10TH ANNIVERSARY EDITION

4th Edition

Covers Versions Through Oracle Database 10g Release 2

Oracle PL/SQL Programming

Steven Feuerstein
with Bill Pribyl

O'REILLY®

CHAPTER 20

Managing PL/SQL Code

Writing the code for an application is just one step toward putting that application into production and then maintaining the code base. It is not possible within the scope of this book to fully address the entire life cycle of application design, development, and deployment. We do have room, however, to offer some ideas and advice about the following topics:

Managing and analyzing code in the database
: When you compile PL/SQL programs, the source code is loaded into the data dictionary in a variety of forms (the text of the code, dependency relationships, parameter information, etc.). You can therefore use SQL to query these dictionary views to help you manage your code base.

Using native compilation
: Beginning with Oracle9*i* Database, PL/SQL source code may optionally be compiled into native object code that is linked into Oracle. Native compilation can noticeably improve overall application performance (its impact is felt in compute-intensive programs, but does not affect SQL performance).

Using the optimizing compiler and compile-time warnings
: Oracle Database 10g Release 1 added significant new and transparent capabilities to the PL/SQL compiler. The compiler will now automatically optimize your code, often resulting in substantial improvements in performance. In addition, the compiler will provide warnings about your code that will help you improve its readability, performance, and/or functionality.

Testing PL/SQL programs
: This section offers suggestions for PL/SQL program testing based on the open source unit-testing framework, utPLSQL.

Debugging PL/SQL programs
: Many development tools now offer graphical debuggers based on Oracle's DBMS_DEBUG API. These provide the most powerful way to debug programs, but they are still just a small part of the overall debugging process. This section

also discusses program tracing and explores some of the techniques and (dare I say) philosophical approaches you should utilize to debug effectively.

Tuning PL/SQL programs
: This section offers a roundup of some of the more useful and generally applicable tuning tips, along with instructions for how you can analyze your program's execution with built-in profiling and tracing utilities.

Protecting stored code
: Oracle offers a way to "wrap" source code so that confidential and proprietary information can be hidden from prying eyes. This feature is most useful to vendors who sell applications based on PL/SQL stored code.

Managing Code in the Database

When you CREATE OR REPLACE a PL/SQL program, the source code for that program, along with other representations of that software, is stored in the database itself and exposed through a wide range of data dictionary views. This is a tremendous advantage for two key reasons:

Information about that code is available to you via the SQL language
: I can write queries and even entire PL/SQL programs to read the contents of these data dictionary views and obtain lots of fascinating and useful information about my code base.

The database manages dependencies between your stored objects
: For example, if a stored function relies on a certain table, and that table's structure is changed, the status of that function is automatically set to INVALID. Recompilation then takes place automatically when someone tries to execute that function.

This SQL interface to your code base allows you to manage your code repository—running analyses on your code, documenting what has been written and changed, and so on. The following sections introduce you to some of the most commonly accessed sources of information in the data dictionary.

Data Dictionary Views for PL/SQL Programmers

The Oracle data dictionary is a jungle—lushly full of incredible information, but often with less than clear pathways to your destination. There are hundreds of views built on hundreds of tables, many complex interrelationships, special codes, and, all too often, nonoptimized view definitions. A subset of this multitude is particularly handy to PL/SQL developers; we will take a closer look at the key views in a moment. First, it is important to know that there are three types or levels of data dictionary views:

*USER_**
: Views that show information about the database objects owned by the currently connected schema.

*ALL_**
: Views that show information about all of the database objects to which the currently connected schema has access (either because it owns them or because it has been granted access to them). Generally they have the same columns as the corresponding USER view, with the addition of an OWNER column in the ALL views.

*DBA_**
: Views that show information about all the objects in the database. Generally the same columns as the corresponding ALL view.

We will work with the USER views in this chapter; you can easily modify any scripts and techniques to work with the ALL views by adding an OWNER column to your logic. The following are some views a PL/SQL developer is most likely to find useful:

USER_DEPENDENCIES
: The dependencies to and from objects you own. This view is mostly used by Oracle to mark objects INVALID when necessary, and also by IDEs to display the dependency information in their object browsers.

USER_ERRORS
: The current set of errors for all stored objects you own. This view is accessed by the SHOW ERRORS SQL*Plus command, described in Chapter 2. You can, however, write your own queries against it as well.

USER_OBJECTS
: The objects you own. You can, for instance, use this view to see if an object is marked INVALID, find all the packages that have "EMP" in their names, etc.

USER_OBJECT_SIZE
: The size of the objects you own. Actually, this view will show you the source, parsed, and compile sizes for your code. Use it to identify the large programs in your environment, good candidates for *pinning* into the SGA.

USER_PLSQL_OBJECT_SETTINGS
: (Introduced in Oracle Database 10*g* Release 1) Information about the characteristics of a PL/SQL object that can be modified through the ALTER and SET DDL commands, such as the optimization level, debug settings, and more.

USER_PROCEDURES
: (Introduced in Oracle9*i* Database Release 1) Information about stored programs, such as the AUTHID setting, whether the program was defined as DETERMINISTIC, and so on.

USER_SOURCE
: The text source code for all objects you own (in Oracle9*i* Database and above, including database triggers and Java source). This is a very handy view, because you can run all sorts of analysis of the source code against it using SQL and, in particular, Oracle Text.

USER_TRIGGERS and USER_TRIG_COLUMNS
: The database triggers you own, and any columns identified with the triggers. You can write programs against this view to enable or disable triggers for a particular table.

USER_ARGUMENTS
: The arguments (parameters) in all the procedures and functions in your schema.

You can view the structures of each of these views either with a DESCRIBE command in SQL*Plus or by referring to the appropriate Oracle documentation. The following sections provide some examples of the ways you can use these views.

Display information about stored objects

The USER_OBJECTS view contains the following key information about an object:

OBJECT_NAME
: Name of the object

OBJECT_TYPE
: Type of the object (e.g., 'PACKAGE', 'FUNCTION', 'TRIGGER')

STATUS
: Status of the object: VALID or INVALID

LAST_DDL_TIME
: Timestamp indicating the last time that this object was changed.

The following SQL*Plus script displays the status of PL/SQL code objects:

```
/* File on web: psobj.sql */
SET PAGESIZE 66
COLUMN object_type FORMAT A20
COLUMN object_name FORMAT A30
COLUMN status FORMAT A10
BREAK ON object_type SKIP 1
SPOOL psobj.lis
SELECT object_type, object_name, status
  FROM user_objects
 WHERE object_type IN (
    'PACKAGE', 'PACKAGE BODY', 'FUNCTION', 'PROCEDURE',
    'TYPE', 'TYPE BODY', 'TRIGGER')
 ORDER BY object_type, status, object_name
/
SPOOL OFF
```

The output from this script file contains the following list:

```
OBJECT_TYPE         OBJECT_NAME                    STATUS
------------------- ------------------------------ ----------
FUNCTION            DEVELOP_ANALYSIS               INVALID
                    NUMBER_OF_ATOMICS              INVALID

PACKAGE             CONFIG_PKG                     VALID
                    EXCHDLR_PKG                    VALID
```

Notice that a two of my modules are marked as INVALID. See the section "Recompiling Invalid Code" for more details on the significance of this setting and how you can change it to VALID.

Display and search source code

You should always maintain the source code of your programs in text files (or via a development tool specifically designed to store and manage PL/SQL code outside of the database). When you store these programs in the database, however, you can take advantage of SQL to analyze your source code across all modules, which may not be a straightforward task with your text editor.

The USER_SOURCE view contains all of the source code for objects owned by the current user. The structure of USER_SOURCE is as follows:

```
Name                           Null?    Type
------------------------------ -------- ----
NAME                           NOT NULL VARCHAR2(30)
TYPE                                    VARCHAR2(12)
LINE                           NOT NULL NUMBER
TEXT                                    VARCHAR2(4000)
```

where:

NAME
: Is the name of the object

TYPE
: Is the type of the object (ranging from PL/SQL program units to Java source to trigger source)

LINE
: Is the line number

TEXT
: Is the text of the source code

USER_SOURCE is a very valuable resource for developers. With the right kind of queries, you can do things like:

- Display source code for a given line number
- Validate coding standards

- Identify possible bugs or weaknesses in your source code
- Look for programming constructs not identifiable from other views

Suppose, for example, that we have set as a rule that individual developers should never hardcode one of those application-specific error numbers between –20,999 and –20,000 (such hardcodings can lead to conflicting usages and lots of confusion). I can't stop a developer from writing code like this:

```
RAISE_APPLICATION_ERROR (-20306, 'Balance too low');
```

but I can create a package that allows me to identify all the programs that have such a line in them. I call it my "validate standards" package; it is very simple, and its main procedure looks like this:

```
/* Files on web: valstd.pks, valstd.pkb */
PROCEDURE progwith (str IN VARCHAR2)
IS
   TYPE info_rt IS RECORD (
      NAME   user_source.NAME%TYPE
    , text   user_source.text%TYPE
   );

   TYPE info_aat IS TABLE OF info_rt
      INDEX BY PLS_INTEGER;

   info_aa   info_aat;
BEGIN
   SELECT NAME || '-' || line
        , text
   BULK COLLECT INTO info_aa
     FROM user_source
    WHERE UPPER (text) LIKE '%' || UPPER (str) || '%'
      AND NAME <> 'VALSTD'
      AND NAME <> 'ERRNUMS';

   disp_header ('Checking for presence of "' || str || '"');

   FOR indx IN info_aa.FIRST .. info_aa.LAST
   LOOP
      pl (info_aa (indx).NAME, info_aa (indx).text);
   END LOOP;
END progwith;
```

Once this package is compiled into my schema, I can check for usages of –20,*NNN* numbers with this command:

```
SQL> EXEC valstd.progwith ('-20')
==================
VALIDATE STANDARDS
==================
Checking for presence of "-20"

CHECK_BALANCE - RAISE_APPLICATION_ERROR (-20306, 'Balance too low');
```

```
MY_SESSION -    PRAGMA EXCEPTION_INIT(dblink_not_open,-2081);
VSESSTAT - CREATE DATE    : 1999-07-20
```

Notice that the second and third lines in my output are not really a problem; they show up only because I couldn't define my filter narrowly enough.

This is a fairly crude analytical tool, but you could certainly make it more sophisticated. You could also have it generate HTML that is then posted on your intranet. You could then run the valstd scripts every Sunday night through a DBMS_JOB-submitted job, and each Monday morning developers could check the intranet for feedback on any fixes needed in their code.

Use program size to determine pinning requirements

The USER_OBJECT_SIZE view gives you the following information about the size of the programs stored in the database:

SOURCE_SIZE
Size of the source in bytes. This code must be in memory during compilation (including dynamic/automatic recompilation).

PARSED_SIZE
Size of the parsed form of the object in bytes. This representation must be in memory when any object that references this object is compiled.

CODE_SIZE
Code size in bytes. This code must be in memory when the object is executed.

Here is a query that allows you to show code objects that are larger than a given size. You might want to run this query to identify the programs that you will want to pin into the database using DBMS_SHARED_POOL (see Chapter 23 for more information on this package) in order to minimize the swapping of code in the SGA:

```
/* File on web: pssize.sql */
SELECT name, type, source_size, parsed_size, code_size
  FROM user_object_size
 WHERE code_size > &&1 * 1024
 ORDER BY code_size DESC
/
```

Obtain properties of stored code

The USER_PLSQL_OBJECT_SETTINGS (introduced in Oracle Database 10*g* Release 1) view provides information about the following compiler settings of a stored PL/SQL object:

PLSQL_OPTIMIZE_LEVEL
Optimization level that was used to compile the object

PLSQL_CODE_TYPE
Compilation mode for the object

PLSQL_DEBUG
Indicates whether or not the object was compiled for debugging

PLSQL_WARNINGS
Compiler warning settings that were used to compile the object

NLS_LENGTH_SEMANTICS
NLS length semantics that were used to compile the object

Possible uses for this view include:

- Identify any programs that are not taking full advantage of the optimizing compiler (an optimization level of 1 or 0):

```
/* File on web: low_optimization_level.sql */
SELECT owner, name
  FROM user_plsql_object_settings
 WHERE plsql_optimize_level IN (1,0);
```

- Determine if any stored programs have disabled compile-time warnings:

```
/* File on web: disable_warnings.sql */
SELECT NAME, plsql_warnings
  FROM user_plsql_object_settings
 WHERE plsql_warnings LIKE '%DISABLE%';
```

The USER_PROCEDURES view lists all functions and procedures, along with associated properties, including whether a function is pipelined, parallel enabled, or aggregate. USER_PROCEDURES will also show you the AUTHID setting for a program (DEFINER or CURRENT_USER). This can be very helpful if you need to see quickly which programs in a package or group of packages use invoker rights or definer rights. Here is an example of such a query:

```
/* File on web: show_authid.sql */
SELECT   AUTHID
       , p.object_name program_name
       , procedure_name subprogram_name
    FROM user_procedures p, user_objects o
   WHERE p.object_name = o.object_name
     AND p.object_name LIKE '<package or program name criteria>'
ORDER BY AUTHID, procedure_name;
```

Analyze and modify trigger state through views

Query the trigger-related views (USER_TRIGGERS, USER_TRIG_COLUMNS) to do any of the following:

- Enable or disable all triggers for a given table. Rather than have to write this code manually, you can execute the appropriate DDL statements from within a PL/SQL program. See the section "Maintaining Triggers" in Chapter 19 for an example of such a program.
- Identify triggers that execute only when certain columns are changed, but do not have a WHEN clause. A best practice for triggers is to include a WHEN clause

to make sure that the specified columns actually *have* changed values (rather than simply writing the same value over itself).

Here is a query you can use to identify potentially problematic triggers lacking a WHEN clause:

```
/* File on web: nowhen_trigger.sql */
SELECT *
  FROM user_triggers tr
 WHERE when_clause IS NULL AND
      EXISTS (SELECT 'x'
                FROM user_trigger_cols
               WHERE trigger_owner = USER
               AND trigger_name = tr.trigger_name);
```

Analyze argument information

A *very* useful view for programmers is USER_ARGUMENTS. It contains information about each of the arguments of each of the stored programs in your schema. It offers, simultaneously, a wealth of nicely parsed information about arguments and a bewildering structure that is very hard to work with.

Here is a simple SQL*Plus script to dump the contents of USER_ARGUMENTS for all the programs in the specified package:

```
/* File on web: desctest.sql */
SELECT object_name, argument_name, overload
     , POSITION, SEQUENCE, data_level, data_type
  FROM user_arguments
 WHERE package_name = UPPER ('&&1');
```

A more elaborate PL/SQL-based program for displaying the contents of USER_ARGUMENTS may be found in the *show_all_arguments.sp* file on the book's web site.

You can also write more specific queries against USER_ARGUMENTS to identify possible quality issues with your code base. For example, Oracle recommends that you stay away from the LONG datatype and instead use LOBs. In addition, the fixed-length CHAR datatype can cause logic problems; you are much better off sticking with VARCHAR2. Here is a query that uncovers the usage of these types in argument definitions:

```
/* File on web: long_or_char.sql */
SELECT object_name, argument_name, overload
     , POSITION, SEQUENCE, data_level, data_type
  FROM user_arguments
 WHERE data_type IN ('LONG','CHAR');
```

You can even use USER_ARGUMENTS to deduce information about a package's program units that is otherwise not easily obtainable. Suppose that I want to get a list of all the procedures and functions defined in a package specification. You will say: "No problem! Just query the USER_PROCEDURES view." And that would be a fine answer, except that it turns out that USER_PROCEDURES doesn't tell you whether a program is a function or procedure (in fact, it can be *both*, depending on how the program is overloaded!).

You might instead, want to turn to USER_ARGUMENTS. It does, indeed, contain that information, but it is far less than obvious. To determine whether a program is a function or a procedure, you must check to see if there is a row in USER_ARGUMENTS for that package-program combination that has a POSITION of 0. That is the value Oracle uses to store the RETURN "argument" of a function. If it is not present, then the program must be a procedure.

The following function uses this logic to return a string that indicates the program type (if it is overloaded with both types, the function returns "FUNCTION, PROCEDURE"). Note that the list_to_string function used in the main body is provided in the file.

```
/* File on web: program_type.sf */
CREATE OR REPLACE FUNCTION program_type (
   owner_in     IN   VARCHAR2
 , package_in   IN   VARCHAR2
 , program_in   IN   VARCHAR2
)
   RETURN VARCHAR2
IS
   TYPE overload_aat IS TABLE OF all_arguments.overload%TYPE
      INDEX BY PLS_INTEGER;

   l_overloads  overload_aat;
   retval       VARCHAR2 (32767);

BEGIN
   SELECT   DECODE (MIN (POSITION), 0, 'FUNCTION', 'PROCEDURE')
   BULK COLLECT INTO l_overloads
       FROM all_arguments
      WHERE owner = owner_in
        AND package_name = package_in
        AND object_name = program_in
   GROUP BY overload;

   IF l_overloads.COUNT > 0
   THEN
      retval := list_to_string (l_overloads, ',', distinct_in => TRUE);
   END IF;

   RETURN retval;
END program_type;
/
```

Finally, you should also know that the built-in package, DBMS_DESCRIBE, provides a PL/SQL API to provide much of the same information as USER_

ARGUMENTS. There are differences, however, in the way these two elements handle datatypes.

Recompiling Invalid Code

Whenever a change is made to a database object, Oracle uses its dependency-related views (such as PUBLIC_DEPENDENCIES) to identify all objects that depend on the changed object. It then marks those dependent objects as INVALID, essentially throwing away any compiled code. This all happens automatically and is one of the clear advantages to compiling programs into the database. The code will then have to be recompiled before it can be executed.

Oracle will automatically attempt to recompile invalid programs as they are called. You can also manually recompile your invalid code, and this section shows how you can do this. Manual recompilation is generally recommended over automatic recompilation, particularly when it involves a production application. Recompilation can take quite a long time; on-demand compilation caused by a user request will generally result in a high level of user frustration.

Recompile individual program units

You can use the ALTER command to recompile a single program. Here are examples of using this DDL command:

```
ALTER FUNCTION a_function COMPILE REUSE SETTINGS;
ALTER PACKAGE my_package COMPILE REUSE SETTINGS;
ALTER PACKAGE my_package COMPILE SPECIFICATION REUSE SETTINGS;
ALTER PACKAGE my_package COMPILE BODY REUSE SETTINGS;
```

You should include the REUSE SETTINGS clause so that other settings for this program (such as compile-time warnings and optimization level) are not inadvertently set to the settings of the current session.

Of course, if you have many invalid objects, you will not want to type ALTER COMPILE commands for each one. You could write a simple query, like the one below, to *generate* all the ALTER commands:

```
SELECT 'ALTER ' || object_type || ' ' || object_name
       || ' COMPILE REUSE SETTINGS;'
  FROM user_objects
 WHERE status = 'INVALID';
```

The problem with this "bulk" approach is that as you recompile one invalid object, you may cause many others to be marked INVALID. You are much better off relying on Oracle's own utilities to recompile entire schemas or to use a sophisticated, third-party script created by Solomon Yakobson. These are described in the next section.

Use UTL_RECOMP

Starting with Oracle Database 10*g* Release 1, the UTL_RECOMP built-in package offers two programs that you can use to recompile any invalid objects in your schema: RECOMP_SERIAL and RECOMP_PARALLEL.

To use UTL_RECOMP, you will need to connect as a SYSDBA account. When running the parallel version, it uses the DBMS_JOB package to queue up the recompile jobs. When this happens, all other jobs in the queue are temporarily disabled to avoid conflicts with the recompilation.

Here is an example of calling the serial version to recompile all invalid objects in the SCOTT schema:

```
SQL> CALL utl_recomp.recomp_serial ('SCOTT');
```

If you have multiple processors, the parallel version may help you complete your recompilations more rapidly. As Oracle notes in its documentation of this package, however, compilation of stored programs results in updates to many catalog structures and is I/O intensive; the resulting speedup is likely to be a function of the speed of your disks.

Here is an example of requesting recompilation of all invalid objects in the SCOTT schema, using up to four simultaneous threads for the recompilation steps:

```
SQL> CALL utl_recomp.recomp_parallel ('SCOTT', 4);
```

Oracle also offers the DBMS_UTILITY.RECOMPILE_SCHEMA program to recompile invalid objects. One advantage of using this program over the UTL_RECOMP alternatives is that you do not need to connect as a SYSDBA account. I recommend, however, that you avoid using DBMS_UTILITY.RECOMPILE_SCHEMA altogether; in some cases, it does not seem to successfully recompile all invalid objects. This may have to do with the order in which it performs the compilations.

If you do not want to have to connect to a SYSDBA account to perform your recompilations, you might consider using a recompile utility written by Solomon Yakobson and found in the *recompile.sql* file on the book's web site.

Using Native Compilation

In versions before Oracle9*i* Database Release 1, compilation of PL/SQL source code always resulted in a representation, usually referred to as *bytecode* or *mcode,* that is stored in the database and interpreted at runtime by a virtual machine implemented within Oracle. Oracle9*i* Database introduced a new approach: PL/SQL source code may optionally be compiled into native object code that is linked into Oracle. (Note, however, that an anonymous PL/SQL block is *never* compiled natively.)

When would this feature come in handy? How do you turn on native compilation? This section addresses these questions.

PL/SQL is often used as a thin wrapper for executing SQL statements, setting bind variables, and handling result sets. For these kinds of programs, the execution speed of the PL/SQL code is rarely an issue; it is the execution speed of the SQL that determines the performance. The efficiency of the context switch between the PL/SQL and the SQL operating environments might be a factor, but this is addressed very effectively by the FORALL and BULK COLLECT features introduced in Oracle8*i* Database and described in Chapter 14.

There are many other applications and programs, however, that rely on PL/SQL to perform computationally intensive tasks that are independent of the SQL engine. PL/SQL is, after all, a fully functional procedural language, and almost any real-world code is going to include a certain amount of "low-hanging fruit" that a modern compiler can chomp through, resulting in at least *some* increase in speed. You should realize, however, that the way that Oracle has chosen to implement the native compilation feature is not simply "translate your PL/SQL source into C and then compile it;" instead, Oracle always runs the normal PL/SQL compiler to generate mcode, and in native mode it takes this mcode itself as its input into the C translation process. This architecture has several consequences:

- Generating natively compiled code is by "definition" slower than generating conventional code.
- Any optimizations taken by the PL/SQL compiler will be applied regardless of compilation mode.
- The generated C code is going to be incomprehensible to anyone other than a few rocket scientists who work at Oracle Corporation (normally, the C source code is automatically deleted).

The tasks expressed in C are primarily housekeeping tasks: setting up and destroying temporary variables; managing stack frames for subprogram invocation; and making calls to Oracle's appropriate internal routines. Speedup from using C will be greatest in programs that spend more time processing the mcode relative to the time spent in Oracle's internal routines. To be sure, that's difficult or impossible for customers to predict, but there are even more factors in the speedup equation, including:

- Oracle version in use. Later versions tend to exhibit more efficient runtime engines, which suppress the *relative* benefit of native compilation, although the *total* speedup will be greater.
- Setting of PLSQL_OPTIMIZE_LEVEL (Oracle Database 10*g*). If you are using aggressive PL/SQL optimization, the relative speedup from native compilation will be lower.

- Selection of datatypes. For example, a compute-intensive program that makes extensive use of the new IEEE floating-point types may also exhibit less relative speedup from native compilation.
- Behavior and optimizing capabilities of the C compiler you are using, plus effects that may vary based on your particular hardware.
- Degree of mixing native and interpreted code. Callouts between native and interpreted program units involve a context switch that homogeneous callouts can avoid.

Native compilation gives a broad range of speed increase, quoted by some sources as "up to 40%," but even higher in certain unusual cases. Fortunately—and this is significant—I have never seen native compilation *degrade* runtime performance. That means the only thing you stand to lose with native compilation is speed of the compilation itself.

So how do you turn on this nifty feature? Read on…

Perform One-Time DBA Setup

Native PL/SQL compilation is achieved by translating the PL/SQL source code into C source code that is then compiled on the same host machine running the Oracle server. The compiling and linking of the generated C source code is done by tools external to Oracle that are set up by the DBA and/or system administrator.

Enabling native PL/SQL compilation in Oracle Database 10g can be accomplished in as few as three steps:

1. Get a supported C compiler.
2. Set up directory(ies) in the filesystem that will hold the natively compiled files.
3. Check *$ORACLE_HOME/plsql/spnc_commands*.

Step 1: Get a Supported C Compiler

If you don't already have your platform vendor's usual C compiler, you'll have to get one from somewhere. Fortunately, this does not always require a huge investment; if you happen to be running Oracle Database 10g Release 2, you can use the freely downloadable GNU C compiler. Table 20-1 shows just a few of the combinations of compiler, version, and platform that Oracle supports. For the complete list, go to Oracle's Metalink site and search for the "Certified Compilers" document (doc ID 43208.1).

Table 20-1. Sampling of C compilers required by native compilation

Platform	Oracle Database version(s)	Supported C compiler(s)
Sun Sparc Solaris	9.2	Sun Forte Workshop 6.2 (with particular patches from Sun); *spnc_command.mk* includes gcc-specific comments, but GCC doesn't appear to be officially supported
	10.1	Sun ONE Studio 8, C/C++ 5.5
	10.2	Same as above, plus GCC 3.4
Microsoft Windows 2000, XP, 2003	9.2	Microsoft Visual C++ 6.0
	10.1	Microsoft Visual C++ 6.0 Microsoft Visual C++ .NET 2002 Microsoft Visual C++ .NET 2003
	10.2	Same as above, plus MinGW GCC 3.2.3[a]
Linux Intel 32bit	9.2	GNU GCC 2.95.3
	10.1	Red Hat Linux 2.1: GNU GCC 2.96.108.1 Red Hat Linux 3: GNU GCC 3.2.3-2 UnitedLinux 1.0: GNU GCC 3.2.2-38 Vendor-independent: Intel C++ 7.1.0.28
	10.2	Red Hat: GCC 3.2.3-34 Suse: GCC 3.3.3-43 Vendor-independent: Intel C++ Compiler v7.1.0.28

[a] Obtained by installing *MinGW-3.1.0-1.exe* from *http://www.mingw.org*

With the right combination of luck and spare time, you *may* be able to get an unsupported compiler to work for native compilation; if you have trouble, though, all Oracle will do is tell you to get a certified compiler. I know that some sites have been able to use GCC on Sun Sparc Solaris with Oracle9*i* Database, but others had trouble until they got Sun's compiler. And I have never heard of anyone getting GCC working with Oracle Database 10*g* Release 1.

By the way, you cannot reuse the generated object files on another machine, even if it's the exact same version of the OS and Oracle; you can't even copy the object files to a different database on the same machine. The object files contain database-specific information and must be generated on the exact same database and machine that will ultimately run the files. Besides, you might have a DDL event that triggers some automatic recompiles. You will, therefore, need a C compiler on every machine on which you want to use this feature. And, if you happen to be running an Oracle Real Application Cluster (RAC), you'll need to install your C compiler on each node.

Step 2: Set Up the Directories

When Oracle translates your PL/SQL into C and runs it through the host compiler, the resulting object files have to go somewhere on the server filesystem. Curiously, there is no default for this location; the DBA must create the directories and set one or two initialization parameters. Here is a simple case:

```
# While logged in as oracle (to get the correct ownership):
$ mkdir /u01/app/oracle/oracle/product/10.2.0/db_1/dbs/ncomps

$ sqlplus "/ as sysdba"
...
SQL> ALTER SYSTEM SET plsql_native_library_dir =
  2  '/u01/app/oracle/oracle/product/10.2.0/db_1/dbs/ncomps';
```

Some filesystems start to choke on a few thousand files in a single directory; to support that many modules, you can get Oracle to spread the object files across many subdirectories. To use 1,000 subdirectories, specify:

```
SQL> ALTER SYSTEM SET plsql_native_library_subdir_count = 1000;
```

You will also need to precreate the subdirectories, which in this case must be named d0, d1, d2...d999. Do this to generate a directory-creating script (using a variation on Oracle's suggested method):

```
SET DEFINE OFF
SPOOL makedirs.sh
BEGIN
  FOR dirno IN 0..999
  LOOP
    DBMS_OUTPUT.PUT_LINE('mkdir d' || dirno || ' && echo ' || dirno);
  END LOOP;
END;
/
SPOOL OFF
```

Then, edit out the cruft at the top and bottom of the script, and at the operating system prompt, do something like this:

```
$ cd /u01/app/oracle/oracle/product/10.2.0/db_1/dbs/ncomps
$ sh makedirs.sh
```

Starting with Oracle Database 10g Release 1, the master copy of the object files is really BLOB data in a table named *ncomp_dll$*; the on-disk copy exists so it can be dynamically loaded by the operating system. With this capability, Oracle can regenerate the on-disk copies without recompiling the source, but you still don't want to delete any of the generated files unless your database is shut down.

Step 3: Check $ORACLE_HOME/plsql/spnc_commands

Oracle Database 10g invokes the C compiler by calling a script named *spnc_commands* (*spnc* stands for "stored procedure native compilation," presumably). This file differs by platform, and in some cases includes inline comments indicating how to use different compilers. You'll want to inspect this file to see if the path to the compiler executable is correct for your installation.

If you're running Oracle9*i* Database, there is a file named *spnc_makefile.mk* that you will need to inspect instead; that version has a more complicated setup for native compilation (see the sidebar "Native Compilation Prior to Oracle Database 10g").

Native Compilation Prior to Oracle Database 10g

With native compilation in Oracle9*i* Database, the setup is slightly different, and there are several limitations that you'll want to keep in mind:

- In addition to getting a supported C compiler, setting the directory parameters, and inspecting the *spnc_makefile.mk* as discussed above, there are extra parameters required at the system or session level: PLSQL_NATIVE_MAKE_UTILITY (typically *make*) and PLSQL_NATIVE_MAKE_FILE_NAME (the fully qualified path to *spnc_makefile.mk*).
- While a CREATE or REPLACE of a module replaces the existing DLL, a DROP of a module does not delete the DLL from the filesystem. You will need to remove these files manually.
- There is no storage of the content of the object file in the data dictionary as in Oracle Database 10*g*. You definitely want to adjust your system-level backup to include the native file directories.
- Package specifications and package bodies must match in compilation style (both interpreted, or both native). In Oracle Database 10*g*, they don't have to match.

Native compilation does take longer than interpreted mode compilation; our tests have shown an increase of a factor of about two. That's because native compilation involves several extra steps: generating C code from the initial output of the PL/SQL compilation, writing this to the filesystem, invoking and running the C compiler, and linking the resulting object code into Oracle.

Interpreted Versus Native Compilation Mode

After your administrator sets everything up, you are ready to go native. The first thing to do is set the compiler parameter. A user may set it as described here.

In Oracle9*i* Database, specify:

```
ALTER SESSION
   SET plsql_compiler_flags = 'NATIVE';    /* vs. 'INTERPRETED' */
```

Starting with Oracle Database 10*g* Release 1, the PLSQL_COMPILER_FLAGS parameter is deprecated, so you should use this instead:

```
ALTER SESSION
   SET plsql_code_type = 'NATIVE';    /* vs. 'INTERPRETED' */
```

The compilation mode will then be set for subsequently compiled PL/SQL library units during that session, as long as they are compiled by one of the following:

- A script or explicit CREATE [OR REPLACE] command
- An ALTER...COMPILE statement
- The DBMS_UTILITY.COMPILE_SCHEMA packaged procedure

In addition, the DBA can change the mode on a system-wide basis using ALTER SYSTEM.

Oracle stores the compilation mode with the library unit's metadata so that if the program is implicitly recompiled as a consequence of dependency checking, the last mode used will be used again. Note that this "stickiness" applies *only* to automatic recompilations; other rebuilds or recompiles will use the session's current setting. You can determine the saved compilation mode for your stored programs by querying the data dictionary using the statement shown here (for Oracle Database 10*g*):

```
SELECT name, type, plsql_code_type
  FROM USER_PLSQL_OBJECT_SETTINGS
  ORDER BY name;
```

The result will show something like this:

```
NAME                           TYPE         PLSQL_CODE_TYPE
------------------------------ ------------ ---------------------
ANIMAL_HIST_TRG                TRIGGER      NATIVE
DEMO                           PACKAGE BODY INTERPRETED
DEMO                           PACKAGE      INTERPRETED
ORDER_SEEDS                    PROCEDURE    NATIVE
PLVTMR                         PACKAGE      NATIVE
PLVTMR                         PACKAGE BODY NATIVE
PRINTANY                       FUNCTION     INTERPRETED
```

In Oracle9*i* Database, the WHERE clause would instead look for PLSQL_COMPILER_FLAGS, and you would get additional information about whether the unit has been compiled with debug mode.

Incidentally, PL/SQL debuggers will not work with natively compiled programs. This is one of the only downsides to native compilation, but in most cases you could work around it by using interpreted mode during development, and native mode in testing and production.

Oracle recommends that all of the PL/SQL library units called from a given top-level unit be compiled in the same mode (see the sidebar "Converting an Entire Database to Native (or Interpreted)"). That's because there is a cost for the context switch when a library unit compiled in one mode invokes one compiled in the other mode. Significantly, this recommendation includes the Oracle-supplied library units. These are always shipped compiled in interpreted mode because they may need to get recompiled during subsequent upgrades, and Oracle cannot assume that you have installed a supported C compiler.

Our conclusion? If your application contains a significant amount of compute-intensive logic, consider switching your entire database—including Oracle's supplied library units—to use native compilation. Making such a change is likely to offer the most dramatic performance improvements for applications that are unable to take advantage of the optimizing compiler introduced in Oracle Database 10*g*.

Converting an Entire Database to Native (or Interpreted)

The simplest way to follow Oracle's recommendation that all PL/SQL library units called from a given top-level unit be compiled in the same mode is to convert the whole database so that all PL/SQL library units are compiled native, and to set the system-wide parameter PLSQL_COMPILER_FLAGS (Oracle9*i* Database) or PLSQL_CODE_TYPE (Oracle Database 10*g*) to NATIVE.

While this is not technically difficult to do, it can be very time-consuming if you have a large number of modules. There are a number of nonobvious steps to the process; we suggest following Oracle's explicit instructions closely.

- For Oracle9*i* Database: *http://otn.oracle.com/tech/pl_sql/htdocs/README_2188517.htm*
- For Oracle Database 10*g*: *http://www.oracle.com/technology/tech/pl_sql/htdocs/ncomp_faq.html#ncomping_db*

The latter link is actually part of a larger FAQ that contains a wealth of useful information on native compilation.

Using the Optimizing Compiler and Compile-Time Warnings

You don't have to make any changes to your code to take advantage of two of the most important enhancements to Oracle Database 10*g* PL/SQL: the optimizing compiler and compile-time warnings.

The Optimizing Compiler

PL/SQL's optimizing compiler can improve runtime performance dramatically, with a relatively slight cost at compile time. The benefits of optimization apply to both interpreted and natively compiled PL/SQL because optimizations are applied by analyzing patterns in source code.

The optimizing compiler is enabled by default. However, you may want to alter its behavior, either by lowering its aggressiveness or by disabling it entirely. For example, if, in the course of normal operations, your system must perform recompilation of many lines of code, or if an application generates many lines of dynamically executed PL/SQL, the overhead of optimization may be unacceptable. Keep in mind, though, that Oracle's tests show that the optimizer doubles the runtime performance of computationally intensive PL/SQL.

In some cases, the optimizer may even alter program behavior. One such case might occur in code written for Oracle9*i* Database that depends on the relative timing of initialization sections in multiple packages. If your testing demonstrates such a problem, yet you wish to enjoy the performance benefits of the optimizer, you may want to rewrite the offending code or to introduce an initialization routine that ensures the desired order of execution.

The optimizer settings are defined through the PLSQL_OPTIMIZE_LEVEL initialization parameter (and related ALTER DDL statements), which can be set to 0, 1, or 2. The higher the number, the more aggressive is the optimization, meaning that the compiler will make a greater effort, and possibly restructure more of your code to optimize performance.

Set your optimization level according to the best fit for your application or program, as follows:

PLSQL_OPTIMIZE_LEVEL = 0
: Zero essentially turns off optimization. The PL/SQL compiler maintains the original evaluation order of statement processing of Oracle9*i* Database and earlier releases. Your code will still run faster than in earlier versions, but the difference will not be so dramatic.

PLSQL_OPTIMIZE_LEVEL = 1
: The compiler will apply many optimizations to your code, such as eliminating unnecessary computations and exceptions. It will not, in general, change the order of your original source code.

PLSQL_OPTIMIZE_LEVEL = 2
: This is the default value and the most aggressive setting. It will apply many modern optimization techniques beyond level 1, and some of those changes may result in moving source code relatively far from its original location. Level 2 optimization offers the greatest boost in performance. It may, however, cause the compilation time in some of your programs to increase substantially. If you encounter this situation (or, alternatively, if you are developing your code and want to minimize compile time, knowing that when you move to production, you will apply the highest optimization level), try cutting back the optimization level to 1.

You can set the optimization level for the instance as a whole, but then override the default for a session or for a particular program. Here are some examples:

```
ALTER SESSION SET PLSQL_OPTIMIZE_LEVEL = 0;
```

Oracle retains optimizer settings on a module-by-module basis. When you recompile a particular module with nondefault settings, the settings will "stick," allowing you to recompile later using REUSE SETTINGS. For example:

```
ALTER PROCEDURE bigproc COMPILE PLSQL_OPTIMIZE_LEVEL = 0;
```

and then:

```
ALTER PROCEDURE bigproc COMPILE REUSE SETTINGS;
```

To view all the compiler settings for your modules, including optimizer level, interpreted versus native, and compiler warning levels, query the USER_PLSQL_OBJECT_SETTINGS view.

For lots more information on the optimizing compiler, see Chapter 23 and visit:

> *http://www.oracle.com/technology/tech/pl_sql/htdocs/new_in_10gr1.htm#faster*

Compile-Time Warnings

Compile-time warnings can greatly improve the maintainability of your code and reduce the chance that bugs will creep into it. Compile-time warnings differ from compile-time errors; with warnings, your program will still compile and run. You may, however, encounter unexpected behavior or reduced performance as a result of running code that is flagged with warnings.

This section explores how compile-time warnings work and which issues are currently detected. Let's start with a quick example of applying compile-time warnings in your session.

A quick example

A very useful compile-time warning is *PLW-06002: Unreachable code*. Consider the following program (available in the *cantgothere.sql* file on the book's web site). Because I have initialized the salary variable to 10,000, the conditional statement will *always* send me to line 9. Line 7 will never be executed.

```
     /* File on web: cantgothere.sql */
 1  CREATE OR REPLACE PROCEDURE cant_go_there
 2  AS
 3     l_salary NUMBER := 10000;
 4  BEGIN
 5     IF l_salary > 20000
 6     THEN
 7        DBMS_OUTPUT.put_line ('Executive');
 8     ELSE
 9        DBMS_OUTPUT.put_line ('Rest of Us');
10     END IF;
11  * END cant_go_there;
```

If I compile this code in any release prior to Oracle Database 10*g* Release 1, I am simply told "Procedure created." If, however, I have enabled compile-time warnings in my session on the new release and then try to compile the procedure, I get this response from the compiler:

```
SP2-0804: Procedure created with compilation warnings

SQL> sho err
Errors for PROCEDURE CANT_GO_THERE:
```

```
LINE/COL ERROR
-------- -------------------------------------
7/7      PLW-06002: Unreachable code
```

Given this warning, I can now go back to that line of code, determine why it is unreachable, and make the appropriate corrections.

If you see a "no message file" message

If you are running 10.1.0.2.0 on Windows, and try to reproduce what I showed in the previous section, you will see this message:

```
7/7      PLW-06002: Message 6002 not found;
            No message file for product=plsql, facility=PLW
```

The problem is that Oracle didn't ship the message file, *plwus.msb*, with the Oracle Database 10*g* software until 10.1.0.3.0, and the download available on OTN is 10.1.0.2.0. If you encounter this problem, you will need to contact Oracle Support to obtain this file (reference Bug 3680132) and place it in the *\plsql\mesg* subdirectory. You will then be able to see the actual warning message.

Verify your SQL*Plus version

If you are running a pre-Oracle Database 10*g* version of SQL*Plus, it will not be able to display warnings; because Oracle9*i* Database did not support compile-time warnings, commands like SHOW ERRORS don't even try to obtain warning information.

Specifically, the ALL_ERRORS family of data dictionary views has two new columns in Oracle Database 10*g*: ATTRIBUTE and MESSAGE_NUMBER. The earlier SQL*Plus versions don't know how to interpret these columns.

To determine if you are using a pre-Oracle Database 10*g* version of SQL*Plus, execute these commands in SQL*Plus:

```
CREATE TABLE t (n BINARY_FLOAT)
/
DESCRIBE t
```

In such versions of SQL*Plus, you will see that "n" is characterized as "UNDEFINED." Starting with Oracle Database 10*g* Release 1, SQL*Plus will properly show the type of this column to be "BINARY_FLOAT."

How to turn on compile-time warnings

Oracle allows you to turn compile-time warnings on and off, and also to specify the type of warnings that interest you. There are three categories of warnings:

Severe
: Conditions that could cause unexpected behavior or actual wrong results, such as aliasing problems with parameters

Performance
: Conditions that could cause performance problems, such as passing a VARCHAR2 value to a NUMBER column in an UPDATE statement

Informational
: Conditions that do not affect performance or correctness, but that you might want to change to make the code more maintainable

Oracle lets you enable/disable compile-time warnings for a specific category, for all categories, and even for specific, individual warnings. You can do this with either the ALTER DDL command or the DBMS_WARNING built-in package.

To turn on compile-time warnings in your system as a whole, issue this command:

```
ALTER SYSTEM SET PLSQL_WARNINGS='string'
```

The following command, for example, turns on compile-time warnings in your system for all categories:

```
ALTER SYSTEM SET PLSQL_WARNINGS='ENABLE:ALL';
```

This is a useful setting to have in place during development because it will catch the largest number of potential issues in your code.

To turn on compile-time warnings in your session for severe problems only, issue this command:

```
ALTER SESSION SET PLSQL_WARNINGS='ENABLE:SEVERE';
```

And if you want to alter compile-time warnings settings for a particular, already-compiled program, you can issue a command like this:

```
ALTER PROCEDURE hello COMPILE PLSQL_WARNINGS='ENABLE:ALL' REUSE SETTINGS;
```

Make sure to include REUSE SETTINGS to make sure that all *other* settings (such as the optimization level) are not affected by the ALTER command.

You can tweak your settings with a very high level of granularity by combining different options. For example, suppose that I want to see all performance-related issues, that I will not concern myself with server issues for the moment, and that I would like the compiler to treat *PLW-05005: function exited without a RETURN* as a compile error. I would then issue this command:

```
ALTER SESSION SET PLSQL_WARNINGS=
  'DISABLE:SEVERE'
 ,'ENABLE:PERFORMANCE'
 ,'ERROR:05005';
```

I especially like this “treat as error” option. Consider the *PLW-05005: function returns without value* warning. If I leave PLW-05005 simply as a warning, then when

I compile my no_return function, shown below, the program does compile, and I *can* use it in my application.

```
SQL> CREATE OR REPLACE FUNCTION no_return
  2     RETURN VARCHAR2
  3  AS
  4  BEGIN
  5     DBMS_OUTPUT.PUT_LINE (
  6        'Here I am, here I stay');
  7  END no_return;
  8  /

SP2-0806: Function created with compilation warnings

SQL> sho err
Errors for FUNCTION NO_RETURN:

LINE/COL ERROR
-------- -----------------------------------------------------------------
1/1      PLW-05005: function NO_RETURN returns without value at line 7
```

If I now alter the treatment of that error with the ALTER SESSION command shown above and then recompile no_return, the compiler stops me in my tracks:

```
Warning: Procedure altered with compilation errors
```

By the way, I could also change the settings for that particular program only, to flag this warning as a "hard" error with a command like this:

```
ALTER PROCEDURE no_return COMPILE PLSQL_WARNINGS = 'error:6002' REUSE SETTINGS
/
```

This ability to treat a warning as an error did not work in 10.1.0.2; this program was fixed in Oracle Database 10*g* Release 2 and is reported to be back-ported to 10.1.0.3.

You can, in each of these variations of the ALTER command, also specify ALL as a quick and easy way to refer to all compile-time warnings categories, as in:

```
ALTER SESSION SET PLSQL_WARNINGS='ENABLE:ALL';
```

Oracle also provides the DBMS_WARNING package, which provides the same capabilities to set and change compile-time warning settings through a PL/SQL API. DBMS_WARNING also goes beyond the ALTER command, allowing you to make changes to those warning controls that you care about while leaving all the others intact. You can also easily restore the original settings when you're done.

DBMS_WARNING was designed to be used in install scripts in which you might need to disable a certain warning, or treat a warning as an error, for individual program units being compiled. You might not have any control over the scripts surrounding those for which you are responsible. Each script's author should be able to set the warning settings he wants, while inheriting a broader set of settings from a more global scope.

Warnings Available in Oracle Database 10g

In the following sections, let's take a look at most of the compile-time warnings that were introduced in Oracle Database 10*g* Release 1. I will offer an example of the type of code that will elicit the warning and also point out some interesting behavior (where present) in the way that Oracle has implemented compile-time warnings.

PLW-05000: mismatch in NOCOPY qualification between specification and body

The NOCOPY compiler hint tells Oracle that, if possible, you would like it to *not* make a copy of your IN OUT arguments. This can improve the performance of programs that pass large data structures, such as collections or CLOBs.

You need to include the NOCOPY hint in both the specification and the body of your program (relevant for packages and object types). If the hint is not present in both, Oracle will apply whatever is specified in the specification.

Here is an example of code that will generate this warning:

```
/* File on web: plw5000.sql */
CREATE OR REPLACE PACKAGE plw5000
IS
   TYPE collection_t IS
      TABLE OF VARCHAR2 (100);

   PROCEDURE proc (
      collection_in IN OUT NOCOPY
         collection_t);
END plw5000;
/
CREATE OR REPLACE PACKAGE BODY plw5000
IS
   PROCEDURE proc (
      collection_in IN OUT
         collection_t)
   IS
   BEGIN
      DBMS_OUTPUT.PUT_LINE ('Hello!');
   END proc;
END plw5000;
/
```

Compile-time warnings will display as follows:

```
SQL> SHOW ERRORS PACKAGE BODY plw5000
Errors for PACKAGE BODY PLW5000:

LINE/COL ERROR
-------- -----------------------------------------------------------------
```

```
3/20     PLW-05000: mismatch in NOCOPY qualification between specification
         and body

3/20     PLW-07203: parameter 'COLLECTION_IN' may benefit from use of the
         NOCOPY compiler hint
```

PLW-05001: previous use of 'string' (at line string) conflicts with this use

This warning will make itself heard when you have declared more than one variable or constant with the same name. It can also pop up if the parameter list of a program defined in a package specification is different from that of the definition in the package body.

You may be saying to yourself: I've seen that error before, but it is a compilation error, not a warning. And, in fact, you are right, in that the following program simply will not compile:

```
CREATE OR REPLACE PROCEDURE plw5001
IS
   a   BOOLEAN;
   a   PLS_INTEGER;
BEGIN
   a := 1;
   DBMS_OUTPUT.put_line ('Will not compile');
END plw5001;
/
```

You receive the following compile error: *PLS-00371: at most one declaration for 'A' is permitted in the declaration section*.

So why is there a *warning* for this situation? Consider what happens when I remove the assignment to the variable named a:

```
SQL> CREATE OR REPLACE PROCEDURE plw5001
  2  IS
  3     a   BOOLEAN;
  4     a   PLS_INTEGER;
  5  BEGIN
  6     DBMS_OUTPUT.put_line ('Will not compile?');
  7  END plw5001;
  8  /
Procedure created.
```

The program compiles! Oracle does not flag the PLS-00371 because I have not actually *used* either of the variables in my code. The PLW-05001 warning fills that gap by giving us a heads-up if we have declared, but not yet used, variables with the same name, as you can see here:

```
SQL> ALTER PROCEDURE plw5001 COMPILE plsql_warnings = 'enable:all';
SP2-0805: Procedure altered with compilation warnings
SQL> SHOW ERRORS
Errors for PROCEDURE PLW5001:
```

```
LINE/COL ERROR
-------- -----------------------------------------------------------------
4/4      PLW-05001: previous use of 'A' (at line 3) conflicts with this use
```

PLW-05003: same actual parameter(string and string) at IN and NOCOPY may have side effects

When you use NOCOPY with an IN OUT parameter, you are asking PL/SQL to pass the argument by reference, rather than by value. This means that any changes to the argument are made immediately to the variable in the outer scope. "By value" behavior (NOCOPY is not specified or the compiler ignores the NOCOPY hint), on the other hand, dictates that changes within the program are made to a local copy of the IN OUT parameter. When the program terminates, these changes are then copied to the actual parameter. (If an error occurs, the changed values are *not* copied back to the actual parameter.)

Use of the NOCOPY hint increases the possibility that you will run into the issue of argument aliasing, in which two different names point to the same memory location. Aliasing can be difficult to understand and debug; a compile-time warning that catches this situation will come in very handy.

Consider this program:

```
/* File on web: plw5003.sql */
CREATE OR REPLACE PROCEDURE very_confusing (
   arg1   IN              VARCHAR2
 , arg2   IN OUT          VARCHAR2
 , arg3   IN OUT NOCOPY   VARCHAR2
)
IS
BEGIN
   arg2 := 'Second value';
   DBMS_OUTPUT.put_line ('arg2 assigned, arg1 = ' || arg1);
   arg3 := 'Third value';
   DBMS_OUTPUT.put_line ('arg3 assigned, arg1 = ' || arg1);
END;
/
```

It's a simple enough program. pass in three strings, two of which are IN OUT; assign values to those IN OUT arguments; and display the value of the first IN argument's value after each assignment.

Now I will run this procedure, passing the very same local variable as the argument for each of the three parameters:

```
SQL> DECLARE
        str   VARCHAR2 (100) := 'First value';
     BEGIN
        DBMS_OUTPUT.put_line ('str before = ' || str);
        very_confusing (str, str, str);
        DBMS_OUTPUT.put_line ('str after = ' || str);
     END;
```

```
  8  /
str before = First value
arg2 assigned, arg1 = First value
arg3 assigned, arg1 = Third value
str after = Second value
```

Notice that while still running very_confusing, the value of the arg1 argument was not affected by the assignment to arg2. Yet when I assigned a value to arg3, the value of arg1 (an IN argument) was changed to "Third value"! Furthermore, when very_confusing terminated, the assignment to arg2 was applied to the str variable. Thus, when control returned to the outer block, the value of the str variable was set to "Second value", effectively writing over the assignment of "Third value".

As I said earlier, parameter aliasing can be very confusing. So, if you enable compile-time warnings, programs such as plw5003 may be revealed to have potential aliasing problems:

```
SQL> CREATE OR REPLACE PROCEDURE plw5003
  2  IS
  3     str   VARCHAR2 (100) := 'First value';
  4  BEGIN
  5     DBMS_OUTPUT.put_line ('str before = ' || str);
  6     very_confusing (str, str, str);
  7     DBMS_OUTPUT.put_line ('str after = ' || str);
  8  END plw5003;
  9  /

SP2-0804: Procedure created with compilation warnings

SQL> sho err
Errors for PROCEDURE PLW5003:

LINE/COL ERROR
-------- -----------------------------------------------------------------
6/4      PLW-05003: same actual parameter(STR and STR) at IN and NOCOPY
         may have side effects

6/4      PLW-05003: same actual parameter(STR and STR) at IN and NOCOPY
         may have side effects
```

PLW-05004: identifier string is also declared in STANDARD or is a SQL built-in

Many PL/SQL developers are unaware of the STANDARD package, and its implications for their PL/SQL code. For example, it is common to find programmers who assume that names like INTEGER and TO_CHAR are reserved words in the PL/SQL language. That is not the case. They are, respectively, a datatype and a function declared in the STANDARD package.

STANDARD is one of the two default packages of PL/SQL (the other is DBMS_STANDARD). Because STANDARD is a default package, you do not need to qualify references to datatypes like INTEGER, NUMBER, PLS_INTEGER, etc., with "STANDARD"—but you could, if you so desired.

PLW-5004 notifies you if you happen to have declared an identifier with the same name as an element in STANDARD (or a SQL built-in; most built-ins—but not all—are declared in STANDARD).

Consider this procedure definition:

```
 1  CREATE OR REPLACE PROCEDURE plw5004
 2  IS
 3     INTEGER   NUMBER;
 4
 5     PROCEDURE TO_CHAR
 6     IS
 7     BEGIN
 8        INTEGER := 10;
 9     END TO_CHAR;
10  BEGIN
11     TO_CHAR;
12  * END plw5004;
```

Compile-time warnings for this procedure will display as follows:

```
LINE/COL ERROR
-------- -----------------------------------------------------------------
3/4      PLW-05004: identifier INTEGER is also declared in STANDARD
         or is a SQL builtin
5/14     PLW-05004: identifier TO_CHAR is also declared in STANDARD
         or is a SQL builtin
```

You should avoid reusing the names of elements defined in the STANDARD package unless you have a very specific reason to do so.

PLW-05005: function string returns without value at line string

This warning makes me happy. A function that does not return a value is a very badly designed program. This is a warning that I would recommend you ask Oracle to treat as an error with the "ERROR:5005" syntax in your PLSQL_WARNINGS setting.

You already saw one example of such a function—no_return. That was a very obvious chunk of code; there wasn't a single RETURN in the entire executable section. Your code will, of course, be more complex. The fact that a RETURN may not be executed could well be hidden within the folds of complex conditional logic.

At least in some of these situations, though, Oracle will *still* detect the problem. The following program demonstrates one of those situations:

```
SQL> CREATE OR REPLACE FUNCTION no_return (
  2     check_in IN BOOLEAN)
  3     RETURN VARCHAR2
  4  AS
  5  BEGIN
  6     IF check_in
  7      THEN
  8         RETURN 'abc';
  9      ELSE
 10         DBMS_OUTPUT.put_line (
 11         'Here I am, here I stay');
 12      END IF;
 13   END no_return;
 14   /

SP2-0806: Function created with compilation warnings

SQL> SHOW ERRORS
Errors for FUNCTION NO_RETURN:

LINE/COL ERROR
-------- -----------------------------------------------------------------
1/1      PLW-05005: function NO_RETURN returns without value at line 13
```

Oracle has detected a branch of logic that will not result in the execution of a RETURN, so it flags the program with a warning. The *plw5005.sql* file on the book's web site contains even more complex conditional logic, demonstrating that the warning is raised for less trivial code structures as well.

PLW-06002: unreachable code

Oracle will now perform static (compile-time) analysis of your program to determine if any lines of code in your program will never be reached during execution. This is extremely valuable feedback to receive, but you may find that the compiler warns you of this problem on lines that do not, at first glance, seem to be unreachable. In fact, Oracle notes in the description of the action to take for this error that you should "disable the warning if much code is made unreachable intentionally and the warning message is more annoying than helpful." I will come back to this issue at the end of the section.

You already saw an example of this compile-time warning in the "A quick example" section at the beginning of this section. Now consider the following code:

```
/* File on web: plw6002.sql */
CREATE OR REPLACE PROCEDURE plw6002
AS
   l_checking BOOLEAN := FALSE;
BEGIN
   IF l_checking
   THEN
      DBMS_OUTPUT.put_line ('Never here...');
   ELSE
      DBMS_OUTPUT.put_line ('Always here...');
      GOTO end_of_function;
   END IF;
   <<end_of_function>>
   NULL;
14* END plw6002;
```

Oracle shows the following compile-time warnings for this program:

```
LINE/COL ERROR
-------- -----------------------------
5/7      PLW-06002: Unreachable code
7/7      PLW-06002: Unreachable code
13/4     PLW-06002: Unreachable code
```

I see why line 7 is marked as unreachable: l_checking is set to FALSE, and so line 7 can never run. But why is line 5 marked "unreachable." It seems as though, in fact, that code would *always* be run! Furthermore, line 13 will always be run as well because the GOTO will direct the flow of execution to that line through the label. Yet it is tagged as unreachable.

The reason for this behavior is simple: the unreachable code warning is generated after optimization of the code. To determine unreachability, the compiler has to translate the source code into an internal representation so that it can perform the necessary analysis of the control flow.

The compiler does *not* give you false positives; when it says that line *N* is unreachable, it is telling you that the line truly will never be executed, accurately reflecting the optimized code.

There are currently scenarios of unreachable code that are *not* flagged by the compiler. Here is one example:

```
/* File on web: plw6002.sql */
CREATE OR REPLACE FUNCTION plw6002 RETURN VARCHAR2
AS
BEGIN
   RETURN NULL;
   DBMS_OUTPUT.put_line ('Never here...');
END plw6002;
/
```

Certainly, the call to DBMS_OUTPUT.PUT_LINE is unreachable, but the compiler does not currently detect that state. This scenario, and others like it, may be covered in future releases of the compiler.

PLW-07203: parameter 'string' may benefit from use of the NOCOPY compiler hint

As mentioned earlier in relation to PLW-05005, use of NOCOPY with complex, large IN OUT parameters can improve the performance of programs under certain conditions. This warning will flag programs whose IN OUT parameters might benefit from NOCOPY. Here is an example of such a program:

```
/* File on web: plw7203.sql */
CREATE OR REPLACE PACKAGE plw7203
IS
   TYPE collection_t IS TABLE OF VARCHAR2 (100);
   PROCEDURE proc (collection_in IN OUT collection_t);
END plw7203;
/
```

This is another one of those warnings that will be generated for lots of programs and may become a nuisance. The warning/recommendation is certainly valid, but for most programs the impact of this optimization will not be noticeable. Furthermore, you are unlikely to switch to NOCOPY without making other changes in your code to handle situations where the program terminates before completing, possibly leaving your data in an uncertain state.

PLW-07204: conversion away from column type may result in sub-optimal query plan

This warning will surface when you call a SQL statement from within PL/SQL and rely on implicit conversions within that statement. Here is an example:

```
/* File on web: plw7204.sql */
CREATE OR REPLACE FUNCTION plw7204
   RETURN PLS_INTEGER
AS
   l_count PLS_INTEGER;
BEGIN
   SELECT COUNT(*) INTO l_count
     FROM employee
    WHERE salary = '10000';
   RETURN l_count;
END plw7204;
/
```

The salary column is numeric, but I am comparing it to a string value. The optimizer may well disable the use of an index on salary because of this implicit conversion.

Related tightly to this warning is *PLW-7202: bind type would result in conversion away from column type.*

Conditional Compilation

Introduced in Oracle Database 10*g* Release 2, conditional compilation allows the compiler to compile selected parts of a program based on conditions you provide with the $IF directive.

Conditional compilation will come in very handy when you need to:

- Write a program that will run under different versions of Oracle, taking advantage of features specific to those versions. More specifically, you want to take advantage of new features of Oracle where available, but you also need that program to compile and run in older versions. Without conditional compilation, you would have to maintain multiple files or use complex SQL*Plus substitution variable logic.

- Run certain code during testing and debugging, but then omit that code from the production code. Prior to conditional compilation, you would need to either comment out lines of code or add some overhead to the processing of your application—even in production.
- Install/compile different elements of your application based on user requirements, such as the components for which a user is licensed. Conditional compilation greatly simplifies the maintenance of a code base with this complexity.

You implement conditional compilation by placing compiler *directives* (commands) in your source code. When your program is compiled, the PL/SQL preprocessor evaluates the directives and selects those portions of your code that should be compiled. This pared-down source code is then passed to the compiler for compilation.

There are three types of directives:

Selection directives
: Use the $IF directive to evaluate expressions and determine which code should be included or avoided.

Inquiry directives
: Use the $$*identifier* syntax to refer to conditional compilation flags. These inquiry directives can be referenced within an $IF directive or used independently in your code.

Error directives
: Use the $ERROR directive to report compilation errors based on conditions evaluated when the preprocessor prepares your code for compilation.

First we'll look at some simple examples, then delve more deeply into the capabilities of each directive. We'll also learn how to use two packages related to conditional compilation, DBMS_DB_VERSION and DBMS_PREPROCESSOR.

Examples of Conditional Compilation

Let's start with some examples of several types of conditional compilation.

Use application package constants in $IF directive

The $IF directive can reference constants defined in your own packages. In the example below, I vary the way that the bonus is applied depending on whether or not the location in which this third-party application is installed is complying with the Sarbanes-Oxley guidelines. Such a setting is unlikely to change for a long period of time. If I rely on the traditional conditional statement in this case, I will leave in place a branch of logic that should never be applied. With conditional compilation, the code is removed before compilation.

```
/* File on web: cc_my_package.sql */
CREATE OR REPLACE PROCEDURE apply_bonus (
   id_in IN employee.employee_id%TYPE
  ,bonus_in IN employee.bonus%TYPE)
IS
BEGIN
   UPDATE employee
      SET bonus =
       $IF employee_rp.apply_sarbanes_oxley
       $THEN
          LEAST (bonus_in, 10000)
       $ELSE
          bonus_in
       $END
     WHERE employee_id = id_in;
   NULL;
END apply_bonus;
/
```

Toggle tracing through conditional compilation flags

We can now set up our own debug/trace mechanisms and have them conditionally compiled into our code. This means that when our code rolls into production, we can have this code completely removed, so that there will be no runtime overhead to this logic. Note that I can specify both Boolean and PLS_INTEGER values through the special PLSQL_CCFLAGS compile parameter.

```
/* File on web: cc_debug_trace.sql */
ALTER SESSION SET PLSQL_CCFLAGS = 'oe_debug:true, oe_trace_level:10';

CREATE OR REPLACE PROCEDURE calculate_totals
IS
BEGIN
$IF $$oe_debug AND $$oe_trace_level >= 5
$THEN
   DBMS_OUTPUT.PUT_LINE ('Tracing at level 5 or higher');
$END
   NULL;
END calculate_totals;
/
```

The Inquiry Directive

An inquiry directive is a directive that makes an inquiry of the compilation environment. Of course, that doesn't really tell you much. So let's take a closer look at the syntax for inquiry directives and the different sources of information available through the inquiry directive.

The syntax for an inquiry directive is as follows:

```
$$identifier
```

where *identifier* is a valid PL/SQL identifier that can represent any of the following:

- Compilation environment settings: the values found in the USER_PLSQL_OBJECT_SETTINGS data dictionary view
- Your own custom-named directive, defined with the ALTER...SET PLSQL_CCFLAGS command, described in a later section
- Implicitly defined directives: $$PLSQL_LINE and $$PLSQL_UNIT, providing you with the line number and program name

Inquiry directives are designed for use within conditional compilation clauses, but they can also be used in other places in your PL/SQL code. For example, I can display the current line number in my program with this code:

```
DBMS_OUTPUT.PUT_LINE ($$PLSQL_LINE);
```

I can also use inquiry directives to define and apply application-wide constants in my code. Suppose, for example, that the maximum number of years of data supported in my application is 100. Rather than hardcode this value in my code, I could do the following:

```
ALTER SESSION SET PLSQL_CCFLAGS = 'max_years:100';

CREATE OR REPLACE PROCEDURE work_with_data (num_years_in IN PLS_INTEGER)
IS
BEGIN
   IF num_years_in > $$max_years THEN ...
END  work_with_data;
```

Even more valuable, I can use inquiry directives in places in my code where a variable is not allowed. Here are two examples:

```
DECLARE
   l_big_string VARCHAR2($$MAX_VARCHAR2_SIZE);

   l_default_app_err EXCEPTION;
   PRAGMA EXCEPTION_INIT (l_default_app_err, $$DEF_APP_ERR_CODE);
BEGIN
```

The DBMS_DB_VERSION package

The DBMS_DB_VERSION built-in package offers a set of constants that give you absolute and relative information about the version of your installed database. The constants defined in the Oracle Database 10g Release 2 version of this package are shown in Table 20-2.

Table 20-2. DBMS_DB _VERSION constants

Name of packaged constant	Significance	Value in Oracle Database 10g Release 2
DBMS_DB_VERSION.VERSION	The RDBMS version number, as in 10 for Oracle Database 10*g*	10

Table 20-2. DBMS_DB _VERSION constants (continued)

Name of packaged constant	Significance	Value in Oracle Database 10g Release 2
DBMS_DB_VERSION.RELEASE	The RDBMS release number, as in 2 for Oracle Database 10*g* Release 2	2
DBMS_DB_VERSION.VER_LE_9	TRUE if the current version is less than or equal to Oracle9*i* Database	FALSE
DBMS_DB_VERSION.VER_LE_9_1	TRUE if the current version is less than or equal to Oracle9*i* Database Release 1	FALSE
DBMS_DB_VERSION.VER_LE_9_2	TRUE if the current version is less than or equal to Oracle9*i* Database Release 2	FALSE
DBMS_DB_VERSION.VER_LE_10	TRUE if the current version is less than or equal to Oracle Database 10*g*	TRUE
DBMS_DB_VERSION.VER_LE_10_1	TRUE if the current version is less than or equal to Oracle Database 10*g* Release 1	FALSE
DBMS_DB_VERSION.VER_LE_10_2	TRUE if the current version is less than or equal to Oracle Database 10*g* Release 2	TRUE

While this package was designed for use with conditional compilation, you can, of course, use it for your own purposes.

With each new release of the database, Oracle will add additional constants and will update the values returned by the VERSION and RELEASE constants.

Interestingly, you can write expressions that include references to as-yet undefined constants in the DBMS_DB_VERSION package. As long as they are not evaluated, as in the case below, they will not cause any errors. Here is an example:

```
$IF DBMS_DB_VERSION.VER_LE_10_2
$THEN
   Use this code.
$ELSEIF DBMS_DB_VERSION.VER_LE_11
   This is a placeholder for future.
$ENDIF
```

Setting compilation environment parameters

The following information (corresponding to the values in the USER_PLSQL_OBJECT_SETTINGS data dictionary view) is available via inquiry directives:

$$PLSQL_DEBUG
: Debug setting for this compilation unit

$$PLSQL_OPTIMIZE_LEVEL
: Optimization level for this compilation unit

$$PLSQL_CODE_TYPE
: Compilation mode for the unit

$$PLSQL_WARNINGS
: Compilation warnings setting for this compilation unit

$$NLS_LENGTH_SEMANTICS
: Value set for the NLS length semantics

See the *cc_plsql_parameters.sql* file on the book's web site for a demonstration that uses each of these parameters.

Referencing unit name and line number

Oracle implicitly defines two very useful inquiry directives for use in $IF and $ERROR directives:

$$PLSQL_UNIT
: Name of the compilation unit in which the reference appears

$$PLSQL_LINE
: Line number of the compilation unit where the reference appears

You can call DBMS_UTILITY.FORMAT_CALL_STACK and DBMS_UTILITY. FORMAT_ERROR_BACKTRACE to obtain current line numbers, but then you must also parse those strings to find the line number and program unit name. These inquiry directives provide the information more directly. Here is an example:

```
BEGIN
   IF l_balance < 10000
   THEN
      raise_error (
         err_name => 'BALANCE TOO LOW'
        ,failed_in => $$plsql_unit
        ,failed_on => $$plsql_line
      );
   END IF;
   ...
END;
```

Run *cc_line_unit.sql* to see a demonstration of using these last two directives.

Note that when $$PLSQL_UNIT is referenced inside a package, it will return the name of the package, not the individual procedure or function within the package.

Using the PLSQL_CCFLAGS parameter

Oracle offers a new initialization parameter, PLSQL_CCFLAGS, that you can use with conditional compilation. Essentially, it allows you to define name-value pairs, and the name can then be referenced as an inquiry directive in your conditional compilation logic. Here is an example:

```
ALTER SESSION SET PLSQL_CCFLAGS = 'use_debug:TRUE, trace_level:10';
```

The flag name can be set to any valid PL/SQL identifier, including reserved words and keywords (the identifier will be prefixed with $$, so there will be no confusion with normal PL/SQL code). The value assigned to the name must be one of the following: TRUE, FALSE, NULL, or a PLS_INTEGER literal.

The PLSQL_CCFLAGS value will be associated with each program that is then compiled in that session. If you want to keep those settings with the program, then future compilations with the ALTER...COMPILE command should include the REUSE SETTINGS clause.

Because you can change the value of this parameter and then compile selected program units, you can easily define different sets of inquiry directives for different programs.

Note that you can refer to a flag that is *not* defined in PLSQL_CCFLAGS; this flag will evaluate to NULL. If you enable compile-time warnings, this reference to an undefined flag will cause Oracle to report a *PLW-06003: unknown inquiry directive* warning (unless the source code is wrapped).

The $IF Directive

Use the selection directive, implemented through the $IF statement, to direct the conditional compilation step in the preprocessor. Here is the general syntax of this directive:

```
$IF Boolean-expression
$THEN
   code-fragment
[ $ELSEIF Boolean-expression
$THEN
   code-fragment]
[ $ELSE
   code-fragment]
$END
```

where *Boolean-expression* is a static expression (it can be evaluated at the time of compilation) that evaluates to TRUE, FALSE, or NULL The *code-fragment* can be any set of PL/SQL statements, which will then be passed to the compiler for compilation, as directed by the expression evaluations.

Static expressions can be constructed from any of the following elements:

- Boolean, PLS_INTEGER, and NULL literals, plus combinations of these literals.
- Boolean, PLS_INTEGER, and VARCHAR2 static expressions.
- Inquiry directives: identifiers prefixed with $$. These directives can be provided by Oracle (e.g., $$PLSQL_OPTIMIZE_LEVEL; the full list is provided in the earlier section “The Optimizing Compiler”) or set via the PLSQL_CCFLAGS compilation parameter (also explained earlier).
- Static constants defined in a PL/SQL package.

- It can include most comparison operations (>, <, =, <> are fine, but you cannot use an IN expression), logical Boolean operations such as AND and OR, concatenations of static character expressions, and tests for NULL.

A static expression may not contain calls to procedures or functions that require execution; they cannot be evaluated during compilation and therefore will render invalid the expression within the $IF directive. You will get a compile error as follows:

```
PLS-00174: a static boolean expression must be used
```

Here are examples of static expressions in $IF directives:

- If the user-defined inquiry directive controlling debugging is not null, then initialize the debug subsystem:

  ```
  $IF $$app_debug_level IS NOT NULL $THEN
     debug_pkg.initialize;
  $END
  ```

- Check the value of a user-defined package constant along with the optimization level:

  ```
  $IF $$PLSQL_OPTIMIZE_LEVEL = 2 AND appdef_pkg.long_compilation
  $THEN
     $ERROR 'Do not use optimization level 2 for this program!'
  $END
  ```

String literals and concatenations of strings are allowed only in the $ERROR directive; they may not appear in the $IF directive.

The $ERROR Directive

Use the $ERROR directive to cause the current compilation to fail and return the error message provided. The syntax of this directive is:

```
$ERROR VARCHAR2-expression $END
```

Suppose that I need to set the optimization level for a particular program unit to 1, so that compilation time will be improved. In the following example, I use the $$ inquiry directive to check the value of the optimization level from the compilation environment. I then raise an error with the $ERROR directive as necessary.

```
/* File on web: cc_opt_level_check.sql */
SQL> CREATE OR REPLACE PROCEDURE long_compilation
     IS
     BEGIN
     $IF $$plsql_optimize_level != 1
     $THEN
        $error 'This program must be compiled with optimization level = 1' $end
     $END
        NULL;
     END long_compilation;
     /
```

```
Warning: Procedure created with compilation errors.

SQL> SHOW ERRORS
Errors for PROCEDURE LONG_COMPILATION:

LINE/COL ERROR
-------- -----------------------------------------------------------------
6/4      PLS-00179: $ERROR: This program must be compiled with
         optimization level = 1
```

Synchronizing Code with Packaged Constants

Use of packaged constants within a selection directive allows you to easily synchronize multiple program units around a specific conditional compilation setting. This is possible because Oracle's automatic dependency management is applied to selection directives. In other words, if program unit PROG contains a selection directive that references package PKG, then PROG is marked as dependent on PKG. When the specification of PKG is recompiled, all program units using the packaged constant are marked invalid and must be recompiled.

Suppose I want to use conditional compilation to automatically include or exclude debugging and tracing logic in my code base. I define a package specification to hold the required constants:

```
/* File on web: cc_debug.pks */
CREATE OR REPLACE PACKAGE cc_debug
IS
   debug_active CONSTANT BOOLEAN := TRUE;
   trace_level CONSTANT PLS_INTEGER := 10;
END cc_debug;
/
```

I then use these constants in procedure calc_totals:

```
CREATE OR REPLACE PROCEDURE calc_totals
IS
BEGIN
$IF cc_debug.debug_active AND cc_debug.trace_level > 5 $THEN
   log_info (...);
$END
   ...
END calc_totals;
/
```

During development, the debug_active constant is initialized to TRUE. When it is time to move the code to production, I change the flag to FALSE and recompile the package. The calc_totals program and all other programs with similar selection directives are marked invalid and must then be recompiled.

Program-Specific Settings with Inquiry Directives

Packaged constants are useful for coordinating settings across multiple program units. Inquiry directives, drawn from the compilation settings of individual programs, are a better fit when you need different settings applied to different programs.

Once you have compiled a program with a particular set of values, it will retain those values until the next compilation (either from a file or a simple recompilation using the ALTER…COMPILE statement). Furthermore, a program is guaranteed to be recompiled with the same postprocessed source as was selected at the time of the *previous* compilation if all of the following conditions are TRUE:

- None of the conditional compilation directives refer to package constants. Instead, they rely only on inquiry directives.
- When the program is recompiled, the REUSE SETTINGS clause is used *and* the PLSQL_CCFLAGS parameter isn't included in the ALTER...COMPILE command.

This capability is demonstrated by the *cc_reuse_settings.sql* script, whose output is shown below. I first set the value of app_debug to TRUE and then compile a program with that setting, A query against USER_PLSQL_OBJECT_SETTINGS shows that this value is now associated with the program unit:

```
/* File on web: cc_reuse_settings.sql */

SQL> ALTER SESSION SET plsql_ccflags = 'app_debug:TRUE';

SQL> CREATE OR REPLACE PROCEDURE test_ccflags
  2  IS
  3  BEGIN
  4     NULL;
  5  END test_ccflags;
  6  /

SQL> SELECT name, plsql_ccflags
  2    FROM user_plsql_object_settings
  3   WHERE NAME LIKE '%CCFLAGS%';

NAME                           PLSQL_CCFLAGS
------------------------------ ----------------------------
TEST_CCFLAGS                   app_debug:TRUE
```

I now alter the session, setting $$app_debug to evaluate to FALSE. I compile a new program with this setting:

```
SQL> ALTER SESSION SET plsql_ccflags = 'app_debug:FALSE';

SQL> CREATE OR REPLACE PROCEDURE test_ccflags_new
  2  IS
  3  BEGIN
  4     NULL;
  5  END test_ccflags_new;
  6  /
```

Then I recompile my existing program with REUSE SETTINGS:

```
SQL> ALTER  PROCEDURE test_ccflags COMPILE REUSE SETTINGS;
```

A query against the data dictionary view now reveals that my settings are different for each program:

```
SQL> SELECT name, plsql_ccflags
  2    FROM user_plsql_object_settings
  3   WHERE NAME LIKE '%CCFLAGS%';

NAME                          PLSQL_CCFLAGS
----------------------------- ----------------------------
TEST_CCFLAGS                  app_debug:TRUE
TEST_CCFLAGS_NEW              app_debug:FALSE
```

Working with Postprocessed Code

You can use the DBMS_PREPROCESSOR package to display or retrieve the source text of your program in its postprocessed form. DBMS_PREPROCESSOR offers two programs, overloaded to allow you to specify the object of interest in various ways, as well as to work with individual strings and collections:

DBMS_PREPROCESSOR.PRINT_POST_PROCESSED_SOURCE
: Retrieves the postprocessed source and then displays it with the function DBMS_OUTPUT.PUTLINE.

DBMS_PREPROCESSOR.GET_POST_PROCESSED_SOURCE
: Returns the postprocessed source as either a single string or a collection of strings.

When working with the collection version of either of these programs, you will need to declare that collection based on the following package-defined collection:

```
TYPE DBMS_PREPROCESSOR.source_lines_t IS TABLE OF VARCHAR2(32767)
   INDEX BY BINARY_INTEGER;
```

The following sequence demonstrates the capability of these programs. I compile a very small program with a selection directive based on the optimization level. I then display the postprocessed code, and it shows the correct branch of the $IF statement.

```
/* File on web: cc_postprocessor.sql
CREATE OR REPLACE PROCEDURE post_processed
IS
BEGIN
$IF $$PLSQL_OPTIMIZE_LEVEL = 1
$THEN
   -- Slow and easy
  NULL;
$ELSE
   -- Fast and modern and easy
   NULL;
$END
```

```
END post_processed;
/

SQL> BEGIN
  2     DBMS_PREPROCESSOR.PRINT_POST_PROCESSED_SOURCE (
  3        'PROCEDURE', USER, 'POST_PROCESSED');
  4  END;
  5  /

PROCEDURE post_processed
IS
BEGIN
-- Fast and modern and easy
NULL;
END post_processed;
```

In the following block, I use the "get" function to retrieve the postprocessed code, and then display it using DBMS_OUTPUT.PUT_LINE:

```
DECLARE
   l_postproc_code   DBMS_PREPROCESSOR.SOURCE_LINES_T;
   l_row             PLS_INTEGER;
BEGIN
   l_postproc_code :=
      DBMS_PREPROCESSOR.GET_POST_PROCESSED_SOURCE (
         'PROCEDURE', USER, 'POST_PROCESSED');
   l_row := l_postproc_code.FIRST;

   WHILE (l_row IS NOT NULL)
   LOOP
      DBMS_OUTPUT.put_line (  LPAD (l_row, 3)
                             || ' - '
                             || rtrim ( l_postproc_code (l_row),chr(10))
                            );
      l_row := l_postproc_code.NEXT (l_row);
   END LOOP;
END;
/
```

Conditional compilation opens up all sorts of possibilities for PL/SQL developers and application administrators. And its usefulness only increases as new versions of Oracle are released and the DBMS_DB_VERSION constants can be put to full use, allowing us to take full advantage of each version's unique PL/SQL features.

Testing PL/SQL Programs

I get great satisfaction out of creating new things, and that is one of the reasons I so enjoy writing software. I love to take an interesting idea or challenge, and then come up with a way of using the PL/SQL language to meet that challenge.

I have to admit, though, that I don't really like having to take the time to test my software (nor do I like to write documentation for it). I do it, but I don't really do enough of it. And I have this funny feeling that I am not alone. The overwhelming reality is that developers generally perform an inadequate number of inadequate tests and figure that if the users don't find a bug, there is no bug. Why does this happen? Let me count the ways…

The psychology of success and failure
: We are so focused on getting our code to work correctly that we generally shy away from bad news—or from taking the chance of getting bad news. Better to do some cursory testing, confirm that everything seems to be working OK, and then wait for others to find bugs, if there are any (as if there were any doubt).

Deadline pressures
: Hey, it's Internet time! Time to market determines all. We need everything yesterday, so let's release pre-beta software as production and let our users test/suffer through our applications.

Management's lack of understanding
: IT management is notorious for not really understanding the software development process. If we aren't given the time and authority to write (and I mean "write" in the broadest sense, including testing, documentation, refinement, etc.) code properly, we will always end up with buggy junk that no one wants to admit ownership of.

Overhead of setting up and running tests
: If it's a big deal to write and run tests, they won't get done. We'll decide that we don't have time; after all, there is always something else to work on. One consequence of this is that more and more of the testing is handed over to the QA department, if there is one. That transfer of responsibility is, on the one hand, positive. Professional quality assurance professionals can have a tremendous impact on application quality. Yet developers must take and exercise responsibility for unit testing their own code; otherwise, the testing/QA process is much more frustrating and extended.

The bottom line is that our code almost universally needs more testing. I recently spent a fair amount of time thinking about how to improve my testing procedures. I studied test frameworks developed by other programmers who work primarily with object-oriented languages. An obsessive coder, I then proceeded to construct my own framework for unit testing PL/SQL programs, which I named utPLSQL, an open source project that is being used by developers around the world. It is complemented by Ounit, a graphical interface to utPLSQL. Let's take a look at how these tools can help.

Typical, Tawdry Testing Techniques

Say that I am writing a big application with lots of string manipulation. I've got a "hangnail" called SUBSTR; this function bothers me, and I need to take care of it.

What's the problem? SUBSTR is great when you know the starting location of a string and the number of characters you want. In many situations, though, I have only the start and end locations, and then I have to compute the number of characters. But which formula is it?

```
end - start
end - start +1
end - start - 1
```

I can never remember, so I write a program that will remember it for me—the betwnstr function:

```
CREATE OR REPLACE FUNCTION betwnStr (
   string_in IN VARCHAR2,
   start_in IN INTEGER,
   end_in IN INTEGER
   )
   RETURN VARCHAR2
IS
BEGIN
   RETURN (
      SUBSTR (
         string_in,
         start_in,
         end_in - start_in + 1
         )
      );
END;
```

The best way to test this program is to come up with a list of all the different test cases; here is a subset of the total, just to give you the idea:

String	Start	End	Result
abcdefg	1	3	abc
abcdefg	3	6	cdef
N/A	NULL	NOT NULL	NULL
N/A	NOT NULL	NULL	NULL
NULL	N/A	N/A	NULL
abcdefg	Positive number	Smaller than start	NULL
abcdefg	1	Number larger than length of string	abcdefg

From this grid, I can construct a simple test script like the following:

```
SET SERVEROUTPUT ON FORMAT WRAPPED
BEGIN
   DBMS_OUTPUT.PUT_LINE (betwnstr ('abcdefg', 1, 3));
   DBMS_OUTPUT.PUT_LINE (betwnstr ('abcdefg', 3, 6));
   DBMS_OUTPUT.PUT_LINE (betwnstr ('abcdefg', NULL, 2));
   DBMS_OUTPUT.PUT_LINE (betwnstr ('abcdefg', 3, NULL));
   DBMS_OUTPUT.PUT_LINE (betwnstr ('abcdefg', 5, 1));
```

```
      DBMS_OUTPUT.PUT_LINE (betwnstr ('abcdefg', 1, 100));
   END;
```

And when I run this code in SQL*Plus, I see the following results:

```
SQL> @betwnstr.tst
abc
cdef




abcdefg
```

And then I review the results and decide if the outcome matches my expectations. Of course, I have to be able to figure out just how many blank lines there were between "cdef" and "abcdefg". Plus, if I am going to test this code thoroughly, I will probably have upwards of 30 test cases (what about *negative* start and end values?). It will take me at least several minutes to scan the results of my test. And this is a ridiculously simple piece of code. The thought of extending this technique to my "real" code is frightening.

If we are going to test effectively and thoroughly, we will need to take a different path. We need a way to define our tests so that they can easily be maintained over time. We need to be able to easily run our tests and then, most importantly, determine without lengthy analysis the outcome: success or failure.

Let's take a look at how I would tackle the testing of betwnstr with a unit-testing framework such as utPLSQL.

utPLSQL: A Unit-Testing Framework

I don't have room in this book to provide a complete explanation of how utPLSQL works. So I will instead try to impress you with how much it can do for you. Then you will be so excited that you will rush to the web site and take utPLSQL out for a drive all on your own.

Using utPLSQL with betwnstr

In the previous section, I started creating a grid of inputs and expected results for calls to betwnstr. I will now transform that grid into a delimited string that looks like this:

```
DECLARE
     test_grid   VARCHAR2 (1000) := '
betwnstr|1|start at 1|start at 1|abcdefgh;1;3|abc|eq|N
betwnstr|1|start at 3|start at 3|abcdefgh;3;6|cde|eq|N
betwnstr|1|null start|null start|abcdefgh;!null;2|null|isnull|Y
betwnstr|1|null end||abcdefgh;!3;!null|null|isnull|Y
betwnstr|1|null string||!null;1;2|NULL|isnull|Y
betwnstr|1|big start small end||abcdefgh;10;5|null|isnull|Y
betwnstr|1|end past string||abcdefgh;1;100|abcdefgh|eq|N';
```

I will then pass that string to a program in the utGen package, which will generate all of my test code for me:

```
BEGIN
   utgen.testpkg_from_string ('betwnstr',
      test_grid,
      output_type_in=> utgen.c_file,
      dir_in=> 'TEMP'
   );
END;
```

I then compile the *ut_bewtnstr.pks* and *ut_betwnstr.pkb* files that were generated:

```
SQL> @ut_betwnstr.pks
SQL> @ut_betwnstr.pkb
```

I am now ready to run my test, so I open a SQL*Plus session and issue this statement:

```
SQL> exec utplsql.test ('betwnstr')
```

I am then presented with this information:

```
>  FFFFFFF   AA     III  L      U     U RRRRR   EEEEEEE
>  F        A  A     I   L      U     U R    R  E
>  F       A    A    I   L      U     U R     R E
>  F      A      A   I   L      U     U R     R E
>  FFFF   A      A   I   L      U     U RRRRRR  EEEE
>  F      AAAAAAAA   I   L      U     U R   R   E
>  F      A      A   I   L      U     U R    R  E
>  F      A      A   I   L       U   U  R     R E
>  F      A      A  III  LLLLLLL  UUU   R     R EEEEEEE
.
 FAILURE: "betwnstr"
.
> Individual Test Case Results:
>
SUCCESS - EQ "start at 1" Expected "abc" and got "abc"
FAILURE - EQ "start at 3" Expected "cde" and got "cdef"
SUCCESS - ISNULL "null start" Expected "" and got ""
SUCCESS - ISNULL "null end" Expected "" and got ""
SUCCESS - ISNULL "null string" Expected "" and got ""
SUCCESS - ISNULL "big start small end" Expected "" and got ""
SUCCESS - EQ "end past string" Expected "abcdefgh" and got "abcdefgh"
```

Notice that utPLSQL shows me which of the test cases failed, what it expected, and what it received after running the test. So the first thing I do is go back to my test code (which in this case is simply a grid of test case inputs and outputs) and make sure I didn't make any mistakes. I focus on this line:

```
betwnstr|1|start at 3|start at 3|abcdefgh;3;6|cde|eq|N
```

It doesn't take me long to realize that the "cde" or expected results is wrong. It should be "cdef." So I change my test case information, regenerate my test code, run my test, and then am delighted to see this on my screen:

```
SQL> exec utplsql.test ('betwnstr')
.
>    SSSS   U     U   CCC     CCC   EEEEEEE   SSSS     SSSS
>   S    S  U     U  C   C   C   C  E        S    S   S    S
>  S        U     U C     C C     C E       S        S
>   S       U     U C       C       E        S        S
>    SSSS   U     U C       C       EEEE      SSSS     SSSS
>        S  U     U C       C       E             S        S
>         S U     U C     C C     C E              S        S
>   S    S   U   U   C   C   C   C  E        S    S   S    S
>    SSSS     UUU     CCC     CCC   EEEEEEE   SSSS     SSSS
.
 SUCCESS: "betwnstr"
```

This is a very brief introduction to utPLSQL, but you can see that this framework automatically runs my test, and then tells me whether or not my test succeeded. It even reports on individual test cases.

utPLSQL was able to generate 100% of my test package for betwnstr, which is a bit of a special case in that it is a deterministic function (see Chapter 17 for more details on this characteristic). For most of the code you have written, you will be able to generate a *starting point* for your test package, but then complete it (and maintain it) manually.

utPLSQL doesn't take all the pain out of building and running test code, but it provides a standardized process and a test harness from which you can run your tests and easily view results.

Where to find utPLSQL and Ounit

While there is a lot more to be said and demonstrated about utPLSQL, you should now have enough of an understanding of it to decide whether it might be of interest to you. To learn more about utPLSQL, the utAssert assertion routines, and the rest of this unit-testing framework, visit the project home for utPLSQL at:

https://sourceforge.net/projects/utplsql/

You can also download utPLSQL along with a graphical interface to the test framework, named Ounit, at:

http: www.ounit.com

Both products are free.

Onxo also offers a graphical interface to utPLSQL and adds test package generation capabilities as well. Check it out at:

http://www.qnxo.com

Debugging PL/SQL Programs

When you test a program, you find errors in your code. When you debug a program, you uncover the cause of an error and fix it. These are two very different processes and should not be confused. Once a program is tested, and bugs are uncovered, it is certainly the responsibility of the developer to fix those bugs. And so the debugging begins!

Many programmers find that debugging is by far the hardest part of programming. This difficulty often arises from the following factors:

Lack of understanding of the problem being solved by the program
: Most programmers like to code. They tend to not like reading and understanding specifications, and will sometimes forgo this step so that they can quickly get down to writing code. The chance of a program meeting its requirements under these conditions is slim at best.

Poor programming practice
: Programs that are hard to read (lack of documentation, too much documentation, inconsistent use of whitespace, bad choices for identifier names, etc.), programs that are not properly modularized, and programs that try to be too clever present a much greater challenge to debug than programs that are well designed and structured.

The program simply contains too many errors
: Without the proper analysis and coding skills, your code will have a much higher occurrence of bugs. When you compile a program and get back five screens of compile errors, do you just want to scream and hide? It is easy to be so overwhelmed by your errors that you don't take the organized, step-by-step approach needed to fix those errors.

Limited debugging skills
: There are many different approaches to uncovering the causes of your problems. Some approaches only make life more difficult for you. If you have not been trained in the best way to debug your code, you can waste many hours, raise your blood pressure, and upset your manager.

The following sections review the debugging methods that you will want to avoid at all costs, and then offer recommendations for more effective debugging strategies.

The Wrong Way to Debug

As I present the various ways you shouldn't debug your programs, I expect that just about all of you will say to yourselves, "Well, that sure is obvious. Of course you shouldn't do that. I never do that."

And yet the very next time you sit down to do your work, you may very well follow some of these obviously horrible debugging practices.

If you happen to see little bits of yourself in the paragraphs that follow, I hope you will be inspired to mend your ways.

Disorganized debugging

When faced with a bug, you become a whirlwind of frenzied activity. Even though the presence of an error indicates that you did not fully analyze the problem and figure out how the program should solve it, you do not now take the time to understand the program. Instead you place MESSAGE statements (in Oracle Forms) or SRW.MESSAGE statements (in Oracle Reports) or DBMS_OUTPUT.PUT_LINE statements (in stored modules) all over your program in the hopes of extracting more clues.

You do not save a copy of the program before you start making changes because that would take too much time; you are under a lot of pressure right now, and you are certain that the answer will pop right out at you. You will just remove your debug statements later.

You spend lots of time looking at information that is mostly irrelevant. You question everything about your program, even though most of it uses constructs you've employed successfully for years.

You skip lunch but make time for coffee, lots of coffee, because it is free and you want to make sure your concentration is at the most intense level possible. Even though you have no idea what is causing the problem, you think that maybe if you try this one change, it might help. You make the change and take several minutes to compile, generate, and run through the test case, only to find that the change didn't help. In fact, it seemed to cause another problem because you hadn't thought through the impact of the change on your application.

So you back out of that change and try something else in hopes that it might work. But several minutes later, you again find that it doesn't. A friend, noticing that your fingers are trembling, offers to help. But you don't know where to start explaining the problem because you don't really know what is wrong. Furthermore, you are kind of embarrassed about what you've done so far (turned the program into a minefield of tracing statements) and realize you don't have a clean version to show your friend. So you snap at the best programmer in your group and call your family to let them know you aren't going to be home for dinner that night.

Why? Because you are determined to fix that bug!

Irrational debugging

You execute your report, and it comes up empty. You spent the last hour making changes both in the underlying data structures and in the code that queries and formats the data. You are certain, however, that your modifications could not have made the report disappear.

You call your internal support hotline to find out if there is a network problem, even though File Manager clearly shows access to network drives. You further probe as to whether the database has gone down, even though you just connected successfully. You spend another 10 minutes of the support analyst's time running through a variety of scenarios before you hang up in frustration.

"They don't know anything over there," you fume. You realize that you will have to figure this one out all by yourself. So you dive into the code you just modified. You are determined to check every single line until you find the cause of your difficulty. Over the course of the next two hours, you talk aloud to yourself—a lot.

"Look at that! I called the stored procedure inside an IF statement. I never did that before. Maybe you can't call stored programs that way." So you remove the IF statement and instead use a GOTO statement to perform the branching to the stored procedure. But that doesn't fix the problem.

"My code seems fine. But it calls this other routine that Joe wrote ages ago." Joe has since moved on, making him a ripe candidate for the scapegoat. "It probably doesn't work anymore; after all, we did upgrade to a new voicemail system." So you decide to perform a standalone test of Joe's routine, which hasn't changed for two years and has no interface to voicemail. But his program seems to work fine—when it's not run from your program.

Now you are starting to get desperate. "Maybe this report should only run on weekends. Hey, can I put a local module in an anonymous block? Maybe I can use only local modules in procedures and functions! I think maybe I heard about a bug in this tool. Time for a workaround…"

You get angry and begin to understand why your eight-year-old hits the computer monitor when he can't beat the last level of Ultra Mystic Conqueror VII. And just as you are ready to go home and take it out on your dog, you realize that you are connected to the development database, which has almost no data at all. You switch to the test instance, run your report, and everything looks just fine.

Except, of course, for that GOTO and all the other workarounds you stuck in the report…

Debugging Tips and Strategies

In this chapter, I do not pretend to offer a comprehensive primer on debugging. The following tips and techniques, however, should improve on your current set of error-fixing skills.

Use a source code debugger

The single most effective thing you can do to minimize the time spent debugging your code is to use a source code debugger. One is now available in just about every

PL/SQL Integrated Development Environment (IDE). If you are using Quest's Toad or SQL Navigator, Allround Automations' PL/SQL Developer, or Oracle JDeveloper (or any other such GUI tool), you will be able to set visual breakpoints in your code with the click of a mouse, step through your code line by line, watch variables as they change their values, and so on.

The other tips in this section apply whether or not you are using a GUI-based debugger, but there is no doubt that if you are still debugging the old-fashioned way (inserting calls to DBMS_OUTPUT.PUT_LINE in dozens of places in your code), you are wasting a lot of your time. (Unfortunately, if your code is deployed at some customer site, debugging with a GUI tool is not always possible, in which case you usually have to resort to some sort of logging mechanism.)

Gather data

Gather as much data as possible about when, where, and how the error occurred. It is very unlikely that the first occurrence of an error will give you all the information you will want or need to figure out the source of that error. Upon noticing an error, the temptation is to show off one's knowledge of the program by declaring, "Got it! I know what's going on and exactly how to fix it." This can be very gratifying when it turns out that you do have a handle on the problem, and that may be the case for simple bugs. Some problems can appear simple, however, and turn out to require extensive testing and analysis. Save yourself the embarrassment of pretending (or believing) that you know more than you actually do. Before rushing to change your code, take these steps:

Run the program again to see if the error is reproducible
: This will be the first indication of the complexity of the problem. It is almost impossible to determine the cause of a problem if you are unable to get it to occur predictably. Once you work out the steps needed to get the error to occur, you will have gained much valuable information about its cause.

Narrow the test case needed to generate the error
: I recently had to debug a problem in one of my Oracle Forms modules. A pop-up window would lose its data under certain circumstances. At first glance, the rule seemed to be: "For a new call, if you enter only one request, that request will be lost." If I had stopped testing at that point, I would have had to analyze all code that initialized the call record and handled the INSERT logic. Instead, I tried additional variations of data entry and soon found that the data was lost only when I navigated to the pop-up window directly from a certain item. Now I had a very narrow test case to analyze, and it became very easy to uncover the error in logic.

Examine the circumstances under which the problem does not occur

"Failure to fail" can offer many insights into the reason an error does occur. It also helps you narrow down the sections of code and the conditions you have to analyze when you go back to the program.

The more information you gather about the problem at hand, the easier it will be to solve that problem. It is worth the extra time to assemble the evidence. So even when you are absolutely sure you are on to that bug, hold off and investigate a little further.

Remain logical at all times

Symbolic logic is the lifeblood of programmers. No matter which programming language you use, the underlying logical framework is a constant. PL/SQL has one particular syntax. The C language uses different keywords, and the IF statement looks a little different. The elegance of LISP demands a very different way of building programs. But underneath it all, symbolic logic provides the backbone on which you hang the statements that solve your problems.

The reliance on logical and rational thought in programming is one reason that it is so easy for a developer to learn a new programming language. As long as you can take the statement of a problem and develop a logical solution step by step, the particulars of a language are secondary.

With logic at the core of our being, it amazes me to see how often we programmers abandon this logic and pursue the most irrational path to solving a problem. We engage in wishful thinking and highly superstitious, irrational, or dubious thought processes. Even though we know better—much better—we find ourselves questioning code that conforms to documented functionality, that has worked in the past, and that surely works at that moment. This irrationality almost always involves shifting the blame from oneself to the "other"—the computer, the compiler, Joe, the word processor, whatever. Anything and anybody but our own pristine selves!

When you attempt to shift blame, you only put off solving your problem. Computers and compilers may not be intelligent, but they're very fast and very consistent. All they can do is follow rules, and you write the rules in your program. So when you uncover a bug in your code, take responsibility for that error. Assume that *you* did something wrong—don't blame the PL/SQL compiler, Oracle Forms, or the text editor.

If you do find yourself questioning a basic element or rule in the compiler that has always worked for you in the past, it is time to take a break. Better yet, it is time to get someone else to look at your code. It is amazing how another pair of eyes can focus your own analytical powers on the real causes of a problem.

Strive to be the Spock of Programming. Accept only what is logical. Reject that which has no explanation.

Analyze instead of trying

So you have a pile of data and all the clues you could ask for in profiling the symptoms of your problem. Now it is time to analyze that data. For many people, analysis takes the following form: "Hmm, this looks like it could be the answer. I'll make this change, recompile, and try it to see if it works."

What's wrong with this approach? When you try a solution to see what will happen, what you are really saying is:

- You are not sure that the change really is a solution. If you were sure, you wouldn't "try" it to see what would happen. You would make the change and then test that change.
- You have not fully analyzed the error to understand its causes. If you know why an error occurs, then you know if a particular change will fix that problem. If you are unsure about the source of the error, you will be tempted to simply try a change and examine the impact. This is, unfortunately, very faulty logic.
- Even if the change stops the error from occurring, you can't be sure that your "solution" really solved anything. Because you aren't sure why the problem occurred, the simple fact that the problem doesn't reappear in your particular tests doesn't mean that you fixed the bug. The most you can say is that your change stopped the bug from occurring under certain, perhaps even most, circumstances.

To truly solve a problem, you must completely analyze the cause of the problem. Once you understand why the problem occurs, you have found the root cause and can take the steps necessary to make the problem go away in all circumstances.

When you identify a potential solution, perform a walk-through of your code based on that change. Don't execute your form. Examine your program, and mentally try out different scenarios to test your hypothesis. Once you are certain that your change actually does address the problem, you can then perform a test of that solution. You won't be *trying* anything; you will be *verifying* a fix.

Analyze your bug fully before you test solutions. If you say to yourself, "Why don't I try this?" in the hope that it will solve the problem, then you are wasting your time and debugging inefficiently.

Take breaks, and ask for help

We are often our own biggest obstacles when it comes to sorting out our problems, whether a program bug or a personal crisis. When you are stuck on the inside of a problem, it is hard to maintain an objective distance and take a fresh look.

When you are making absolutely no progress and feel that you have tried everything, try these two radical techniques:

- Take a break
- Ask for help

When I have struggled with a bug for any length of time without success, I not only become ineffective, I also tend to lose perspective. I pursue irrational and superstitious leads. I lose track of what I have already tested and what I have assumed to be right. I get too close to the problem to debug it effectively.

My frustration level usually correlates closely to the amount of time I have sat in my ergonomic chair and perched over my wrist-padded keyboard and stared at my low-radiation screen. Often the very simple act of stepping away from the workstation will clear my head and leave room for a solution to pop into place. Did you ever wake up the morning after a very difficult day at work to find the elusive answer sitting there at the end of your dream?

Make it a rule to get up and walk around at least once an hour when you are working on a problem—heck, even when you are writing your programs. Give your brain a chance to let its neural networks make the connections and develop new options for your programming. There is a whole big world out there. Even when your eyes are glued to the monitor and your source code, the world keeps turning. It never hurts to remind yourself of the bigger picture, even if that only amounts to taking note of the weather outside your air-conditioned cocoon.

Even more effective than taking a break is asking another person to look at your problem. There is something entirely magical about the dynamic of adding another pair of eyes to the situation. You might struggle with a problem for an hour or two, and finally, at the exact moment that you break down and explain the problem to a coworker, the solution will jump out at you. It could be a mismatch on names, a false assumption, or a misunderstanding of the IF statement logic. Whatever the case, chances are that you yourself will find it (even though you couldn't for the last two hours) as soon as you ask someone else to find it for you.

And even if the error does not yield itself quite so easily, you still have lots to gain from the perspective of another person who (a) did not write the code and has no subconscious assumptions or biases about it, and (b) isn't mad at the program.

Other benefits accrue from asking for help. You improve the self-esteem and self-confidence of other programmers by showing that you respect their opinions. If you are one of the best developers in the group, then your request for help demonstrates that you, too, sometimes make mistakes and need help from the team. This builds the sense (and the reality) of teamwork, which will improve the overall development and testing efforts on the project.

Change and test one area of code at a time

One of my biggest problems when I debug my code is that I am overconfident about my development and debugging skills, so I try to address too many problems at once. I make five or ten changes, rerun my test, and get very unreliable and minimally useful results. I find that my changes cause other problems (a common phenomenon until a program stabilizes, and a sure sign that lots more debugging and testing is needed), that some, but not all, of the original errors are gone, and that I have no idea which changes fixed which errors and which changes caused new errors.

In short, my debugging effort is a mess, and I have to back out of changes until I have a clearer picture of what is happening in my program.

Unless you are making very simple changes, you should fix one problem at a time and then test that fix. The amount of time it takes to compile, generate, and test may increase, but in the long run you will be much more productive.

Another aspect of incremental testing and debugging is performing unit tests on individual modules before you test a program that calls these various modules. If you test the programs separately and determine that they work, when you debug your application as a whole (in a system test), you do not have to worry about whether those modules return correct values or perform the correct actions. Instead, you can concentrate on the code that calls the modules. (See the earlier section "Testing PL/SQL Programs," for more on unit testing.)

You will also find it helpful to come up with a system for keeping track of your troubleshooting efforts. Dan Clamage, a reviewer for this book, reports that he maintains a simple text file with running commentary of his efforts to reproduce the problem and what he has done to correct it. This file will usually include any SQL written to analyze the situation, setup data for test cases, a list of the modules examined, and any other items that may be of interest in the future. With this file in place, it's much easier to return at any time (e.g., after you have had a good night's sleep and are ready to try again) and follow your original line of reasoning.

Tracing Execution of Your Code

Earlier versions of Oracle offered some PL/SQL trace capabilities, but Oracle8*i* Database introduced an API that allows you to easily specify and control the tracing of the execution of PL/SQL procedures, functions, and exceptions. The DBMS_TRACE built-in package provides programs to start and stop PL/SQL tracing in a session. When tracing is turned on, the engine collects data as the program executes. The data is then written out to the Oracle Server trace file.

In addition to DBMS_TRACE, you can take advantage of the built-in function, DBMS_UTILITY.FORMAT_CALL_STACK, to obtain the execution call stack at any point within your application.

The PL/SQL trace facility provides a trace file that shows you the specific steps executed by your code. The DBMS_PROFILER package (described later in this chapter) offers a much more comprehensive analysis of your application, including timing information and counts of the number of times a specific line was executed.

DBMS_UTILITY.FORMAT_CALL_STACK

This function returns the execution call stack (the sequence of program calls) down to the point at which you call the function. Here is an example of the formatting of this stack string:

```
----- PL/SQL Call Stack -----
object   line    object
handle   number  name

88ce3f74    8  package STEVEN.VALIDATE_REQUEST
88e49fc4    2  function STEVEN.COMPANY_TYPE
88e49390    1  procedure STEVEN.CALC_NET_WORTH
88e2bd20    1  anonymous block
```

One of the best places to use this function is within an exception handler, as in:

```
EXCEPTION
   WHEN OTHERS
   THEN
      DBMS_OUTPUT.PUT_LINE (
         DBMS_UTILITY.FORMAT_CALL_STACK);
END;
```

Better yet, grab this information and write it to your log table, so that the support and debug teams can immediately see how you got to the point where the problem reared its ugly head.

There is, by the way, one big problem with the exception section above: if your call stack is deep, the formatted string will exceed 255 characters in length. Before Oracle Database 10g Release 2, DBMS_OUTPUT.PUT_LINE would raise an exception in such cases. To avoid this problem, you might consider using Darko Egersdorfer's callstack package, found in the *callstack.pkg* file on the book's web site.

Installing DBMS_TRACE

This package may not have been installed automatically with the rest of the built-in packages. To determine whether DBMS_TRACE is present, connect to SYS (or another account with SYSDBA privileges) and execute this command:

```
BEGIN DBMS_TRACE.CLEAR_PLSQL_TRACE; END;
```

If you see this error:

```
PLS-00201: identifier 'DBMS_TRACE.CLEAR_PLSQL_TRACE' must be declared
```

then you must install the package. To do this, remain connected as SYS (or another account with SYSDBA privileges), and run the following files in the order specified:

$ORACLE_HOME/rdbms/admin/dbmspbt.sql
$ORACLE_HOME/rdbms/admin/prvtpbt.plb

DBMS_TRACE programs

The following programs are available in the DBMS_TRACE package:

SET_PLSQL_TRACE
: Starts PL/SQL tracing in the current session

CLEAR_PLSQL_TRACE
: Stops the dumping of trace data for that session

PLSQL_TRACE_VERSION
: Gets the major and minor version numbers of the DBMS_TRACE package

To trace execution of your PL/SQL code, you must first start the trace with a call to:

```
DBMS_TRACE.SET_PLSQL_TRACE (trace_level INTEGER);
```

in your current session, where *trace_level* is one of the following values:

- Constants that determine which elements of your PL/SQL program will be traced:

  ```
  DBMS_TRACE.trace_all_calls          constant INTEGER := 1;
  DBMS_TRACE.trace_enabled_calls      constant INTEGER := 2;
  DBMS_TRACE.trace_all_exceptions     constant INTEGER := 4;
  DBMS_TRACE.trace_enabled_exceptions constant INTEGER := 8;
  DBMS_TRACE.trace_all_sql            constant INTEGER := 32;
  DBMS_TRACE.trace_enabled_sql        constant INTEGER := 64;
  DBMS_TRACE.trace_all_lines          constant INTEGER := 128;
  DBMS_TRACE.trace_enabled_lines      constant INTEGER := 256;
  ```

- Constants that control the tracing process:

  ```
  DBMS_TRACE.trace_stop               constant INTEGER := 16384;
  DBMS_TRACE.trace_pause              constant INTEGER := 4096;
  DBMS_TRACE.trace_resume             constant INTEGER := 8192;
  DBMS_TRACE.trace_limit              constant INTEGER := 16;
  ```

By combining the DBMS_TRACE constants, you can enable tracing of multiple PL/SQL language features simultaneously. Note that the constants that control the tracing behavior (such as DBMS_TRACE.trace_pause) should not be used in combination with the other constants (such as DBMS_TRACE.trace_enabled_calls).

To turn on tracing from all programs executed in your session, issue this call:

```
DBMS_TRACE.SET_PLSQL_TRACE (DBMS_TRACE.trace_all_calls);
```

To turn on tracing for all exceptions raised during the session, issue this call:

```
DBMS_TRACE.SET_PLSQL_TRACE (DBMS_TRACE.trace_all_exceptions);
```

You then run your code. When you are done, you stop the trace session by calling:

```
DBMS_TRACE.CLEAR_PLSQL_TRACE;
```

You can then examine the contents of the trace file. The names of these files are generated by Oracle; you will usually look at the modification dates to figure out which file to examine. The location of the trace files is discussed in the later section, "Format of collected data."

Note that you cannot use PL/SQL tracing with the shared server (formerly known as the multithreaded server, or MTS).

Controlling trace file contents

The trace files produced by DBMS_TRACE can get *really* big. You can focus the output by enabling only specific programs for trace data collection. Note that you cannot use this approach with remote procedure calls.

To enable a specific program for tracing, you can alter the session to enable any programs that are created or replaced in the session. To take this approach, issue this command:

```
ALTER SESSION SET PLSQL_DEBUG=TRUE;
```

If you don't want to alter your entire session, you can recompile a specific program unit in debug mode as follows (not applicable to anonymous blocks):

```
ALTER [PROCEDURE | FUNCTION | PACKAGE BODY] program_name COMPILE DEBUG;
```

After you have enabled the programs in which you're interested, the following call will initiate tracing just for those program units:

```
DBMS_TRACE.SET_PLSQL_TRACE (DBMS_TRACE.trace_enabled_calls);
```

You can also restrict the trace information to only those exceptions raised within enabled programs with this call:

```
DBMS_TRACE.SET_PLSQL_TRACE (DBMS_TRACE.trace_enabled_exceptions);
```

If you request tracing for all programs or exceptions and also request tracing only for enabled programs or exceptions, the request for "all" takes precedence.

Pausing and resuming the trace process

The SET_PLSQL_TRACE procedure can do more than just determine which information will be traced. You can also request that the tracing process be paused and resumed. The following statement, for example, requests that no information be gathered until tracing is resumed:

```
DBMS_TRACE.SET_PLSQL_TRACE (DBMS_TRACE.trace_pause);
```

DBMS_TRACE will write a record to the trace file to show when tracing was paused and/or resumed.

Use the DBMS_TRACE.trace_limit constant to request that only the last 8,192 trace events of a run be preserved. This approach helps ensure that you can turn tracing on without overwhelming the database with trace activity. When the trace session ends, only the last 8,192 records are saved.

Format of collected data

If you request tracing only for enabled program units and the current program unit is not enabled, no trace data is written. If the current program unit is enabled, call tracing writes out the program unit type, name, and stack depth.

Exception tracing writes out the line number. Raising an exception records trace information on whether the exception is user-defined or predefined, and records the exception number in the case of predefined exceptions. If you raise a user-defined exception, you will always see an error code of 1.

Here is an example of the output from a trace of the showemps procedure:

```
*** 1999.06.14.09.59.25.394
*** SESSION ID:(9.7) 1999.06.14.09.59.25.344
------------ PL/SQL TRACE INFORMATION -----------
Levels set :  1
Trace:  ANONYMOUS BLOCK: Stack depth = 1
Trace:   PROCEDURE SCOTT.SHOWEMPS: Call to entry at line 5 Stack depth = 2
Trace:    PACKAGE BODY SYS.DBMS_SQL: Call to entry at line 1 Stack depth = 3
Trace:     PACKAGE BODY SYS.DBMS_SYS_SQL: Call to entry at line 1 Stack depth = 4
Trace:     PACKAGE BODY SYS.DBMS_SYS_SQL: ICD vector index = 21 Stack depth = 4
Trace:    PACKAGE PLVPRO.P: Call to entry at line 26 Stack depth = 3
Trace:    PACKAGE PLVPRO.P: ICD vector index = 6 Stack depth = 3
Trace:    PACKAGE BODY PLVPRO.P: Call to entry at line 1 Stack depth = 3
Trace:    PACKAGE BODY PLVPRO.P: Call to entry at line 1 Stack depth = 3
Trace:     PACKAGE BODY PLVPRO.P: Call to entry at line 1 Stack depth = 4
```

Tuning PL/SQL Programs

Tuning an Oracle application is a complex process: you need to tune the SQL in your code base, make sure the System Global Area is properly configured, optimize algorithms, and so on. Tuning individual PL/SQL programs is a bit less daunting, but still more than enough of a challenge. Before spending lots of time improving the performance of your PL/SQL code, you should first:

Tune access to code and data in the SGA
: Before your code can be executed (and perhaps run too slowly), it must be loaded into the SGA of the Oracle instance. This process can benefit from a focused tuning effort, usually performed by a DBA. You will find more information about the SGA and other aspects of PL/SQL internals in Chapter 23.

Optimize your SQL
: In virtually any application you write against the Oracle RDBMS, the vast majority of tuning will take place by optimizing the SQL statements executed against your data. The potential inefficiencies of a 16-way join dwarf the usual issues

found in a procedural block of code. To put it another way, if you have a program that runs in 20 hours, and you need to reduce its elapsed time to 30 minutes, virtually your only hope will be to concentrate on the SQL within your code. There are many third-party tools available to both DBAs and developers that perform very sophisticated analyses of SQL within applications and recommend more efficient alternatives.

Once you are confident that the "context" in which your PL/SQL code is run is not obviously inefficient, you should turn your attention to the code base. I suggest the following steps:

Write your application with best practices and standards in mind
: While you shouldn't take clearly inefficient approaches to meeting requirements, you also shouldn't obsess about the performance implications of every line in your code. Remember that most of the code you write will never be a bottleneck in your application's performance, so you don't have to optimize it. Instead, get the application done and then...

Analyze your application's execution profile
: Does it run quickly enough? If it does, great: you don't need to do any tuning (at the moment). If it's too slow, identify which specific elements of the application are causing the problem and then focus directly on those programs (or parts of programs). Once identified, you can then...

Tune your algorithms
: As a procedural language, PL/SQL is often used to implement complex formulas and algorithms. You can use conditional statements, loops, perhaps even GOTOs and (I hope) reusable modules to get the job done. These algorithms can be written in many different ways, some of which perform very badly. How do you tune poorly written algorithms? This is a tough question with no easy answers. Tuning algorithms is much more complex than tuning SQL (which is "structured" and therefore lends itself more easily to automated analysis).

Take advantage of any PL/SQL-specific performance features
: Over the years, Oracle has added statements and optimizations that can make a substantial difference to the execution of your code. Consider using constructs ranging from the RETURNING clause to FORALL. Make sure you aren't living in the past and paying the price in application inefficiencies.

It's outside the scope of this book to offer substantial advice on SQL tuning and database/SGA configuration. Even a comprehensive discourse on PL/SQL tuning alone would require multiple chapters. Further, developers often find that many tuning tips have limited or no impact on their particular environments. In the remainder of this chapter, I will present some ideas on how to analyze the performance of your code and then offer a limited amount of tuning advice that will apply to the broadest range of applications.

Analyzing Performance of PL/SQL Code

Before you can tune your application, you need to figure out what is running slowly and where you should focus your efforts. Oracle and third-party vendors offer a variety of products to help you do this; generally they focus on analyzing the SQL statements in your code, offering alternative implementations, and so on. These tools are very powerful, yet they can also be very frustrating to PL/SQL developers. They tend to offer an overwhelming amount of performance data without telling you what you really want to know: how fast did a particular program run and how much did the performance improve after making this change?

To answer these questions, Oracle offers a number of built-in utilities. Here are the most useful:

DBMS_PROFILER
: This built-in package allows you to turn on execution profiling in a session. Then, when you run your code, Oracle uses tables to keep track of detailed information about how long each line in your code took to execute. You can then run queries on these tables or—much preferred—use screens in products like Toad or SQL Navigator to present the data in a clear, graphical fashion.

DBMS_UTILITY.GET_TIME
: Use this built-in function to calculate the elapsed time of your code down to the hundredth of a second. The scripts *tmr.ot* and *plvtmr.pkg* (available on the book's web site) offer an interface to this function that allows you to use "timers" (based on DBMS_UTILITY.GET_TIME) in your code. These make it possible to time exactly how long a certain operation took to run and even to compare various implementations of the same requirement.

> In Oracle Database 10g, you can also call DBMS_UTILITY.GET_CPU_TIME to calculate elapsed CPU time.

In case you do not have access to a tool that offers an interface to DBMS_PROFILER, here are some instructions and examples.

First of all, Oracle does not install DBMS_PROFILER for you automatically. To see if DBMS_PROFILER is installed and available, connect to your schema in SQL*Plus and issue this command:

```
SQL> DESC DBMS_PROFILER
```

If you then see the message:

```
ERROR:
ORA-04043: object dbms_profiler does not exist
```

you will have to install the program.

For early Oracle versions, such as Oracle7 and Oracle8 Database, you need to ask your DBA to run the following scripts under a SYSDBA account (the first creates the package specification, the second the package body):

$ORACLE_HOME/rdbms/admin/dbmspbp.sql
$ORACLE_HOME/rdbms/admin/prvtpbp.plb

For later versions, you need to run the *$ORACLE_HOME/rdbms/admin/profload.sql* file instead, also under a SYSDBA account.

You then need to run the $*ORACLE_HOME/rdbms/admin/proftab.sql* file in your own schema to create three tables populated by DBMS_PROFILER:

PLSQL_PROFILER_RUNS
Parent table of runs

PLSQL_PROFILER_UNITS
Program units executed in run

PLSQL_PROFILER_DATA
Profiling data for each line in a program unit

Finally, you will probably find it helpful to take advantage of some sample queries and reporting packages offered by Oracle in the following files:

$ORACLE_HOME/plsql/demo/profrep.sql
$ORACLE_HOME/plsql/demo/profsum.sql

Once all these objects are defined, you gather profiling information for your application by writing code like this:

```
BEGIN
   DBMS_OUTPUT.PUT_LINE (
      DBMS_PROFILER.START_PROFILER (
         'showemps ' ||
         TO_CHAR (SYSDATE, 'YYYYMMDD HH24:MI:SS')
         )
      );
   showemps;
   DBMS_OUTPUT.PUT_LINE (
      DBMS_PROFILER.STOP_PROFILER);
END;
```

Once you have finished running your application code, you can run queries against the data in the PLSQL_PROFILER_ tables. Here is an example of such a query that displays those lines of code that consumed at least 1% of the total time of the run:

```
/* File on web: slowest.sql */
SELECT      TO_CHAR (
               p1.total_time / 10000000,
               '99999999')
         || '-'
         || TO_CHAR (p1.total_occur) AS time_count,
            p2.unit_owner || '.' || p2.unit_name unit,
            TO_CHAR (p1.line#)
         || '-'
         || p3.text text
    FROM plsql_profiler_data p1,
         plsql_profiler_units p2,
         all_source p3,
         plsql_profiler_grand_total p4
   WHERE p2.unit_owner NOT IN ('SYS', 'SYSTEM')
     AND p1.runid = &&firstparm
     AND (p1.total_time >= p4.grand_total / 100)
     AND p1.runid = p2.runid
     AND p2.unit_number = p1.unit_number
     AND p3.TYPE = 'PACKAGE BODY'
     AND p3.owner = p2.unit_owner
     AND p3.line = p1.line#
     AND p3.NAME = p2.unit_name
ORDER BY p1.total_time DESC;
```

As you can see, these queries are fairly complex (I modified one of the canned queries from Oracle to produce the above four-way join). That's why it is far better to rely on a graphical interface in a PL/SQL development tool.

After you've analyzed your code and identified bottlenecks, the following sections can help you determine what kinds of changes to make to improve code performance.

Optimizing PL/SQL Performance

This section contains brief recommendations for ways to improve the performance of your code and points you to other sections in the book that cover each topic more thoroughly.

Use the most aggressive compiler optimization level possible

Oracle Database 10*g* Release 1 introduced an optimizing compiler for PL/SQL programs. The default optimization level of 2 takes the most aggressive approach possible in terms of transforming your code to make it run faster. You should use this default level unless compilation time is unacceptably slow, and you are not seeing benefits from optimization. See the "The Optimizing Compiler" section in this chapter for detailed information.

Use BULK COLLECT when querying multiple rows

The BULK COLLECT statement retrieves multiple rows of data through either an implicit or an explicit query with a single round trip to and from the database. BULK COLLECT reduces the number of context switches between the PL/SQL and SQL engines and thereby reduces the overhead of retrieving data. Rather than using a cursor FOR loop or other row-by-row querying mechanism, switch to BULK COLLECT for a dramatic improvement in performance. See the "BULK COLLECT" section in Chapter 15 for more about this feature.

Use FORALL when modifying multiple rows

As with BULK COLLECT, FORALL greatly reduces context switching between the PL/SQL and SQL engines, but this time for updates, inserts, and deletes. You can expect to see an order of magnitude (or greater) improvement in performance for multiple-row DML execution with FORALL. See the "Bulk DML with the FORALL Statement" section in Chapter 14 for detailed information.

Use the NOCOPY hint when passing large structures

The NOCOPY parameter hint requests that the PL/SQL runtime engine pass an IN OUT argument by reference rather than by value. This can speed up the performance of your programs, because by-reference arguments are not copied within the program unit. When you pass large, complex structures like collections, records, or objects, this copy step can be expensive. See the "The NOCOPY Parameter Mode Hint" section in Chapter 17.

Use PLS_INTEGER for intensive integer computations.

When you declare an integer variable as PLS_INTEGER, it will use less memory than INTEGER and rely on machine arithmetic to get the job done more efficiently. In a program that requires intensive integer computations, simply changing the way that you declare your variables could have a noticeable impact on performance. See the section "The PLS_INTEGER Type" in Chapter 9 for a more detailed discussion.

In Oracle8*i* Database and Oracle9*i* Database, PLS_INTEGER will perform more efficiently than BINARY_INTEGER. In Oracle Database 10*g*, they are equally performant.

Use BINARY_FLOAT or BINARY_DOUBLE for floating-point arithmetic

Oracle Database 10*g* introduces two, new floating-point types: BINARY_FLOAT and BINARY_DOUBLE. These types conform to the IEEE 754 floating-point standard and use native machine arithmetic, making them more efficient than NUMBER or INTEGER variables. See "The BINARY_FLOAT and BINARY_DOUBLE Types" section in Chapter 9.

Group together related programs in a package

Whenever you reference any single element in a package for the first time in your session, the entire package is cached in the shared memory pool. Any other calls to programs in the package require no additional disk I/O, thereby improving the performance of calling those programs. Group related programs into a package to take advantage of this feature. See the "Packaging to improve memory use and performance" section in Chapter 23 for details.

Pin into shared memory large and frequently executed programs.

Pin frequently accessed programs in the shared memory pool with the DBMS_SHARED_POOL.PIN procedure. A pinned program will not be flushed out of the pool using the default least-recently-used algorithm. This guarantees that the code will already be present when it is need. See the "What to Do if You Run Out of Memory" section in Chapter 23.

Protecting Stored Code

Virtually any application we write contains propriety information. If I write my application in PL/SQL and sell it commercially, I really don't want to let customers (or worse, competitors) see my secrets. Oracle offers a program known as *wrap* that hides or *obfuscates* most, if not all, of these secrets.

Some people refer to "wrapping" code as "encrypting" code, but wrapping is not true encryption. If you need to deliver information, such as a password, that *really* needs to be secure, you should not rely upon this facility. Oracle does provide a way of incorporating true encryption into your own applications using the built-in package DBMS_CRYPTO (or DBMS_OBFUSCATION_TOOLKIT in releases before Oracle Database 10*g*). Chapter 22 describes encryption and other aspects of PL/SQL application security.

When you wrap PL/SQL source, you convert your readable ASCII text source code into unreadable ASCII text source code. This unreadable code can then be distributed to customers, regional offices, etc., for creation in new database instances. The Oracle database maintains dependencies for this wrapped code as it would for programs compiled from readable text. In short, a wrapped program is treated within the database just as normal PL/SQL programs are treated; the only difference is that prying eyes can't query the USER_SOURCE data dictionary to extract trade secrets.

Oracle has, for years, provided a *wrap* executable that would perform the obfuscation of your code. With Oracle Database 10*g* Release 2, you can also use the DBMS_DDL.WRAP and DBMS_DDL.CREATE_WRAPPED programs to wrap dynamically constructed PL/SQL code.

Restrictions on and Limitations of Wrapping

You should be aware of the following issues when working with wrapped code:

- Wrapping makes reverse engineering of your source code difficult, but you should still avoid placing passwords and other highly sensitive information in your code.

- You cannot wrap the source code in triggers. If it is critical that you hide the contents of triggers, move the code to a package and then call the packaged program from the trigger.
- Wrapped code cannot be compiled into databases of a version lower than that of the *wrap* program. Wrapped code is upward-compatible only.
- You cannot include SQL*Plus substitution variables inside code that must be wrapped.

Using the Wrap Executable

To wrap PL/SQL source code, you run the *wrap* executable. This program, named *wrap.exe*, is located in the *bin* directory of the Oracle instance. The format of the *wrap* command is:

```
wrap iname=infile [oname=outfile]
```

where *infile* points to the original, readable version of your program, and *outfile* is the name of the file that will contain the wrapped version of the code. If *infile* does not contain a file extension, then the default of *sql* is assumed.

If you do not provide an oname argument, then *wrap* creates a file with the same name as *infile* but with a default extension of *plb*, which stands for "PL/SQL binary" (a misnomer, but it gets the idea across: binaries are, in fact, unreadable).

Here are some examples of using the *wrap* executable:

- Wrap a program, relying on all the defaults:

  ```
  wrap iname=secretprog
  ```

- Wrap a package body, specifying overrides of all the defaults. Notice that the wrapped file doesn't have to have the same filename or extension as the original:

  ```
  wrap iname=secretbody.spb oname=shhhhhh.bin
  ```

Dynamic Wrapping with DBMS_DDL

Oracle Database 10*g* Release 2 provides a way to wrap code that is generated dynamically: the WRAP and CREATE_WRAPPED programs of the DBMS_DDL package:

DBMS_DDL.WRAP
: Returns a string containing an obfuscated version of your code

DBMS_DDL.CREATE_WRAPPED
: Compiles an obfuscated version of your code into the database

Both programs are overloaded to work with a single string and with arrays of strings based on the DBMS_SQL.VARCHAR2A and DBMS_SQL.VARCHAR2S collection types. Here are two examples that use these programs:

- Obfuscate and display a string that creates a tiny procedure:

```
SQL> DECLARE
  2     l_program   VARCHAR2 (32767);
  3  BEGIN
  4     l_program := 'CREATE OR REPLACE PROCEDURE dont_look IS BEGIN NULL; END;';
  5     DBMS_OUTPUT.put_line (SYS.DBMS_DDL.wrap (l_program));
  6  END;
  7  /
CREATE OR REPLACE PROCEDURE dont_look wrapped

a000000
369
abcd
....
XtQ19EnOI8a6hBSJmk2NebMgPHswg5nnm7+fMr2ywFy4CP6Z9P4I/v4rpXQruMAy/tJepZmB
CC0r
uIHHLcmmpkOCnm4=
```

- Read a PL/SQL program definition from a file, obfuscate it, and compile it into the database:

```
/* File on web: obfuscate_from_file.sql */
CREATE OR REPLACE PROCEDURE obfuscate_from_file (
   dir_in    IN   VARCHAR2
 , file_in   IN   VARCHAR2
)
IS
   l_file    UTL_FILE.file_type;
   l_lines   DBMS_SQL.varchar2s;

   PROCEDURE read_file (lines_out IN OUT NOCOPY DBMS_SQL.varchar2s)
   IS BEGIN ... not critical to the example ... END read_file;
BEGIN
   read_file (l_lines);
   SYS.DBMS_DDL.create_wrapped (l_lines, l_lines.FIRST, l_lines.LAST);
END obfuscate_from_file;
```

Guidelines for Working with Wrapped Code

I have found the following guidelines useful in working with wrapped code:

- Create batch files so that you can easily, quickly, and uniformly wrap one or more files. In Windows NT, I create *bat* files that contain lines like this in my source code directories:

  ```
  c:\orant\bin\wrap iname=plvrep.sps oname=plvrep.pls
  ```

  Of course, you can also create parameterized scripts and pass in the names of the files you want to wrap.

- You can only wrap package specifications and bodies, object type specifications and bodies, and standalone functions and procedures. You can run the wrapped binary against any other kind of SQL or PL/SQL statement, but those files will not be changed.

- You can tell that a program is wrapped by examining the program header. It will contain the keyword WRAPPED, as in:

  ```
  PACKAGE BODY package_name WRAPPED
  ```

  Even if you don't notice the keyword WRAPPED on the first line, you will immediately know that you are looking at wrapped code because the text in USER_SOURCE will look like this:

  ```
     LINE TEXT
  ------- ----------------------
       45 abcd
       46 95a425ff
       47 a2
       48 7 PACKAGE:
  ```

  and no matter how bad your coding style is, it surely isn't *that* bad!

- Wrapped code is much larger than the original source. I have found in my experience that a 57 KB readable package body turns into a 153 KB wrapped package body, while an 86 KB readable package body turns into a 357 KB wrapped package body. These increases in file size do result in increased requirements for storing source code in the database. The size of compiled code stays the same, although the time it takes to compile may increase.