INIT (no_such_table, −942);

snapshot_too_old EXCEPTION;
PRAGMA EXCEPTION_INIT (snapshot_too_old, −1555);
```

Why does PLVexc predefine these exceptions (and invite you to add more)? So every developer in your organization does not have to take these steps over and over again. Specifically, to achieve:

- *Consistent error handling in the application:* developers will not define their own named exceptions willy−nilly throughout their programs. They make use of existing names for exceptions. Instead of coding their own handlers for WHEN OTHERS, they can use the prebuilt program, **`display_error`**.

- *Less buggy code:* Novice and intermediate programmers do not have to deal with the EXCEPTION_INIT pragma and other complicated issues related to exception handlers. With the package, these details are shielded from the developer.

Once such exceptions are defined in a package, individual developers no longer need to know about the specific error numbers, nor do they have to be hassled with EXCEPTION_INIT syntax.

22.1.3 Impact of Predefined Exceptions

Compare the two PL/SQL blocks below. The first demonstrates the kind of code one would have to write without the predefined exceptions as found in PLVexc.

```
DECLARE
   no_such_table EXCEPTION;
   PRAGMA EXCEPTION_INIT (no_such_table, −942);
BEGIN
   perform_action;
EXCEPTION
   WHEN no_such_table
   THEN
      do_something;
END;
```

The second block shows how one's code would look with PLVexc in place:

```
BEGIN
   perform_action;
EXCEPTION
   WHEN PLVexc.no_such_table
   THEN
      PLVexc.handle
        ('action', SQLCODE, PLVexc.c_recNstop);
END;
```

No declaration section required, no need for the developer to write all that extra code. Just pick from the package−based selections and keep on developing! The benefits of this approach go well beyond short−term productivity gains. If all developers work from this predefined set of exceptions, overall application code volume is reduced, and along with that the number of bugs that are introduced and then tested *out* of the application.

The PLVexc package contains only two predefined exceptions. When you apply this technique into your own environment you will undoubtedly want to add to this section any of the unnamed system exceptions your developers will encounter routinely. And even if you don't think of them all before development begins, you can always add to the set of exceptions as you proceed. Simply set as a guideline for programmers that they never use the EXCEPTION_INIT pragma in their code. Instead, they take the time to add that exception to the package and then reference the package−based exception.

22.1.4 Exception Handling Actions

One of the major improvements offered by PLVexc is the ability for a developer to simply state the kind of action needed in a particular exception handler. You can do this in two ways: pass the action as the third argument to the low−level handle procedure or call a high−level handler whose very name encapsulates the action.

PLVexc supports four different exception−handling actions. These are presented in the table below, along with the corresponding packaged constants used in the call to handle and the corresponding high−level handler program.

Action	Constant	Handler Program	Description
Continue processing	`c_go`	`go`	Continue processing; do not record the error. This is the equivalent of WHEN OTHERS THEN NULL, which means "ignore this error."
Record and then continue	`c_recNgo`	`recNgo`	Record the error and then continue processing. This action would be appropriate when the exception affects the current block but is not severe enough to stop the entire session.
Halt processing	`c_stop`	`stop`	Stop processing; do not record the error. This action causes PLVexc to raise the **`process_halted`** exception. This action would be appropriate when the exception is so severe (either in terms of the database or the application) that it requires termination of the entire session.
Record and then halt processing	`c_recNstop`	`recNstop`	Record the error and then stop processing. This action causes PLVexc to raise the **`process_halted`** exception, as with **`c_stop`**.

By providing named constants, users of PLVexc do not have to be aware of the specific literal values used by PLVexc. This advantage is even greater when calling the **`stop`** or **`recNgo`** programs. Instead of passing cryptic acronyms, users can rely on named elements to make their code self−documenting and easy to maintain. Most importantly, users do not have to be aware of either how the recording process takes place or how the program is halted. They can just make the request.

22.1.5 What About the Reraise Action?

Another kind of action you might want to take is to reraise the same exception that brought you into the exception section. You can do this in PL/SQL by issuing an unqualified RAISE statement, as shown below:

```
EXCEPTION
   WHEN OTHERS
   THEN
      p.l ('Error code: ', SQLCODE);
      RAISE;
```

In this fragment, when any error occurs, I display the error code and then reraise that same error to propagate the exception out to the enclosing block.

This is a very useful feature of PL/SQL and it seemed only reasonable to implement this action in PLVexc when I encountered this functionality about six months ago (yet another aspect of PL/SQL of which I was ignorant when writing *Oracle PL/SQL Programming*). Eager to please, I spent an hour or two adding the necessary constant, high−level handler and accompanying code. A simplified version of my high−level handler looked like this:

```
PROCEDURE reraise
IS
BEGIN
   p.l ('description of error');
   RAISE;
END;
```

Confident of my new approach, I even put together the following test script to test out my new functionality:

```
BEGIN
   p.l ('Divide by zero!', 1/0);
EXCEPTION
   WHEN OTHERS
   THEN
      PLVrec.reraise;
END;
```

Finally, after all of my furious coding, it was time to compile my PLVexc package and run some tests. Imagine my surprise when I received the following compile error on the **reraise** procedure:

```
PLS−00367: a RAISE statement with no exception name must be inside an
           exception
```

It turned out that my great idea was a completely *invalid* idea! You can only issue the unqualified RAISE statement inside an exception section, which makes a whole lot of sense. If you are not inside an exception section, there is no current exception, so the statement:

```
RAISE;
```

makes no sense at all. What are you raising? A roof? A stink? I sheepishly deleted all **reraise**−related code from my package. Some PL/SQL guru!

The reason I relate this story to you is that it taught me (well, reminded me of) an important lesson: before you embark on building powerful, generic utilities, do the research necessary to prove that your ideas are possible *and* practical.

22.1.6 Handling Errors

Now that you are familiar with the different kind of actions you can request in PLVexc, let's look at the programs you can call to handle your errors. PLVexc provides two levels of handlers. The low−level program, the **handle** procedure, allows (and expects) you to fully specify all the information about the exception in the argument list. The high−level programs rely on a higher level of abstraction, making it easier to use the PLVexc package; PL/Vision automatically figures out the where, what, and when of the problem.

22.1.6.1 The handle procedure

The header for the low−level, generic exception handler program is as follows:

```
PROCEDURE handle
   (context_in IN VARCHAR2,
    err_code_in IN INTEGER,
    handle_action_in IN VARCHAR2,
    string_in IN VARCHAR2 := SQLERRM);
```

The **handle** procedure accepts four arguments, which are explained below:

context_in
: The context or program in which the error occurred. This is free−form text, but as I explain later in Section 22.2, "Application−Specific Exception Packages", you want to use named constants to avoid chaos in this area.

err_code_in
: The error code. You generally pass in the SQLCODE function for this argument, since that shows the current SQL layer error. You can, however, pass any integer, including application−specific errors in

the −20NNN range as well.

handle_action_in
The string constant that informs PLVexc of the action you want to take, such as "record and continue" or "halt." The only values you should pass are the PLVexc constants: **c_go**, **c_recNgo**, **c_stop**, or **c_recNstop**.

string_in
The error message, the default of which is that string returned by the SQLERRM function.

If your error code falls within the range −20,000 to −20,999 and you request that the program be halted, then PLVexc automatically calls RAISE_APPLICATION_ERROR to raise the error and communicate the error information back to the client program. Otherwise, when you want the program halted, **PLVexc.handle** uses the RAISE statement to raise the **process_halted** exception defined in the package.

The various arguments allow developers to handle exceptions in different ways and pass very specific information to the exception−handling component. Let's look at some examples.

1. Write an exception handler to detect NO_DATA_FOUND, in which case information about the current record is saved and the application continues.

```
EXCEPTION
   WHEN NO_DATA_FOUND
   THEN
      PLVexc.handle
         ('getcompany',
          SQLCODE,
          PLVexc.c_recNgo,
          TO_CHAR (v_company_id));
END;
```

   NOTE: Remember that logging or displaying must be turned on in order for a "record" request like that shown above to actually have an impact. See Section 22.1.7, "Recording Errors" for more information on these features.

2. Write an exception to detect a failure in the current program and, instead of relying on SQLCODE, pass a −20NNN error number with an accompanying message. This would be necessary if the error occurs inside a database trigger and must be passed back to the client application.

```
EXCEPTION
   WHEN OTHERS
   THEN
      PLVexc.handle
         ('check_emp_age',
          −20500,
          PLVexc.c_recNstop,
          'Employee too young: ' || TO_CHAR (:new.birthdate));
END;
```

22.1.6.2 The high−level handlers

The **PLVexc.handle** is a significant improvement over writing your own exception−handling code over and over again. Still, it requires that you enter lots of information. You have to provide the current program, the error code, and the action (the error message is optional). Doesn't it seem silly that you would have to tell a PL/SQL program the name of the program that is currently executing? Shouldn't that information be accessible from PL/SQL itself? And shouldn't the PL/SQL runtime engine know about the current error?

The answer to all these questions is "yes and no." Yes, the PL/SQL runtime engine should know about the current program name and the current errors −− and in many circumstances it does know about this information. Unfortunately (and here's the "no" part), while the DBMS_UTILITY.FORMAT_CALL_STACK does return the active PL/SQL execution stack, it does not tell you which program *inside* a package is being executed (see Chapter 21, *PLVlog and PLVtrc: Logging and Tracing*, for more information on this phenomenon). And it is quite impossible for PL/SQL to know the current error when it is an application−specific problem (you raised a programmer−defined exception).

It is possible, on the other hand, to overcome these complications. Using the PL/Vision message, trace, and exception−handling packages in an integrated fashion, you can greatly simplify the task of providing comprehensive exception handling in your applications.

The high−level handler programs of PLVexc hide almost all the details and data needed to respond to and record exceptions. The headers for these handlers are:

```
PROCEDURE recNgo (msg_in IN VARCHAR2 := NULL);
PROCEDURE recNgo (err_code_in IN INTEGER);

PROCEDURE go (msg_in IN VARCHAR2 := NULL);
PROCEDURE go (err_code_in IN INTEGER);

PROCEDURE recNstop (msg_in IN VARCHAR2 := NULL);
PROCEDURE recNstop (err_code_in IN INTEGER);

PROCEDURE stop (msg_in IN VARCHAR2 := NULL);
PROCEDURE stop (err_code_in IN INTEGER);
```

Notice that there is a handler name for each action and there are two versions for each action: one that accepts an additional message and one that has an integer argument. If you call a handler with a string, that string is recorded or displayed as the error message. If you pass a string to a high−level handler like **recNstop**, it will use the value returned by the SQLCODE function as the error number, and your string as the error message.

If you call a handler with an integer error code, the error message is retrieved from PLVmsg facility based on that numeric code (see Section 22.1.6.4, "Defining error messages with PLVmsg"). And those are the only two types of information to pass to PLVexc.

What about the action code? With the high−level handlers, the action you want taken has been moved from the parameter list of the program to the very name of the program itself.

What about the current program name? You don't provide it in the call. Instead, you define the current program with a call to **PLVtrc.startup** at the beginning of each program unit (see Section 22.1.6.5, "Integrating PLVexc with PLVtrc "). Can exception handling get any easier than that?

22.1.6.3 Using the high−level handlers

Let's look at some examples of using these handlers.

1. When my implicit query retrieves too many rows, I want to simply continue processing. For any other errors, record and halt.

```
EXCEPTION
   WHEN TOO_MANY_ROWS
   THEN
      PLVexc.go;

   WHEN OTHERS
```

```
         THEN
            PLVexc.recNstop;
```

2. When my database trigger on the **emp** table detects that the employee is underage, it raises a packaged exception in the −20NNN range. This error number is given a name in the **empmaint** package, which is then passed to the call to **PLVexc.recNhalt** so that this application−specific exception can be registered.

```
BEGIN
   IF :new.birthdate > ADD_MONTHS (SYSDATE, −216 /* 12 x 18 */)
   THEN
      RAISE empmaint.too_young;
   END IF;

EXCEPTION
   WHEN too_young

   THEN
      PLVexc.recNhalt (empmaint.en_too_young);
END;
```

3. When a duplicate value in an index exception is raised, store the current primary key and other unique information for the new company so that you can figure out what went wrong. Then continue with the processing.

```
EXCEPTION
   WHEN DUP_VAL_ON_INDEX
   THEN
      PLVexc.recNgo (TO_CHAR (v_comp_id) || '−' || v_comp_nm));
END;
```

22.1.6.4 Defining error messages with PLVmsg

There are two scenarios under which PLVexc obtains the text of an error message from the PLVmsg package:

1. When you use an integer version of a high−level handler (you pass in an error number).

2. When you use the string version of a high−level handler, but pass in a NULL string (the default value).

In both of these cases, the handler calls **PLVmsg.text**, passing it either the value returned by SQLCODE or your own error number, in order to obtain the error text. If the error number is between −20,000 and −20,999, the PLVmsg package tries to get the error message from the text table maintained by the PLVmsg package. Let's look at how you would put all these pieces in place for the trigger discussed in the previous section. I passed the error number **empmaint.en_too_young** in my call to **recNhalt**. Suppose this constant is defined as:

```
en_too_young CONSTANT INTEGER := −20033;
```

in the **empmaint** package. Then in the initialization section of that package I should also execute the code necessary to store the message for this error in the PLVmsg table. I would do this by issuing a call to **PLVmsg.add_text** procedure as shown below:

```
PACKAGE BODY empmaint
IS
```

```
   en_too_young CONSTANT INTEGER := −20033;

   ... all the code ..

/* The initialization section */
BEGIN
   PLVmsg.add_text
      (en_too_young,

       'Employee must be at least 18 years old.');
END;
```

In fact, for every error number defined in the package, I would need a call to **`PLVmsg.add_text`** to make that string accessible through the PLVmsg interface and, thus, to the PLVexc handler programs.

22.1.6.5 Integrating PLVexc with PLVtrc

As noted in the previous section, when you use the high−level handler programs such as **`go`** and **`recNstop`**, you do not have to tell PLVexc the name of the current program. This is true, however, only if you integrate your use of PLVexc with the PLVtrc package.

PLVtrc provides two programs to build and maintain its own execution stack of PL/SQL programs. You call **`PLVtrc.startup`** to indicate to PL/Vision that a new program has started (and this program can be a procedure, function, or even anonymous block). You call **`PLVtrc.terminate`** to indicate to PL/Vision that the current program has ended. The body of the **`calc_totals`** procedure below demonstrates this approach:

```
PROCEDURE calc_totals
IS
BEGIN
   PLVtrc.startup ('ct');

   ALL_OTHER_CODE;

   PLVtrc.terminate;

EXCEPTION
   WHEN NO_DATA_FOUND
   THEN
      PLVexc.continue;

   WHEN balance_too_low
   THEN
      PLVexc.halt;

   WHEN OTHERS
   THEN
      PLVtrc.terminate;
      RAISE;
END;
```

The first line in the body calls **`startup`**. Then all the rest of the code is executed. The last line in the procedure calls the trace **`terminate`** program. These statements manage the execution stack for successful execution of **`calc_totals`**. Now let's look at the exception handlers. In the first two handlers (for NO_DATA_FOUND and **`balance_too_low`**), I call one of my high−level PLVexc handler programs. These programs automatically maintain the PLVtrc execution stack with a call to **`PLVexc.terminate`**, so you do not have to do it yourself. The last, OTHERS handler only reraises the exception. Since it does not use a PLVexc handler, I must include an explicit call to **`PLVexc.terminate`** to update the execution stack.

In the next example, I use the **`startup`** and **`terminate`** procedures to track execution of a nested, anonymous block.

```
PROCEDURE annual_calcs (val in number)
IS
BEGIN
   PLVtrc.startup ('proc');
   calc_gross_revenue;
   BEGIN
      PLVtrc.startup ('analyze');
      calc_rev_distribution;
      PLVtrc.terminate;
   EXCEPTION
      WHEN OTHERS THEN PLVexc.halt;
   END;
   call_profits;
   PLVtrc.terminate;
EXCEPTION
   WHEN OTHERS THEN PLVexc.rec_continue;
END;
/
```

With this code you can see some of the additional power and flexibility available with PL/Vision. There is no way at all to track through the PL/SQL stack the startup and execution of a local, anonymous block. Sure, you have to code it yourself, but at least you can get the information you need. Furthermore, if you don't turn on the trace, the overhead incurred by the PLVexc program calls is minimal.

Recognizing the difficulty of even remembering to include calls to PLVtrc modules in your programs, the PL/Vision code generator package, PLVgen, generates procedures and functions with calls to **`startup`** and **`terminate`** already in place. So if you start to use PLVgen as a starting point for your program creation, you will be able to leverage all of these components of PL/Vision and actually be *more* productive.

22.1.7 Recording Errors

As you can see from the exception−handling actions, you can record an error in PLVexc. This package gives you two options for how to record the process:

1. Write the error information to the PL/Vision log through calls to PLVlog.

2. Display the error information to the screen (or standard output) using the **`p.l`** procedure.

You can perform both of these steps simultaneously when an error occurs, or you can turn on only one of these options. The programs to manage these record features are discussed below.

22.1.7.1 Logging errors

PLVexc provides a standard PL/Vision toggle to control logging of errors with PLVlog. The headers for these programs are:

```
PROCEDURE log;
PROCEDURE nolog;
FUNCTION logging RETURN BOOLEAN;
```

These toggles allow developers to change the behavior of exception handling in PLVexc without making any changes to their application or to the PLVexc package itself. When you want to record the errors to the PL/Vision log, you simply execute this command before running the application:

```
PLVexc.log;
```

Then each time a PLVexc handler program is executed, the following information is written to the log:

Current program

- Error code
- Error message
- User who raised the exception
- Time/date of logging

The default value for logging in PLVexc is that it is turned on. You do not, in other words, have to call **`PLVexc.log`** to turn on logging if you have just started up your session.

22.1.7.2 Showing errors

PLVexc provides a standard PL/Vision toggle to control showing of errors with PLVshow. The headers for these programs are:

```
PROCEDURE show;
PROCEDURE noshow;
FUNCTION showing RETURN BOOLEAN;
```

These toggles allow developers to change the behavior of exception handling in PLVexc without making any changes to their application or to the PLVexc package itself. When you want to view the exceptions as they occur (or, at least, when the PL/SQL program completes its execution), you simply enter the following command:

```
PLVexc.show;
```

When you are done viewing the errors and only want the information logged (or completely ignored, depending on the value returned by **`PLVexc.logging`**), you execute this command:

```
PLVexc.noshow;
```

The default value for showing exceptions from PLVexc is that it is turned off. You must, in other words, call **`PLVexc.show`** when you want to view exceptions directly from the screen.

Logging and showing of errors are completely independent actions. You do not have to have logging turned on in order to also show the errors.

22.1.7.3 Using the record toggles

The following scenarios will give you a better idea of when and how these toggles would be used.

Scenario 1: Suppose that I want to execute a batch procedure that transfers and transforms several million rows from a temporary table to its final resting place. I know in advance that the load process will generate thousands of exceptions. I know also that it is not necessary to keep track of these errors, so I do not want to clog up my log table with that information. In the following script, therefore, I turn off both logging and display of errors and then execute the batch load.

```
BEGIN
   PLVexc.nolog;
```

```
      PLVexc.noshow;
      batch_load (SYSDATE);
   END;
   /
```

Scenario 2: I am testing a new program and expect errors to pop up. Rather than go through the trouble of querying the contents of the log table, I would like to simply display errors to the screen and respond immediately.

```
SQL> exec PLVexc.nolog;
SQL> exec PLVexc.show;
SQL> exec new_program;
proc1 Code −6502
ORA−06502: PL/SQL: numeric or value error
```

Of course, when the application moves to production status, you will want to log errors to a table or some kind of repository. You rarely ever want to display them directly to users. Consequently, these are the default settings for the PLVexc toggles.

22.1.8 Rolling Back When an Exception Occurs

PLVexc provides a PL/Vision toggle to control whether PLVexc requests a rollback (executed within PLVlog) before error information is written to the log. The headers for these programs are:

```
PROCEDURE rblast;
PROCEDURE rbdef;
PROCEDURE norb;
FUNCTION rb RETURN VARCHAR2;
```

The default is to perform a rollback to the last savepoint set with a call to **`PLVrb.set_savepoint`** (**`rblast`**). If you do not want a rollback to occur before logging the error, issue this command before you start your application:

```
PLVexc.norb;
```

You have only three options concerning rollbacks from within PLVexc:

1. Roll back to the last savepoint set by calling **`PLVrb.set_savepoint`**. Call **`PLVexc.rblast`** to select this behavior (the default).

2. Perform a rollback based on the current/default behavior defined in PLVlog (see Chapter 21). Call **`PLVexc.rbdef`** to select this behavior.

3. Do not perform any rollback before insertion of error information into the PL/Vision log. Call **`PLVexc.norb`** to select this behavior (the default).

If you want to see if PLVexc is currently requesting a rollback, call the **`rb`** function. It returns the PLVlog rollback action code.

22.1.9 Halting in PLVexc

When you request that the processing in your application halt by calling **`recNstop`** or simply **`stop`**, the PLVexc takes all appropriate actions and then issues the following statement:

```
RAISE process_halted;
```

The **`process_halted`** exception is declared by name in the specification of the PLVexc package. It is not associated with any error number in the −20NNN range. As a result, there are only two ways you can trap this exception once it is raised.

1. With an exception handler specifically for this exception, which would look like this:

   ```
   WHEN PLVexc.process_halted
   THEN
      do_something;
   ```

2. With a WHEN OTHERS section, which traps any and every kind of exception.

You can choose to handle the halting exception, in which case the processing of your application might still be able to continue. Generally, however, you will not trap this exception and instead let it propagate up to the top of the PL/SQL calling stack, where it goes unhandled. A WHEN OTHERS handler will, of course, trap this exception. As a final option, you might handle this exception at the outermost block so that you can perform a commit and save writes to the error log (if it is a database table).

Using the bailout feature, you can also truly and completely bail out of your program regardless of the presence of a WHEN OTHERS section. See Section 22.1.10, "Bailing Out with PLVexc" for more information on this feature.

22.1.10 Bailing Out with PLVexc

You have already seen how you can stop your current program by requesting a halting action. This request causes PLVexc to raise an exception as follows:

```
RAISE process_halted;
```

This exception propagates out of, and closes, enclosing PL/SQL blocks until it hits an exception handler specifically for that exception or until it encounters a WHEN OTHERS exception.

But what if you really want to bail out entirely and immediately from your application? I encountered a situation recently where a very long−running program would work fine for an hour or two, but then raise an ORA−3113 error: "end of file on communication channel". My program had lost its connection; there was no point in going on. Yet because I had made rigorous use of my PLVexc exception handler programs, I trapped the error, logged the problem, and continued on to the next transaction. It made no sense, given the error, but on it went, accumulating error log records until the tablespace was full, at which point the application generated ORA−3113 *and* ORA−1547 errors. I had a full−scale mess on my hands.

This experience brought to light a different class of errors: fatal problems whose occurrence should always signal the need for a total shutdown of the application. There is, after all, little point in continuing when you are not connected to the database.

PLVexc supports this functionality by allowing you to specify a set of error codes which, when raised, cause an unstoppable "bail out" from your program. These are called the *bailout errors*. As long as you use PLVexc handlers in all of your exception sections, a bailout error will propagate out of WHEN OTHERS sections even if you specify a continue action.

There are two aspects to the bailout feature: (a) establishing the list of bailout errors; and (b) starting and stopping the bailout itself. It is up to the users of PLVexc to create a list of bailout errors. It is usually left to PLVexc to initiate the bailout; you can, however, do this yourself as well. The programs that support both parts of bailing out are covered below.

22.1.10.1 Managing the bailout error list

You add an error number to the bailout error list by calling the **bailout_on** procedure, whose header is shown below:

```
PROCEDURE bailout_on (err_code_in IN INTEGER);
```

The following statements add several error codes to the bailout error list. In a production environment, you might even place these calls inside a **login.sql** script which is run whenever SQL*Plus is initiated to run a regular, batch process.

```
PLVexc.bailout_on (−3113); /* end−of−file on comm error */
PLVexc.bailout_on (−1547); /* failed to allocate extent */
PLVexc.bailout_on (−1555); /* snapshot too old */
PLVexc.bailout_on (−1562); /* can't extend rollback */
```

To remove an error number from the bailout error list, call the **nobailout_on** procedure:

```
PROCEDURE nobailout_on (err_code_in IN INTEGER);
```

To clear the entire list of bailout errors, call the **clear_bailouts** procedure:

```
PROCEDURE clear_bailouts;
```

To determine if an error number is currently on the bailout list, call the **bailout_error** function, whose header is shown below:

```
FUNCTION bailout_error (err_code_in IN INTEGER) RETURN BOOLEAN;
```

Once you have defined your bailout list, you (and, more to the point, PLVexc) can reference this list to determine if a bailout should be initiated or stopped.

22.1.10.2 Starting and stopping the bailout

The **bailout** procedure raises the **process_halted** exception of the PLVexc package and switches PLVexc into "bailout mode." Here is the header for this procedure:

```
PROCEDURE bailout;
```

This program takes two steps: it sets a Boolean flag to indicate PLVexc is now bailing out and it then raises the **process_halted** exception.

You can stop a bailout in progress with a call to **nobailout**:

```
PROCEDURE nobailout;
```

PLVexc never calls this program. It is up to you to decide if you want to stop propagation of the exception and continue processing in your application.

You can also find out if PLVexc is currently bailing out with a call to the **bailing_out** function:

```
FUNCTION bailing_out RETURN BOOLEAN;
```

The **bailout** procedure is called by the low−level handle procedure if it encounters a bailout error or PLVexc is in bailout mode already. The following IF statement is, in fact, the first line of code in the body of the **handle** procedure:

```
IF bailing_out
THEN
   bailout;
END IF;
```

So if a bailout error was previously encountered or a developer calls **`PLVexc.bailout`** directly, the handle program short−circuits. It does not record the error. (PLVexc assumes that the error was recorded as specified when it was first handled.) It does not display the error; it just raises the **`process_halted`** exception −− again and again for as long as the PLVexc handler programs are called in exception sections of enclosing blocks.

PREVIOUS | HOME | NEXT
21.2 PLVtrc: Tracing Execution of PL/SQL Programs | BOOK INDEX | 22.2 Application−Specific Exception Packages

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

22.2 Application−Specific Exception Packages

PLVexc is a very generic, low−level package. It can be used in many different applications and under many different circumstances. This flexibility is one of its key strengths. I have found, on the other hand, that for complex application development, I will usually not want to make calls *directly* to the **`PLVexc.handle`** procedure. Instead, I will build *another* exception−handling package for my application which, in turn, references PLVexc elements.

This application−specific package offers several advantages, including:

- It is aware of the different kinds of data I need to report on when an error occurs. This awareness helps guarantee a consistent format for error logging; this consistency greatly aids in recovery and analysis efforts.
- It can predefine additional, application−specific exceptions that have no place in PLVexc.

Does this extra layer seem like overkill? Let's explore the motivation for this additional layer of code.

22.2.1 In the Trenches with PLVexc

It is the fall of 1995. I have just built my first version of PLVexc and immediately put it to use in an application that manipulates UPC codes for a retail operation. A UPC code is made up of a 14−digit number and a description. Each UPC (product) has many attributes, such as brand, product type, price, and so on. Each attribute has a name and value.

I find that as I build my exception handlers with **`PLVexc.handle`** I need to pass the same information repeatedly to my exception log. If I am working with UPCs I want to record the UPC number and description with which I am having a problem. When manipulating attributes, I need to keep track of the problematic UPC code, attribute name, and value. I also fully intend to write recovery scripts based on my error log data. For example, if the batch−driven insert of a new UPC fails because I ran out of extents on that table, I would like to be able to reorganize the table and then execute INSERTs from the error log.

If I am going to generate INSERTs from error log text, I need to make sure that text always has the same format.

22.2.1.1 Recording consistent error data

There are two ways I can achieve the desired consistency:

1. Decide on a format and then send a memo to all developers working on this application. The memo explains the format (for example, "concatenate UPC code to description with a single hyphen as delimiter" and "concatenate UPC, attribute name, and value together using a single hyphen as the delimiter between these three elements") and asks developers to follow this format when using the

`PLVexc.handle` procedure. It is then up to each person to cooperate and write the code correctly.

2. Provide a prebuilt procedure that encapsulates the standard format inside the program. With this second approach, a developer conforms to the standard simply by using this program.

If I adopt the first approach (which, I can assure you, is the more−traveled route) here are the kind of exception handlers I find myself writing:

```
WHEN OTHERS
THEN
   PLVexc.handle
      ('ins_upc', SQLCODE,
       PLVexc.c_recNstop,
       TO_CHAR (upc_in) || '-' || desc_in);
END;
```

and:

```
WHEN OTHERS
THEN
   PLVexc.handle
      ('chg_upcattr', SQLCODE,
       PLVexc.c_recNstop,
       TO_CHAR (upc_in) || '-' ||
          attr_in || '-' || new_val_in);
END;
```

and once, because I believe suddenly that the new format is so much clearer, I code my handler this way:

```
WHEN OTHERS
THEN
   PLVexc.handle
      ('ins_upc', SQLCODE,
       PLVexc.c_recNstop,
       'UPC:' || TO_CHAR (upc_in) || ' DESC:' || desc_in);
END;
```

22.2.1.2 Tired fingers, buggy code

This is a lot of tedious, error−prone typing. I find myself expending more brain cells remembering the format than in surmounting application obstacles. And I say to myself: "Gee, it sure would be much easier to just pass the UPC and description to the handler and let *it* format the data properly." That seems like such a good idea that I immediately try out the concept by recoding some exception handlers as follows:

```
WHEN OTHERS
THEN
   upcexc.handle
      (upcexc.c_upc_update, SQLCODE, PLVexc.c_recNstop,
       upc_in, desc_in);
END;
```

and:

```
WHEN OTHERS
THEN
   upcexc.handle
      (upcexc.c_attr_analyze,
       SQLCODE,
       PLVexc.c_recNstop,
       v_curr_upc, new_attr_in, new_val_in);
```

```
END;
```

where **upcexc** is the projected name of a new package that would know about PLVexc and the UPC application. It bridges the gap between the completely generic and the uniquely specific, using module overloading to automatically understand the types of data passed to the **handle** program. Example 22.1 shows the full specification of the **upcexc** package.

Example 22.1: The upcexc Specification

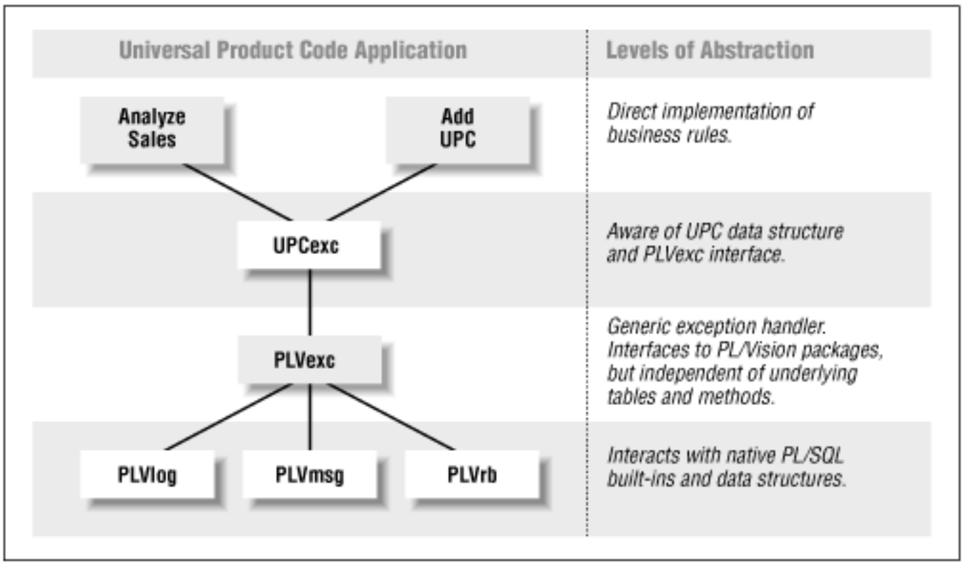
```
CREATE OR REPLACE PACKAGE upcexc
IS
   /* Predefined exceptions, error numbers and messages. */
   upc_update_failure EXCEPTION;
   c_errno_update_upc CONSTANT INTEGER := −20000;
   c_errmsg_update_upc CONSTANT VARCHAR2(100) :=
      'Unable to update upc with new value.';

   no_reg_center EXCEPTION;
   c_errno_no_reg_center CONSTANT INTEGER := −20003;
   c_errmsg_no_reg_center CONSTANT VARCHAR2(100) :=
      'Regional center has not been defined.';

   /* Contexts for exception handling. */
   c_upc_inserts CONSTANT VARCHAR2(3) := 'UI';
   c_upc_updates CONSTANT VARCHAR2(3) := 'UU';
   c_attr_analyze CONSTANT VARCHAR2(3) := 'AA';

   PROCEDURE handle
      (context_in IN VARCHAR2,
       err_code_in IN INTEGER,
       handle_action_in IN VARCHAR2,
       upc_in IN upc.upc%TYPE,
       desc_in IN upc.description%TYPE);

   PROCEDURE handle
      (context_in IN VARCHAR2,
       err_code_in IN INTEGER,
       handle_action_in IN VARCHAR2,
       upc_in IN attribute.upc%TYPE,
       attribute_in IN attribute.attribute%TYPE,
       value_in IN attribute.value%TYPE);
END upcexc;
```

The body of the handler for UPC errors is shown below. Notice that it does just a *little* bit more work than simply calling **PLVexc.handle**; it formats the string passed to **PLVexc.handle** from the individual application−specific arguments.

```
PROCEDURE handle
   (context_in IN VARCHAR2,
    err_code_in IN INTEGER,
    handle_action_in IN VARCHAR2,
    upc_in IN upc.upc%TYPE,
    desc_in IN upc.description%TYPE)
IS
BEGIN
   PLVexc.handle
      (context_in, err_code_in, handle_action_in,
       'UPC=' || upc_in || ' DESC=' || desc_in);
END;
```

Now every time I use the **upcexc.handle** procedure, I am sure that my UPC and description values are formatted properly. At the same time, I use predefined constants from **upcexc** to specify my context or current program. I don't have to make up the name/abbreviation for the program on the fly −− it is already defined for me in the package. This additional consistency makes it easier to analyze and trace errors. Figure 22.1 shows the benefit of this additional layer of coding.

Figure 22.1: Layers of exception handling packages


Universal Product Code Application
Levels of Abstraction
Analyze Sales
Add UPC
Direct implementation of business rules.
UPCexc
Aware of UPC data structure and PLVexc interface.
PLVexc
Generic exception handler. Interfaces to PL/Vision packages, but independent of underlying tables and methods.
PLVlog
PLVmsg
PLVrb
Interacts with native PL/SQL built-ins and data structures.

The final added value of a package like **`upcexc`** is provided by the predefined application errors that fall between −20,000 and −20999. The package contains an exception, error number, and error message for each error in this range. By using these predefined elements, individual developers will not step on each others' error numbers and text.

With **`upcexc`**, I can quickly code my handlers without tripping over syntax and concatenation bars. I am, as a result, much more likely to reuse and fully leverage the underlying PLVexc package.

Special Notes on PLVexc

Remember that the high−level handlers automatically call **`PLVtrc.terminate`**. As a result, you should only use the high−level handlers when you really are leaving a PL/SQL block in which **`PLVtrc.startup`** was called. Otherwise the PLVtrc execution call stack becomes inaccurate.

PREVIOUS | HOME | NEXT
22.1 The Challenge of Exception Handling | BOOK INDEX | 22.3 Implementing PLVexc

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

22.3 Implementing PLVexc

Now that you have seen how to use the PLVexc plug−and−play component, let's go through the steps I followed to build this package. I will even show you two different stages of development of the package. It is very important for you to understand that while it is important to do it right the first time, it is also very unlikely that you will really and truly get it all correct in that first implementation. I present in this book software that I have massaged and fine−tuned repeatedly over the course of a year. You don't see all the pain, indecision, and transitional bugs of PL/Vision, but −− believe me −− they were there.

I learned in the course of building PLVexc that I need to progress through several phases in order to properly design and implement a package component. These phases are:

Phase 1. Understand the challenge and obstacle.

The first step I took to construct PLVexc was to understand fully the challenges faced by developers when building exception handlers in their PL/SQL programs. I have found that the more thoroughly I grasp the problem, the more accurate I will be in providing a solution.

Phase 2. Research the technical issues.

What are all the nuances of exception handling in PL/SQL? What are the gotchas, the no−can−dos? What things are possible, but never before attempted?

Phase 3. Implement the first version of the package.

Admit to yourself right up front: this is Version 1.0. It will change radically before you are done. But put something together so you can move on to the next phase.

Phase 4. Try out the package.

The only way you can really tell if your approach is sound is to try it out. You will encounter requirements, interface issues, and restrictions you would never have discovered from an abstract or theoretical analysis.

Phase 5. Experience phase 3 and phase 4 again and again, each time improving your code and your awareness of best practices.

Be prepared to go back and recode your package. You might even throw away the entire implementation and start again. It is far too easy to believe that just because you spent a lot of time writing and debugging your code and it sure looks pretty that it must be *good*. I try to never think about the hours of coding I have "thrown away." Too painful. Just accept it as part of the process. Be committed to constantly improving and you will eventually end up with something that you like −− and maybe a few others will find useful as well.

Let's look at how I applied these phases as I constructed the PLVexc package.

22.3.1 Analyzing the Need

A developer must ask and answer many questions in order to properly bullet−proof an application from the perspective of handling errors. These questions include the following:

-

Which exceptions should I trap and handle? How do I handle the different kinds of exceptions that occur in my program?

- When should I include a WHEN OTHERS handler? Do I have one at every level or just at the very top or outer block of my program?
- When do I need to use the EXCEPTION_INIT pragma to associate error numbers with messages? How can I avoid using the same error number (in the −20NNN range) in different programs across my application? How can I guarantee a consistent set of messages for these errors?
- When should I call RAISE_APPLICATION_ERROR rather than simply the RAISE statement to raise an exception?
- When I handle an exception, what kind of action should I take? Do I ignore the error and continue, record that the error occurred and then continue, notify the error of the problem, stop execution of the program, reraise the same exception? Will the answer to this question change over time or from program to program?
- If I record my errors, do I write the information to a database table or some memory−based structure, such as a PL/SQL table? What information should I include in the error text? Is all the data I want available from within PL/SQL (consider, for example, how you might determine the name of the program in which an error occurred)? What information will I need in order to do restores and fixes of the data?
- If I write error log records to a database table, how do I make sure that the log is saved, but that the transaction I was working on in my application is rolled back (as necessitated by the error)? Remember that if an exception is passed unhandled out of a PL/SQL block, then all changes to PL/SQL data structures *and* database tables are rolled back implicitly and automatically.
- Will I use the same log table for all of my applications, or should I provide a different log table for each application? Will I then have to create a different program to handle each log table? What should be the structure of the log table?
- How do I write my exception handlers so that when I am in test and debug mode the information is available in one form (such as displayed on the screen), but when the code goes into production that information is redirected? And how, most importantly, can I do this without having to change my code right before it goes into production (which, at least theoretically, would require another round of tests)?

As you can see, it is very easy to ask lots of questions about exception handling. Some questions yield easy answers. Others require advance planning in application design. Still others are too basic to be answered by any generic package. Before I explore what a package can do for us in this situation, let's first examine the usual solution to these questions and the general perspective taken on exception handling.

22.3.2 The Usual Solution

The traditional approach taken by a PL/SQL based−development effort goes something like this:

Step 1. Train developers on how to write exception handlers. The training is basic and does not make any effort to teach programmers "best practices." Students and project managers alike are satisfied to have been at least introduced to the technology. And, of course, there isn't time for more advanced training. These people have an application to write!

Step 2. Develop some basic set of guidelines for displaying and recording errors in the application. In the more sophisticated and experienced organizations, the best developer might actually design a single log table for everyone to use.

Step 3. Distribute the guideline as little more than a glorified memo. Plead with or demand that the team members follow the standards as laid out in the document. Given the state of PL/SQL development and code management utilities, there really isn't any way to enforce the standards.

Step 4. Unleash the developers. Already behind in the schedule, there is a mad scramble to write the programs. The guidelines are followed, at best, unevenly. It is difficult to find time to remember what is written in a document and even more challenging to find time to write code as suggested. The main thing now is to simply churn out code without obvious points of failure.

22.3.2.1 The traditional jumble of exception handling

With this traditional solution to exception handling, you end up with exception handlers that look like the following:

```
WHEN NO_DATA_FOUND
THEN
  DBMS_OUTPUT.PUT_LINE
    ('Calc Totals Failed for ' ||
     v_company_name || ' at ' ||
     v_address1);
  INSERT INTO log_table
    (errcod, errmsg, progname, err_ts)
  VALUES
    (SQLCODE, SQLERRM, 'calc_totals', SYSDATE);
  RAISE;
END;
```

or this:

```
WHEN OTHERS
THEN
   /* Transaction has failed. */
   ROLLBACK TO trans_start;
   IF SQLCODE = −942
   THEN
      INSERT INTO log_table
        (errcod, errmsg, progname, err_ts)
      VALUES
        (SQLCODE, 'Company table not available', 'add_company',
         SYSDATE);
   ELSE
      INSERT INTO log_table
        (errcod, errmsg, progname, err_ts)
      VALUES
        (SQLCODE, TO_CHAR (v_company_id), 'add_company',
         SYSDATE);
   END IF;
```

```
      COMMIT;
   END;
```

22.3.2.2 Drawbacks of the usual handlers

Let's examine these handlers in more detail. In the first example, I display a message to the user, write a record to the log table, and then reraise the same exception (NO_DATA_FOUND). What are the problems with this handler? I can think of three drawbacks:

1. *Reliance on DBMS_OUTPUT.* The hard−coding of the call to DBMS_OUTPUT.PUT_LINE procedure means that this handler sends the message to the screen only if output has been enabled and the host environment recognizes and supports DMBS_OUTPUT. This handler would not, for example, display anything in an Oracle Forms application. The error message, furthermore, will always be displayed when output is enabled. There is no flexibility available when using DBMS_OUTPUT. Finally, the string resulting from the concatenation might be longer than 255 characters, in which case PUT_LINE itself raises the VALUE_ERROR exception.

2. *Exposure of log table implementation.* The structure of the log table is hard−coded into the exception handler. By exposing this level of implementational detail, the programmer makes it very difficult to change or enhance the way information is sent to the log table. This approach also requires every developer to know the structure of the table, from the name of the table and its columns to the type of data passed to each column. Finally, this code assumes that no rollback of a transaction is required and that none will be issued in the future, since such a rollback would also wipe out the insert to the log table.

3. *Experienced developers needed.* On the one hand, you could argue that the code in that first example is simple enough. On the other hand, you could point out that a developer needs to know about the DBMS_OUTPUT builtin package, the SQLCODE and SQLERRM builtin functions, and the fact that the RAISE statement issued without any exception will reraise the same exception.[1]

[1] I will make a confession: at the time that I wrote my first book, *Oracle PL/SQL Programming,* I was not aware of this special form of the RAISE statement.

There was more to that block of code than you might have thought at first! Now let's analyze the second example of the traditional exception handler. In this case, the developer was aware of transaction−related issues. The first step performed is a rollback to the savepoint issued at the start of the transaction. An IF statement then checks for a specific error that occurs when the specified table does not exist. The error information is then inserted in the log table. Immediately after the insert, the developer issues a COMMIT so that the log record is preserved.

While it is admirable that the developer thought of these issues, the handler still hard−codes and exposes the log table implementation. In addition, it is simply not always possible to issue a COMMIT inside an exception handler. That code assumes that any DML changes made prior to when the savepoint for **`trans_start`** was issued should be committed. I hope that is the case. More fundamentally, though, it is not at all clear to me that you want every developer in a project writing code that takes such liberty with the transaction integrity. Hard−coding specific errors like the −942 is also very questionable. Just consider how much time was spent (lost?) in writing that code: find the error number for the problem, write the code, test the different cases.

And of course these two examples would represent just a small percentage of all the code written in any application of substantial complexity to handle the full range of errors. There must be a better way to handle exceptions!

22.3.3 A Package−Based Solution (Version 1)

Let's now examine how a package−based solution can help overcome some of the weaknesses identified in the previous exception sections (the version of PLVexc I will be discussing and presenting may be found in the file **`PLVexc1.spp`** on the companion disk).

Before I dive into developing such a package, it is very important to come up with a set of objectives for the package, as well as an acknowledgment of those issues that the package does not address. My objective for the PLVexc package when I first attempted such a component was to make it easier for developers to handle exceptions in a consistent manner. Most importantly, this meant hiding the implementation of such complexities as writing to a log table, displaying errors to the end user or to the developer (during test mode). In addition, I wanted to provide a mechanism by which individual developers would have to deal with the EXCEPTION_INIT pragma and the −20NNN errors.

Specifically, I envisioned a mode of operation in which a developer could handle the NO_DATA_FOUND exception as shown in the previous example with this kind of code:

```
WHEN NO_DATA_FOUND
THEN
   PLVexc.handle
      ('calc_totals',
       SQLCODE,
       PLVexc.c_recNstop,
       SQLERRM);
END;
```

With PLVexc, the developer calls the generic **`handle`** program to handle the exception. Arguments passed to **`handle`** include the name of the program in which the error occurred, the error code, the type of action to be performed (record and then halt). The fourth argument is the error text, in this case the text returned by SQLERRM.

The **`handle`** program offers the first glimmer of a declarative approach to exception handling. I ask PLVexc to handle my error. I pass it the necessary information, including a description of the type of action I want taken. I leave it to PLVexc to figure out how best to satisfy my request.

22.3.3.1 Implementing the generic handler

I relied heavily on top−down design techniques when implementing the **`handle`** program. Here is the logical flow for **`handle`**:

- If I am recording the exception, write the information to the table.
- If I am displaying the exception, show that information on the screen.
- Finally, if the developer has requested a halting action, raise an exception.

Example 22.2 shows the conversion of these requirements into PL/SQL code. As you can see, the **`handle`** program is very short because it relies heavily on private modules. As a result, the code is very readable. For example, after the call to **`set_context`** (explained below), I can literally *read* my program as follows: "If I am recording the exception, record the exception. If I am displaying errors, display the exception." and so on. No comments are needed. And, best of all, I don't yet have to know how I am going to implement the different programs. I have deferred that level of detail to a later time.

Example 22.2: The Body of the handle Procedure

```
PROCEDURE handle
   (context_in IN VARCHAR2,
    err_code_in IN INTEGER,
    handle_action_in IN VARCHAR2,
    string_in IN VARCHAR2 := SQLERRM)
IS
BEGIN
   set_context (context_in, err_code_in, string_in);

   IF recording_exception (handle_action_in)
   THEN
      record_exception;
   END IF;

   IF showing
   THEN
      display_exception;
   END IF;

   raise_exception (handle_action_in);
END;
```

Well, not all that much later. Let's take a look now at each of the modules called within **`handle`**. We have:

set_context	recording_exception	record_exception
display_exception	raise_exception	

22.3.3.2 Setting the context

The **`set_context`** procedure is a private module; it does not appear in the package specification. This procedure copies the arguments provided in the call to **`handle`** to "current exception" private variables, declared as follows:

```
/* Current exception information. */
curr_err_code PLVexc_log.code%TYPE;
curr_context VARCHAR2(100);
curr_err_info PLVexc_log.text%TYPE;
```

The **`PLVexc_log`** table is the structure used to hold the log of errors written from the PLVexc package.

I then reference these package variables in all of my other private modules that handle calls. By taking this approach, I avoid having to pass these values over and over again as parameters to these programs. You might be tempted to argue that I am writing less structured code since I am switching from parameters to package "globals." That would be true if all of my code and variables were not contained inside a single package. Package−level data such as **`curr_err_code`** does function like global data, but since it is declared in the body of the package I have control over references to and values in that data. Consequently, I do not incur the same risk of side effects as one usually does with nonparameterized references to global data.

Once I have set the context variables, I can move on to the substance of **`handle`**. Next task: record the exception −− if the user has requested this action.

22.3.3.3 Recording the exception

I created a Boolean function to return TRUE if the **`handle`** action is a record action, FALSE otherwise, shown below:

```
FUNCTION recording_exception
```

```
   (handle_action_in IN VARCHAR2)
   RETURN BOOLEAN
IS
BEGIN
   RETURN UPPER (handle_action_in) LIKE 'R%' AND
          logging;
END;
```

There isn't much to this function; it is designed to encapsulate the logic by which I decide if the current exception should be logged. There are two parts to this decision:

1. Has the user passed a record action in the call to **handle**?
2. Has the user turned on logging?

As you can see from the code for **recording_exception**, a record action is one that starts with R. The **c_recNgo** constant is set to RC. The **c_recNstop** constant is set to RH. Notice that this rule is not stated or applied anywhere outside of the **recording_exception** function. It is a level of detail that would be hard (and a waste of brain cells) to remember.

The other aspect to returning TRUE from this function is the call to logging. We now see the toggle coming into use. If the user has called the **nolog** procedure to turn off logging, the **logging** function returns FALSE, which means that **recording_exception** will return FALSE, which means that the **record_exception** program will not execute. Notice that even inside the body of PLVexc I do not make a direct reference to the **log_flag** variable. Instead, I always work through the programmatic interface.

If **recording_exception** does return TRUE, I call the **record_exception** procedure. This program encapsulates or hides the details involved in writing to the log table and is shown below:

```
PROCEDURE record_exception
IS
BEGIN
   INSERT INTO PLVexc_log
      (context, code, text, create_ts)
   VALUES
      (curr_context, curr_err_code,
       curr_err_info, SYSDATE);
EXCEPTION
   WHEN OTHERS THEN NULL;
END;
```

The procedure is nothing more than an INSERT statement based on the current error variables assigned in the **set_context** program.

22.3.3.4 Displaying the exception

Once the error is recorded (or ignored), I display the exception if so requested. In this case, I check the value returned by showing (see "plsql−adv−ex−22−2"; this is the second application of the toggle inside the package). If TRUE, I call the **display_exception** procedure, which deals with all the niceties of constructing a string for display with DBMS_OUTPUT.PUT_LINE:

```
PROCEDURE display_exception IS
BEGIN
   DBMS_OUTPUT.PUT_LINE
      (curr_context || ' Code ' ||
       TO_CHAR (curr_err_code));
   DBMS_OUTPUT.PUT_LINE
      (SUBSTR (curr_err_info, 1, 255));
END;
```

Notice that I perform a SUBSTR on the error information before I pass it to the DBMS_OUTPUT.PUT_LINE procedure. A longer string causes a VALUE_ERROR exception; that is just one of the complications when using the builtin package directly. In version 2 of PLVexc, I rely instead on **`PLVprs.display_wrap`** to automatically wrap the longer strings into a paragraph format.

22.3.3.5 Responding to the exception action

When I have finished with recording and displaying errors, as directed by the user, it is time to respond to the **`handle`** action. I do this with the **`raise_exception`** procedure, the last line in the handle body, shown in full in Example 22.3. In this case, I have moved all the conditional logic ("should I raise an exception and how?") to the body of the private module. First, I check to see if the action (the only argument to **`raise_exception`**) is a "continue action" (**`c_go`** or **`c_recNgo`**). If so, I do nothing by executing NULL −− I do *not* want to raise an exception if the developer has asked that the process simply continue.

On the other hand, if the developer has specified a "halt action" (**`c_stop`** or **`c_recNstop`**), I get to perform an additional service for the user of PLVexc: if the error code is between −20,000 and −20,999 (notice the BETWEEN statement that references the minimum and maximum error code constants), then **`raise_exception`** automatically makes use of the RAISE_APPLICATION_ERROR builtin to raise the exception. Otherwise, the normal RAISE statement is used with the special **`process_halted`** exception. In either case, an exception is propagated out of PLVexc, causing the enclosing block to halt, as was requested.

This feature of PLVexc transfers the burden of having to figure out "which way to raise" from individual developers (some of whom may be novices who are not even aware of RAISE_APPLICATION_ERROR) to the package. If, as I discuss below, you create an application−specific error handling package that contains predefined error numbers and exceptions for codes in the −20NNN range, a developer may not even know that she has raised an error that requires the RAISE_APPLICATION_ERROR builtin. Since PLVexc automatically determines the appropriate raise mechanism, that is one more complexity of which developers can remain blissfully ignorant.

Notice that no single program in PLVexc is more than 20 lines in length. Complex steps and expressions are separated and encapsulated behind named elements (variables, procedures, and functions). The result is a package body that looks simplistic, and one which (I hope) makes my readers say to themselves: "Jeez, I can do *that*!" It is true; you can build packages like this. First, you just need to adopt an impulse to your work that generates the initial ideas (roughly: I can and will improve my environment). Second, you must be fanatically devoted to writing tight, modular code.

Example 22.3: The raise_exception Procedure

```
   PROCEDURE raise_exception
      (handle_action_in IN VARCHAR2)
   IS
   BEGIN
      IF handle_action_in = c_go OR
         handle_action_in = c_recNgo
      THEN
         NULL;
      ELSIF handle_action_in = c_stop OR
            handle_action_in = c_recNstop
      THEN
         IF curr_err_code
            BETWEEN min_err_code AND max_err_code
         THEN
            RAISE_APPLICATION_ERROR
               (curr_err_code, curr_err_info);
         ELSE
            RAISE process_halted;
         END IF;
      END IF;
   END;
```

22.3.3.6 Finding new uses for PLVexc

PLVexc Version 1.0 is so easy to use and so useful that I find myself applying it to other purposes as well. Two realizations ushered in this new phase:

1. PLVexc provides a mechanism for writing a message to a table. This could be used for auditing and logging of *any* information, not simply errors.

2. I don't have to call **`PLVexc.handle`** from within an exception handler. I could call it any time I want to write information out to a table for future analysis.

Suppose that I want to trace the progress of a data transformation and migration job. I find that I can use **`PLVexc.handle`** to write a message out to a table after every 100 transactions:

```
FOR upc_rec IN upc_cur
LOOP
   transform_upc (upc_rec.upc);
   migrate_upc (upc_rec.upc);
   IF MOD (upc_cur%ROWCOUNT, 100) = 0
   THEN
      PLVexc.handle
         ('transmigr', 0,
          PLVexc.c_recNgo,
          upc_rec.upc, upc_rec.description);
   END IF;
END LOOP;
```

This works wonderfully, and can be seen as a validation of the architecture of the PLVexc package. Difficulty arises, however, when I want to examine the contents of the exception log. Since I am now using it for multiple purposes, I have to distinguish between error records and trace records. In addition, I discover that I may want to record my errors without also recording my trace. Since I am using the same package with the same toggles, however, it's all or nothing −− never a good situation for a developer.

I also become uncomfortable with using an exception handler package as a trace mechanism. It's very convenient, but it is also, strictly speaking, outside of the scope of the original package. That fact should set a red flag waving before your eyes. This adaptation of PLVexc is akin to raising an exception to perform a conditional branching in your program. You can make it work, but all it really does is sow confusion and create maintenance/enhancement nightmares.

When you find yourself using programs in ways different from those originally intended, you should perform some analysis and figure out if this use is justified. If not, change your code to a more straightforward implementation.

In the case of PLVexc, such an analysis yields the realization that I am much better off separating my logging code from my exception handling code. Logging of errors is just one special case of all my logging needs. Tracing code execution is another special case. I am able to adapt PLVexc to meet my other needs, but a better long−term solution would be to create a logging package that can be called by PLVexc and by my trace programs.

And so, after my lengthy development cycle for PLVexc, I come to the conclusion that I should rework the internals of the package. Should I be depressed by this development? Not at all. I have learned, over my years of PL/SQL development, that while I should make every effort to do it right the first time, it is also absolutely impossible to get it right the first time. "Right" is a moving target based more on philosophy than on requirements.

As long as I apply my guidelines for best practices at each stage of development, I am certain that my programs will be both immediately useful and easily enhanced to meet future needs. This first version of PLVexc offers a significant improvement over business−as−usual methods for exception handling. That fact does not stop me, however, from crafting yet another version of PLVexc that is much more powerful and easy to use.

22.3.4 Revamping the PLVexc Package

Rather than proceed step−by−step through the implementation of the current version of PLVexc (which is included on the companion disk), I will discuss the ways in which I used other packages of PL/Vision to enhance PLVexc. I will then examine the impact of these changes in specific areas of PLVexc.

I found (as explored in previous sections) that PLVexc actually combined functionality from (what should have been) several different packages. The "final" version of PLVexc makes use of the following packages:

PLVmsg
: Instead of hard−coding the text for messages with a call to SQLERRM or displaying the passed−in string, PLVexc calls the **`PLVmsg.text`** function. This program offers a much more flexible and centralized means of storing and providing message text.

PLVlog
: The PL/Vision logging package is used both to perform rollbacks, if necessary, and to write error information to the log.

PLVtab
: PLVtab provides the table used to maintain the list of bailout errors.

PLVprs
: The **`PLVprs.display_wrap`** program replaces a call to DBMS_OUTPUT.PUT_LINE with a much more interesting and powerful paragraph−wrapping mechanism. You can now view long lines of error text with ease.

PLVtrc
: The trace package is used to maintain an execution stack. This stack can be used to automatically provide the name of the current program, avoiding the need to hard−code this value in the call to the exception handler.

By plugging all of the appropriate elements of PL/Vision into the PLVexc environment, I was able to change that package rapidly into a much more powerful and flexible component. Let's look at how I used PLVlog to greatly enhance the logging capability of PLVexc.

22.3.4.1 Leveraging PLVLog

In the first version of PLVexc, my local **`record_exception`** procedure contained a hard−coded INSERT statement to a specific table. In the PL/Vision version of PLVexc, I replace that INSERT statement −− indeed, that entire local module −− with a call to PLVlog as shown below:

```
IF recording_exception (handle_action_in)
THEN
```

```
   PLVlog.put_line
      (context_in, err_code_in, msg_in, USER,
       rb, TRUE);
END IF;
```

where **`rb`** is the function that returns the current "rollback to" behavior for exception handling and TRUE indicates that this call to **`put_line`** should override the current logging status. This means that even if logging is turned off outside of PLVexc, you will still be able to log your error information.

With this simple substitution, I have accomplished the following:

- Disconnected the exception−handling component from a logging capability. If you want to log information that is not an error, you can do so directly with the PLVlog package. You do not have to tweak PLVexc to coincidentally provide this functionality.

- Greatly increased the logging capability of PLVexc. Since I am now using PLVlog, my exception logging is no longer restricted to simply writing to a database table. I can log to my choice of database table, PL/SQL table, standard output, and string. And as new options appear and are integrated into PLVlog, those options become instantly available to PLVexc as well.

A similar kind of improvement is achieved for showing the error (turned on with a call to **`PLVexc.show`**). Instead of simply calling DBMS_OUTPUT.PUT_LINE in my **`display_exception`** program, I can instead call the **`PLVprs.display_wrap`** so that very long text messages can be displayed in full in wrapped format.

22.3.4.2 Implementing the high−level handlers

The most far−reaching and interesting aspects of the evolution of PLVexc are the high−level handlers. I find it satisfying to be able to type something as simple as:

```
PLVexc.recNgo;
```

and have prebuilt code automatically determine the current program, error number, error message, and everything else I need to record that error and then continue.

You have already seen the headers for these high−level handlers and learned about the difficulties of tracking the current program in PL/SQL. Now let's go inside the PLVexc package to see how I implemented these handlers.

There are a total of eight high−level handlers, with overloadings for string and integer versions for each different action. As you can well imagine, the code that needs to be executed within each of these handlers is very similar. I found, in fact, that the only difference is the action requested and the input (message or error code). Consequently, I built a *private* procedure (it does not appear in the package specification) that is called by all the high−level handlers. I named this program **`terminate_and_handle`** for reasons that will be clear in a moment and called it in my **`recNgo`** handlers as follows:

```
PROCEDURE recNgo (msg_in IN VARCHAR2 := NULL)
IS
BEGIN
   terminate_and_handle (c_recNgo, msg_in);
END;

PROCEDURE recNgo (err_code_in IN INTEGER)
IS
BEGIN
```

```
   terminate_and_handle (c_recNgo, err_code_in);
END;
```

A few things to notice: first, if you do not provide an argument when you call **recNgo**, PLVexc executes the string version, which means that it records the value returned by SQLCODE as the error. Second, I call **terminate_and_handle** in each of these two overloaded procedures −− so **terminate_and_handle** must also be overloaded. Here, in fact, is the body of the code behind the two versions of **terminate_and_handle**:

```
PROCEDURE terminate_and_handle
   (action_in IN VARCHAR2,
    msg_in IN VARCHAR2 := NULL)
IS
BEGIN
   PLVtrc.terminate;
   handle (PLVtrc.prevmod, SQLCODE, action_in,
       NVL (msg_in, PLVmsg.text (SQLCODE)));
END;

PROCEDURE terminate_and_handle
   (action_in IN VARCHAR2,
    err_code_in IN INTEGER)
IS
BEGIN
   PLVtrc.terminate;
   handle
      (PLVtrc.prevmod, err_code_in, action_in,
       PLVmsg.text (err_code_in));
END;
```

You can probably see how I came up with the name for this private procedure. It does only two things: first, it calls **PLVtrc.terminate** to remove the current program from the PLVtrc−managed execution stack. Second, it calls that good old, low−level handler program to handle the error according to the information provided to it from various sources. It obtains the context from the **PLVtrc.prevmod** function (this returns the previous module, since the call to **PLVtrc.terminate** has already popped the execution call stack). This convoluted approach was necessary because I couldn't wait to do the terminate after the call to handle. That program might raise an exception!

If I have called the string version of a high−level handler and, hence, the string version of **terminate_and_handle**, I rely on SQLCODE to retrieve the error, but then pass the input string as the error text (unless it is NULL). If I have called an integer version of the handler, that value is used as the error code. That same number is also passed to **PLVmsg.text** to retrieve the text for that error.

Again, by leveraging all of these other PL/Vision packages, I can actually simplify the code required in my handlers, but end up with a much more robust implementation.

PREVIOUS | HOME | NEXT
22.2 Application−Specific Exception Packages | BOOK INDEX | VI. Testing Your Knowledge

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

Advanced Oracle PL/SQL Programming with Packages

By Steven Feuerstein; ISBN 1−56592−238−7E
First Edition, published 1996−10−01.
(See the catalog page for this book.)

Search the text of *Advanced Oracle PL/SQL Programming with Packages*.

Index

Symbols | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

Table of Contents

Part I: Working With Packages

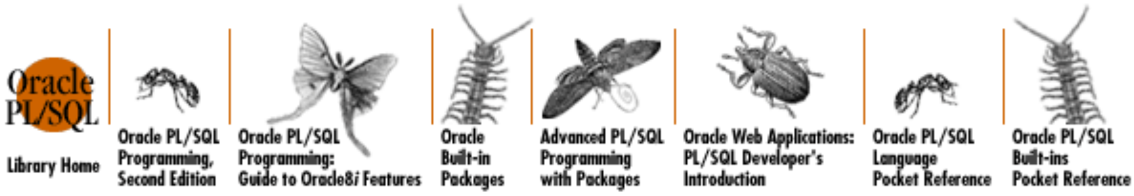
Part II: PL/Vision Overview

Part III: Building Block Packages

Part IV: Developer Utility Packages

Part V: Plug−and−Play Packages

Part VI: Testing Your Knowledge


Oracle PL/SQL
Library Home
Oracle PL/SQL Programming, Second Edition
Oracle PL/SQL Programming: Guide to Oracle8i Features
Oracle Built-in Packages
Advanced PL/SQL Programming with Packages
Oracle Web Applications: PL/SQL Developer's Introduction
Oracle PL/SQL Language Pocket Reference
Oracle PL/SQL Built-ins Pocket Reference

PREVIOUS
Part I
NEXT

Part I: Working With Packages

This part of the book introduces packages, the most important construct in PL/SQL for building reusable code and employing object−oriented design techniques. Chapter 1, *PL/SQL Packages* describes how and when you should create packages and explains why PL/SQL developers should learn to center their entire development process around packages. Chapter 2, *Best Practices for Packages* presents my tried−and−true "best practices" for building packages. Chapter 3, *The PL/SQL Development Spiral* puts packages in the context of overall PL/SQL development by providing a tutorial on solving a typical problem in PL/SQL.

PREVIOUS | HOME | NEXT
Acknowledgments | BOOK INDEX | 1. PL/SQL Packages

Part II: PL/Vision Overview

This part of the book describes PL/Vision, a collection of PL/SQL packages and supporting scripts that can radically change the way you develop applications with the PL/SQL language. Chapter 4, *Getting Started with PL/Vision* introduces the individual PL/Vision packages and provides installation instructions for the software included on the companion disk. Chapter 5, *PL/Vision Package Specifications* is a reference section that summarizes the contents of all of the PL/Vision specifications.

PREVIOUS
3.13 Glancing Backward, Looking Upward
HOME
BOOK INDEX
NEXT
4. Getting Started with PL/Vision

PREVIOUS Part III NEXT

Part III: Building Block Packages

This part of the book (Chapters 6 through 13) describes the building block packages of PL/Vision. These are low−level packages upon which other packages in PL/Vision are built; you can use them to enhance your development in various ways. Examples of building block packages include string parsers, a list manager and an interface to PL/SQL tables.

PREVIOUS
5.29 PLVvu: Code and Error Viewing

HOME
BOOK INDEX

NEXT
6. PLV: Top−Level Constants and Functions

Part IV

Part IV: Developer Utility Packages

This part of the book (Chapters 14 through 18) describes the developer utilities of PL/Vision. These are programs that improve your PL/SQL development environment. Examples include a code generator, a powerful substitute for SHOW ERRORS, and an online help delivery mechanism.

PREVIOUS HOME NEXT
13.10 Tracing PLVfile Activity BOOK INDEX 14. PLVtmr: Analyzing Program Performance

Part V: Plug−and−Play Packages

This part of the book (Chapters 19 through 22) describes the plug−and−play components of PL/Vision. These are pieces of code that can be used as−is in your own applications. Examples include a high−level exception handler mechanism and a generic, reusable logging mechanism.

PREVIOUS | HOME | NEXT
18.2 PLVcat: Cataloguing PL/SQL Source Code | BOOK INDEX | 19. PLVdyn and PLVfk: Dynamic SQL and PL/SQL

Part VI: Testing Your Knowledge

This part of the book contains a set of exercises (and their solutions) to test your knowledge of the PL/SQL language. You might want to try out the exercises before you dive into Parts III, IV, and V of this book.

PREVIOUS	HOME	NEXT
22.3 Implementing PLVexc	BOOK INDEX	A. Appendix: PL/SQL Exercises

Dedication NEXT

Dedication

Many people in the computer software industry are preoccupied with the year 2000. How many systems will break down? How much money will it cost to keep things going? I find myself thinking about the year 2000 as more of a handy "check point" for the state of humanity. After all, if we are going to consider ourselves as somehow more than just mammals, we should hold ourselves to a different standard than simply propagation of the species and individual survival.

So I ask myself: What progress have we made (over the last 100 years, 1000, 2000, whatever!) in fashioning a world grounded in justice and equality? And I would have to say that the picture as we approach the year 2000 is fairly bleak. Most people live in poverty, a handful control most of the resources and power, and−−well, I'm sure you've read the same stories I have. It's easy to get depressed, isn't it? Fortunately, some among us refuse to accept the status quo.

I dedicate this book to the thousands and thousands of people around the world who devote their lives to combating tyranny and exploitation−−those who never receive awards from corporate foundations and often perish "en la lucha" −− especially those who suffer at the hands of my own government.

Forget about the generals, CEOs, and sports celebrities of our day. They just show you how well mammals can adapt to highly varied circumstances and still flourish. Let's instead celebrate and recognize the few among us who represent the best of what we can achieve as moral beings.

−− Steven Feuerstein

HOME
BOOK INDEX

NEXT
Foreword

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

Foreword

Foreword

There is nothing more difficult to take in hand, more perilous to conduct or more uncertain in its success, than to take the lead in the introduction of a new order of things.

− −− Niccolo Machiavelli, *The Prince* , 1513

Paradigm: Shift Happens

− −− Author Unknown

Change is inevitable. If you aren't prepared to accept and embrace it, you are bound to miss many opportunities. I use the above phrases to remind myself to be ready for that great new idea when it comes along. They came in especially handy when I first got to know Steven Feuerstein and his views on PL/SQL.

Back in September 1995, I was the development leader for a team in the Business Applications group at Symantec and we were gearing up for a new project, namely to build a front end to the customer information in Oracle's Financials package using the Forms component of Oracle's Developer/2000 toolset. We had never used Oracle Forms prior to Version 4.5 (probably a good thing) and this would be our largest effort involving PL/SQL−based code to date, so we had become intimately familiar overnight with the concept of "change." During the analysis phase of the project, I had encountered some of Steven's work in several articles, presentations, and his first book, *Oracle PL/SQL Programming*. After reading the book, it was clear that he had something different to say and that we might want to listen. His use of PL/SQL and packages in particular was like nothing I had seen before. I contacted Steven and asked him to come out and deliver his course as a kick−off to the development phase of the project.

After the entire development team had gone through the course and many of us had read a good part of his book, we became strongly motivated to streamline the decision−making process when coding, look for redundant code and put it into reusable modules, and build our code in layers by removing the implementation of a piece of functionality from its usage. Most importantly and more generally, we learned how to develop code in new ways. We knew we had a great opportunity with this project to apply Steven's lessons, producing an application and code base that would be flexible and maintainable.

The first step was to design the layers of code so that commonly used functions could be shared by any Symantec programmer building an application based on PL/SQL. Using packages, we took tasks like sequence generation and global variable management and put them in low−level modules. We put row date−time stamping functions in another layer, then moved on to create an application−specific layer and finally a form−specific layer. These layers were then used to construct highly robust and reusable templates for Oracle Forms development. Each layer used packages in a big way to isolate the functionality needed specifically within that layer. Fortunately, we had an early version of PL/Vision (Steven's library of PL/SQL code) and expanded on it to create many of the functions in the database layers.

Let me give you an example of how Steven's guidelines for improving the development process helped us out. I mentioned earlier that he talked to us about streamlining the decision−making process to reduce decision points and improve productivity. One of his specific recommendations in this vein was: always write explicit cursors when you need to execute a query in PL/SQL. Don't spend any time debating with yourself about whether to use a "quick and dirty" implicit or a programmatic explicit. Upon hearing this dogma, I was skeptical about the payback of the additional code required for the religious use of explicit cursors. This led Steven to reply: "Write the extra code for a week. By the end of the week, you won't even think about it any more."

He was absolutely right. The question of whether or not to use an explicit cursor has disappeared from our thought process. SELECT INTO is no longer used and OPEN−FETCH−CLOSE is now second nature. The result? Our code is more efficient, readable and consistent. Multiply the savings from that decision by 20 or

100 and it is easy to see how we were able to roll out our application in record time.

As the development of our application progressed, many shifts happened and we found it necessary to go back to completed modules and even change fundamentally the way a particular program was implemented. It was at this point that our layering efforts really paid off. We were usually able to make modifications (some of them rather significant) at the lower layers with no impact on the higher ones. We found with a big sigh of relief that it was even possible to add or modify key business rules without everyone having to change their own particular forms and high−level packages.

A few months into the development effort, our success showed that Oracle Forms was a viable tool and so a second, larger project was begun by another team. The task this time was to create an Oracle Developer/2000−based front end to the Order Entry module of Oracle's Financials application. We were determined that this new system would leverage many of the layers we had previously built and, sure enough, everything fell right into place. I have never seen an application move from specification to working code faster than Greyhound (the code name for this second system). Within 4 months, both of our ground−breaking applications were delivered to nearly 600 users in the smoothest roll−out any of us can remember.

In the projects I have just described, the majority of the PL/SQL code was written into the client−side of the equation, such as Oracle Forms PL/SQL libraries and forms. My next set of projects involve building browser−based applications for our corporate intranet: the kind of intranet which has our sales force directly updating core Financials datasets. These applications will now be written almost exclusively in PL/SQL packages residing in the server. Due to this fact, I expect that PL/Vision will become a core component of my toolbox. I have already developed a set of packages to manage data interactions with a Web browser client using PL/Vision and the techniques which have made it such an indispensable tool.

Maybe your users are different, but our internal customers expect to receive a new "shrink−wrapped" release of their applications packed with new features every few weeks (maybe Steven's next book could help us manage customers expectations). If it weren't for the things we have learned from Steven, directly from his code or as a result of his helping us think in new ways, we wouldn't be able to even imagine the maintainability we currently enjoy with our applications.

To help ensure that all our developers become proficient with PL/SQL as quickly as possible, every programmer in the Business Applications group is now given a copy of *Oracle PL/SQL Programming* as part of their new−hire kit. This second book will now be included in that kit and, for that matter, probably any other books Steven chooses to write in the future. I am constantly amazed at the different techniques and new possibilities we are able to learn about PL/SQL through Steven's efforts. I have also come to appreciate his ability to write about PL/SQL code construction in ways that keep me entertained and challenged, rather than nodding off late in the afternoon.

The bottom line is that Stevens "best practices" approach to PL/SQL has changed the way we at Symantec think about coding in PL/SQL. Through him, we have also begun to see the real power of Oracle's packages and how to use them in our everyday work. This "new order of things" has already paid off time and time again, not only for the developers in our organization, but for our customers. If you write PL/SQL code, I have no doubt that the ideas, software, and examples you find in this book will become a permanent part of your PL/SQL landscape.

−− *Brian Shelden*

Manager, Information Systems

Symantec Corporation

PREVIOUS
Dedication

HOME
BOOK INDEX

NEXT
Preface

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8*i* Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

Preface

Contents:

Who would have thought that just one year after the publication of *Oracle PL/SQL Programming,[1]* a 916−page tome on "everything PL/SQL", I'd end up writing a second book about the PL/SQL language? Although back in September 1995 I wasn't arrogant enough to think that I knew all there was to know about PL/SQL, I also underestimated how much more I had still to learn!

I am firmly of the belief that one never stops learning −− as long as one is open to learning. The area of PL/SQL in which I needed lots more education turned out to be packages. In my first book I explained how to build and use packages. I even provided lots of examples of package construction. But I started to realize that this wasn't enough. Over the past year, I have been designing and developing a set of packages to help me build PL/SQL−based applications. This was a thoroughly selfish effort: I wanted to be as productive as possible, and I wanted to overcome a number of weaknesses −− however transient −− in the PL/SQL language. In the process of writing this software, I learned a good deal about the best ways to build PL/SQL code, especially regarding packages. I also discovered some very interesting techniques that can make packaged software more maintainable, accessible, and easy to use.

As my thinking on the construction of packages crystallized, I began to view all of my packages as a library of code that could be used by *any* PL/SQL developer. I also realized that I wanted to share the new techniques and lessons I had uncovered. The result? This book and the PL/Vision product.

How often do you find yourself writing a program and simultaneously thinking: somebody must have done this before! You feel certain that you are reinventing the wheel. Worse, if you are sufficiently honest with yourself, you will also admit that someone else has probably spent more time on the problem and has already come up with a better solution than you are likely to develop for your specific application.

Often, you know you should take the time to "genericize" a program so that you can use it again and again in different circumstances. Somehow, however, you never find the time −− and the mental space −− to take your code to that higher level of abstraction.

So you limp along, accepting a relatively low level of productivity and reusing a truly minimal amount of code. You write the same things over and over and simply push aside the feeling that you are wasting your time.

Oracle developers are fortunate to be able to use an advanced, robust language like PL/SQL. PL/SQL developers are, on the other hand, less than fortunate (at least as of September 1996) to find that the supporting environment for PL/SQL is still very immature. Where are the debuggers, the code formatters and generators, the toolboxes of reusable programs and objects? When will we have a powerful editor that knows about PL/SQL syntax and −− more importantly −− the stored code available for execution?

When, you might also ask, will this guy stop complaining? It is acceptable to identify weaknesses. It is constructive to analyze areas for improvement. At some point, however, you have to stop whining and start improving things for yourself. Best yet, keep on whining but engage in self−improvement at the same time!

This book will help you write better packages. It will also show you how to use the "prebuilt" packages of the PL/Vision software product −− my attempt to change the "situation on the ground" for PL/SQL programmers. Finally, I hope that it will, via examination of my source code and the way I separated functional areas in PL/Vision, offer a blueprint for PL/SQL developers to discover how to take full advantage of PL/SQL packages in their day−to−day programming.

Objectives of This Book

Why did I write this book? I have the following modest objectives:

- *Make sure as many PL/SQL developers as possible know about packages and how to use them.* Students who attend my training sessions soon come to realize after a day or two that the answer to almost any question I ask is: "Build a package." They also often stumble out of the sessions chanting the mantra: "Packages, packages, packages..." I wrote this book and the companion software because I believe that packages are the single most important element of the PL/SQL language. You can never go wrong putting your code inside a package. You will, on the other hand, almost always regret not placing your functions and procedures inside packages from the get−go.

- *Get my software into the hands of as many* PL/SQL *developers as possible.* I like to see others get the benefit of my efforts. I like the idea that my prebuilt software will free up your time. This book contains a full−use version of the PL/Vision product, PL/Vision Lite, along with extensive documentation on how to use this library.

  Of course, I also like to be compensated for my efforts, so you can also purchase a license to PL/Vision Professional; see the RevealNet website at >http://www.revealnet.com">

  (Unless otherwise noted, all references in this book to PL/Vision are to PL/Vision Lite.) You might use just a little bit of PL/Vision; you might leverage every single package into your production applications. Many readers will do nothing more than cannibalize PL/Vision, learning from these packages how to improve their own code. All of these variations are welcome and encouraged!

- *Make my readers more creative and effective problem solvers.* I've had a wonderful time thinking about how to modularize and construct in layers basic packages to improve PL/SQL programming. I learned about effective ways to construct packages and about all kinds of magic you can do when you internalize the features and benefits of packages stored in shared memory. When I encountered an obstacle, I didn't throw up my hands and look for a workaround. Instead, I asked myself: "What kind of package can I build to solve this problem?" That attitude forced me to be creative, and there is nothing more exciting than unleashed creativity. I hope that this book and the software that comes with it inspires every one of my readers to fashion new answers to old questions and newer answers to new questions. You will never regret the conceptual leaps you take in the process.

PREVIOUS | HOME | NEXT
Foreword | BOOK INDEX | Structure of This Book

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

Structure of This Book

This book is separated into six parts as follows:

Part I: Working With Packages

This part introduces packages, the most important construct in PL/SQL for building reusable code and employing object−oriented design techniques. Chapter 1, *PL/SQL Packages* describes how and when you should create packages and explains why PL/SQL developers should learn to center their entire development process around packages Chapter 2, *Best Practices for Packages* presents my tried−and−true "best practices" for building packages. Chapter 3, *The PL/SQL Development Spiral* puts packages in the context of overall PL/SQL development by providing a tutorial on solving a typical problem in PL/SQL.

Part II: PL/Vision Overview

Chapter 4, *Getting Started with PL/Vision* gets you started with PL/Vision, a collection of PL/SQL packages and supporting SQL*Plus scripts that can radically change the way you develop applications with the PL/SQL language. Chapter 5, *PL/Vision Package Specifications* provides a brief summary of all of the specifications for the PL/Vision packages, and is marked with a thumb−tab for quick reference.

Part III: Building Block Packages

Chapters 6 through 13 describe the building block packages of PL/Vision; these are low−level packages that you can use to enhance your development −− for example, string parsers, a list manager, and an interface to PL/SQL tables.

Part IV: Developer Utility Packages

Chapters 14 through 18 describe the developer utilities of PL/Vision; these are programs that improve your PL/SQL development environment −− for example, a code generator, a powerful substitute for SHOW ERORS, and an online help delivery mechanism.

Part V: PL/Vision Plug−and−Play Packages

Chapters 19 through 22 describe the plug−and−play components of PL/Vision; these are pieces of code that can be used as is in your own applications −− for example, a high−level exception handler mechanism and a generic, reusable logging mechanism.

Part VI: Testing Your Knowledge

The appendix provides a set of exercises (and their solutions) to test your knowledge of the PL/SQL language.

Structure of This Book

This book is separated into six parts as follows:

Part I: Working With Packages

This part introduces packages, the most important construct in PL/SQL for building reusable code and employing object−oriented design techniques. Chapter 1, *PL/SQL Packages* describes how and when you should create packages and explains why PL/SQL developers should learn to center their entire development process around packages Chapter 2, *Best Practices for Packages* presents my tried−and−true "best practices" for building packages. Chapter 3, *The PL/SQL Development Spiral* puts packages in the context of overall PL/SQL development by providing a tutorial on solving a typical problem in PL/SQL.

Part II: PL/Vision Overview

Chapter 4, *Getting Started with PL/Vision* gets you started with PL/Vision, a collection of PL/SQL packages and supporting SQL*Plus scripts that can radically change the way you develop applications with the PL/SQL language. Chapter 5, *PL/Vision Package Specifications* provides a brief summary of all of the specifications for the PL/Vision packages, and is marked with a thumb−tab for quick reference.

Part III: Building Block Packages

Chapters 6 through 13 describe the building block packages of PL/Vision; these are low−level packages that you can use to enhance your development −− for example, string parsers, a list manager, and an interface to PL/SQL tables.

Part IV: Developer Utility Packages

Chapters 14 through 18 describe the developer utilities of PL/Vision; these are programs that improve your PL/SQL development environment −− for example, a code generator, a powerful substitute for SHOW ERORS, and an online help delivery mechanism.

Part V: PL/Vision Plug−and−Play Packages

Chapters 19 through 22 describe the plug−and−play components of PL/Vision; these are pieces of code that can be used as is in your own applications −− for example, a high−level exception handler mechanism and a generic, reusable logging mechanism.

Part VI: Testing Your Knowledge

The appendix provides a set of exercises (and their solutions) to test your knowledge of the PL/SQL language.

>

Conventions Used in This Book

The following conventions are used in this book:

Italic
: is used for file and directory names.

Bold
: is used in headers.

`Constant width`
: is used for code examples and for variable, procedure, function, and package names in the text, as well as executable SQL scripts.

UPPERCASE
: in code examples, indicates PL/SQL keywords.

lowercase
: in code examples, indicates user−defined items such as variables and parameters.

punctuation
: in code examples, enter exactly as shown.

indentation
: in code examples, helps to show structure but is not required.

.
: In code examples and related discussions, a dot qualifies a reference by separating an object name from a component name. For example, dot notation is used to specify declarations within a package (e.g., **`PLVvu.code`**).

In general, all of the discussions and examples in this book apply regardless of the machine and/or operating system you are using. In those few cases where a feature is in any way system−dependent, I note that in the text.

PREVIOUS Preface NEXT

About the Disk

The content of the companion PC disk (in MS−DOS format) that accompanies this book has been included on the CD, in the */advprog/disk/* directory. This disk contains a number of different elements (in compressed format), :

- The installation program for RevealNet's PL/Vision Lite Online Reference, a companion to the book.
- The packages of PL/Vision Lite.
- Additional SQL*Plus scripts that improve developer productivity. Some of these scripts provide shortcuts to executing PL/Vision elements. Others are useful independent of PL/Vision.
- Additional programs referenced throughout the book.

The disk itself can be run on any Windows 3.1, Windows for Workgroups, Windows 95, or Windows NT system. If you do not have a Windows system and are running a Macintosh, UNIX, or other type of system, you can obtain the PL/Vision Lite software from the RevealNet site at http://www.revealnet.com.

Chapter 4 explains the detailed contents of the directories created by the PL/Vision Lite installation program **`setup.exe`**.

PREVIOUS | HOME | NEXT
Conventions Used in This Book | BOOK INDEX | About PL/Vision

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

About PL/Vision

PL/Vision is a collection of PL/SQL packages and supporting SQL*Plus scripts which can radically change the way you develop applications with the PL/SQL language. PL/Vision programs can be put to work instantly in your environment, as standalone utilities, low−level functions, and plug−and−play components. You can retrofit existing applications to use PL/Vision or simply apply it to new development. Part 2, *PL/Vision Overview* of this book explains PL/Vision in detail −− what it consists of and how to use it.

PL/Vision is offered in two "flavors" by RevealNet, Inc.: Lite and Professional. PL/Vision Lite is the version of PL/Vision which serves as a companion product to this book. It was developed to help you take advantage of many advanced PL/SQL techniques in your own development environment. While PL/Vision Lite's functionality will not be extended, I will provide fixes to any bugs reported in that version of PL/Vision since the release of this book. You can check out the most current version of PL/Vision Lite on RevealNet's site at http://www.revealnet.com. Chapter 4 describes how to install the PL/Vision Lite software.

PL/Vision Professional is a fully supported and constantly evolving product. It offers a wider range of functionality, comprehensive leveraging of the latest versions of PL/SQL (including PL/SQL Release 2.3, Oracle Web Agent extensions, and PL/SQL Version 3, which will support object technology in the Oracle8 Server), and many code samples to help you take full advantage of PL/Vision. You can register for −− or simply get more information about −− PL/Vision Professional at the http://www.revealnet.com Web site.

I like to think of PL/Vision as more than just a software product; it is an open channel of communication between me and PL/SQL developers around the world. It is the means by which I will share my new discoveries in PL/SQL, especially those discoveries I can transform into packages for your use. It is also the way (through www.revealnet.com) for you to let me know about your suggestions for enhancements to PL/Vision.

PL/Vision is a large and complex body of code. This book and the PL/Vision Online Reference (on the companion disk) should go a long way towards making the software more accessible and useful to you. You can use this book both as user guide and mentor. You can use the Online Reference as a quick syntax checker for PL/Vision and also as a mechanism to view the source code of PL/Vision.

Book as User Guide

One way to use this book is as a user's guide to PL/Vision Lite. Chapter 5 offers a quick reference view of all the different package specifications for the software. From these tables of program headers and package constants, you can quickly glean the syntax you need to follow to execute PL/Vision programs.

Each of the chapters in Parts III, IV, and V presents one or more of the PL/Vision packages. In these chapters I use a consistent format with the following elements:

- Introduction and overview.
- A guide to how to use the elements of the package.

Special notes regarding the package (if any).

- An explanation of how the package was designed and built. This last section is present only for some of the packages and is explained more fully later, in .

To keep the length of the book reasonable, I do not reproduce the full specifications and bodies for the PL/Vision packages in the book, but I do highlight the most interesting aspects of the packages. I encourage you to use the editor of your choice or the PL/Vision Online Reference to examine the code at your leisure or in parallel with the reading of the book.

Book as Mentor

I will be very happy if you decide to use PL/Vision in your development environment. More important than using PL/Vision, however, is learning how and why it is constructed the way it is. I will be happiest of all if, as a result of reading this book and studying my source code, you build your own version of PL/Vision: your own set of highly reusable, plug−and−play component packages.

I have tried to add content to the book itself that will move you along in this direction. In a number of the chapters, I go beyond describing how to use my packages to exploring why and how I build those packages. These implementation stories can go a long way in educating or inspiring you to build your own packages. In particular, see Chapter 8, *PLVtab: Easy Access to PL/SQL Tables* (PLVtab), Chapter 11, *PLVobj: A Packaged Interface to ALL_OBJECTS* (PLVobj), Chapter 15, *PLVvu: Viewing Source Code and Compile Errors* (PLVvu), Chapter 16, *PLVgen: Generating PL/SQL Programs* (PLVgen), Chapter 17, *PLVhlp: Online Help for PL/SQL Programs* (PLVhlp), Chapter 19, *PLVdyn and PLVfk: Dynamic SQL and PL/SQL* (PLVdyn), Chapter 20, *PLVcmt and PLVrb: Commit and Rollback Processing* (PLVcmt and PLVrb), and Chapter 22, *Exception Handling* (PLVexc). You will also find many smaller sections in other chapters offering insights into particular aspects of the packages.

Online Information

Information related to this book and PL/Vision Lite is available on the World Wide Web at:

http://www.revealnet.com/plvision

> ***NOTE:*** Please check the Web site for any code changes to PL/Vision Lite which may have occurred since publication of this book.

PREVIOUS | HOME | NEXT
About the Disk | BOOK INDEX | Comments and Questions

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8*i* Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference

Comments and Questions

Please address comments and questions concerning this book to the publisher:

O'Reilly & Associates
101 Morris Street
Sebastopol, CA 95472
1−800−998−9938 (in the U.S. or Canada)
1−707−829−0515 (international or local)
1−707−829−0104 (FAX)

You can also send us messages electronically. See the insert in the book for information about all of O'Reilly & Associates' online services.

To ask technical questions or to comment on the PL/Vision Lite product, send email to plvision@www.revealnet.com.

Acknowledgments

This book is very much a product of my imagination, grounded in the realities and possibilities of the PL/SQL language. Vague technical fantasies can be translated into packages only with a freedom of time and spirit. Many people helped in many ways to provide me with this freedom.

Bill Hinman, President and CEO of SSC, has supported me fully and encouraged me strenuously from the day I joined his consulting company. I am especially grateful for the weeks of straight writing time I was able to enjoy in the final stages of writing this book. Barrie Hinman, CFO of SSC, has been invaluable in managing the many details of our growing organization and my scattered activities. Hugo Toledo, author of *Oracle Networking*, made many things possible for me over the past year, from helping me with the installation of Oracle Server 7.3 for Windows NT to directing me to the executable which made it possible to print my training materials in "pure black and white." Hugo is my "answer man" and he'll never understand how deeply he is appreciated.

My technical reviewers played a critical role in turning my many hundreds of pages of somewhat raw text into a useful book (I hope). Brian Shelden of Symantec has been an enthusiastic booster and, much more importantly, tester of my concepts and code. Thomas Dunbar of WebCys again provided many excellent criticisms of my text and approach, forcing me to clarify concepts and justify techniques. Bert Scalzo showed an early and steady willingness to point out my mistakes which makes him a very valuable asset in my technical life. John Beresniewicz of Wynnsoft appeared late on the scene, but jumped in with both hands typing. His ideas and experience have enriched both the text and my understanding of my own code. Chuck Sisk of SSC was ready to take on whatever I asked of him even if that changed every week.

Finally, there is David Thompson of SSC. David is a meticulous, principled, and very smart Oracle developer. Both in my first book and in this text, he went beyond critiquing my work to providing well thought−out additions to the text. David built the PLVddd package, initially in a version independent of PL/Vision. He then converted it to use the PL/Vision packages, a process that served as an excellent test of the usefulness and usability of my software.

Many of my reviewers took time during their summer vacations to read my text, and for that I am deeply grateful.

I thank Kasu Sista and Ken Wiegle of Links Technology, Inc., for their generosity in providing a development machine on which I could test PL/Vision code for Release 2.3 of PL/SQL at a critical stage in this book.

I thank Petra Smulders of the European Oracle User Group, Lex de Haan of Oracle EMEA Education, and Ineke Werkman of Oracle Netherlands for their parts in my two wonderful trips to the Netherlands in 1996, where I trained Oracle developers and enjoyed Amsterdam.

I thank Peter Vasterd of Oracle Corporation for keeping a wide line of communication open with me, answering my sometimes, well, brainless questions, helping me get around technical and bureaucratic obstacles, and specifically making sure that I was able to build and test my software on an Oracle Server 7.3 database.

I thank Tom White and Steve Hilker of RevealNet for their enthusiastic support of my work and their help in making PL/Vision a living and breathing product.

I thank the various editors who have published my articles over the last year: Tony Ziemba of *Oracle Developer* (portions of this book appeared originally as articles in that publication), Jerry Coffee of *Oracle Informant*, Kathleen O'Connor of *Oracle Technical Journal*, Rich Niemiec of *Select Magazine*, and the folks at the *Oracle Integrator*. Writing these articles helped me focus my thinking and coding in ways that clarified the contents of this book.

I thank Ernie Martinez and Mark A. Ebel of COM.sortium LLC in Orlando for their help in sorting out my difficulties with UTL_FILE on my Windows NT Oracle 7.3 Server.

I thank Fred King, most able, industrious, and downright amiable computer serviceperson, who went beyond the call of duty to cannibalize a "spare" Toshiba Portege in order to retrieve the text of this book from my dead laptop and keep me writing.

I thank the good people at O'Reilly & Associates for another fine publishing experience. I thank Debby Russell, my editor, for another outstanding effort at transforming lots of interesting ideas and too much text into a text of manageable size and cohesiveness. I do not really understand how she manages to juggle all of her editing and book development responsibilities so effectively. I do know, however, that this book stands as yet another testimony to her skill and dedication. Thanks as well to David Futato, the production manager for the book; Kismet McDonough−Chan, the copyeditor; Mike Sierra, who converted the Microsoft Word files to FrameMaker; Edie Freedman, who designed the cover; Nancy Priest and Mary Jane Walsh, who designed the interior layout; Chris Reilley, who prepared the diagrams; Eden Reiner, who handled the advertising material; and Seth Maislin, who prepared the index.

I thank my wife, Veva Silva Feuerstein, for taking on a bigger−than−usual share (more than big enough to begin with) of "the family thing" and "the house thing" as I buried myself in my computers. I am lucky to have her and I hope she feels the same way about me.

I thank my son, Eli Silva Feuerstein, for never giving up on asking me to play with him, even as I was buried in inches of paper doing final edits. The games of horse and one−on−one in our tiny backyard basketball court definitely improved the text you read in this book.

I thank my son, Chris Silva, for helping us get away from our pets now and then, and for being such a fine big brother to Eli.

I thank my Uncle Dave Gventer for making clear to me early in life that just because most people believe one thing, that doesn't make it true. He opened my eyes to a different way of looking at the world.

Finally, I thank my mother and father, Joan Lee and Sheldon Feuerstein, for their support and love.

PREVIOUS | HOME | NEXT
Comments and Questions | BOOK INDEX | I. Working With Packages

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference