PL/SQL conditional compilation

An Oracle White Paper
October 2005

ORACLE

NOTE

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

PL/SQL conditional compilation

CONTENTS

PL/SQL conditional compilation

ABSTRACT

Oracle Database 10*g* Release 2 delivers a new PL/SQL language feature: conditional compilation.

The feature is elegant, easy to understand, and has many interesting uses; some of these may not have occurred to you. This paper illustrates conditional compilation with code samples and demonstrates every conditional compilation construct. It recommends best practices and discusses alternative implementations.

Unusually, but for very compelling reasons, the feature has been made available in patchsets of releases of Oracle Database earlier than the one that introduced the feature. It is available, with some functionality restrictions, in the first release of Oracle Database 10*g* from *10.1.0.4* onwards and in Oracle9*i* Database from *9.2.0.6* onwards. However, customers who do not use the new conditional compilation constructs will see no change whatsoever in the way their PL/SQL programs are compiled; this is true for all of the releases that support PL/SQL conditional compilation.

The paper's final section explains the restrictions in the feature's functionality in these earlier releases and the rationale for the decision to make the feature so available. Oracle Independent Software Vendors, and in particular Oracle's Applications Division, are most likely to benefit from this retrospective availability. But any customer who needs to deploy the same application in different releases of Oracle Database might find this useful.

This paper does not attempt to be a reference manual for the feature. Rather, it assumes an understanding of the functionality and the syntax. It does, however, describe some aspects of the feature which — for reasons which will become obvious — are not described in the Documentation Library.

INTRODUCTION

"We participated in the Beta Program for Oracle Database 10*g* Release 2 and extensively tested PL/SQL conditional compilation. We found it functionally complete and easy to use. It will allow us to write a more manageable and faster implementation for our component-based architecture. We intend to use it in our products at the earliest opportunity."

— Håkan Arpfors
Senior Software Architect
IFS
www.ifsworld.com

Oracle Database 10*g* Release 2 delivers a new PL/SQL language feature: conditional compilation. The feature is elegant and easy to understand.

Conditional compilation delivers many benefits and is well known in programming environments other than PL/SQL[1]. Broadly speaking, it allows constructs — with formally defined syntax and semantics — to be used to mark up text so that a so-called preprocessor[2] can deterministically derive the text that will be submitted to the compiler proper. The following two uses are, perhaps, the most famous.

- Allowing self-tracing code to be turned on during development and to be turned off when the code goes live. PL/SQL conditional compilation supports this use in a direct and obvious way (see *Latent self-tracing code* on page 23).
- Allowing alternative code fragments, each appropriate for the peculiarities of a particular operating system and inappropriate or illegal for other operating systems, to coexist in the same source text so the correct fragment can be selected for compilation according to the circumstances. PL/SQL is, by its nature, operating system independent; but analogous challenges present themselves when a PL/SQL compilation unit must be deployed in several different releases of Oracle Database. Newer releases introduce new features with new syntax and programs that take advantage of these are illegal in earlier releases. PL/SQL conditional compilation supports this use in an elegant and powerful way (see *Spanning different releases of Oracle Database with a single source code corpus* on page 46).

There are many other uses; the following cases have been selected for discussion in this paper. (The order is the most natural for explanation. Readers may decide which they find the most valuable.)

- Allowing *assertions*[3] to be turned on during development and to be turned off when the code goes live. PL/SQL conditional compilation supports this use in a direct and obvious way (see *Latent assertions* on page 24).
- PL/SQL conditional compilation allows new approaches to unit testing. For example, tests for private helper subprograms may be coded in the body of the package that contains them (see *Unit testing of subprograms declared only in a package body* on page 27).
- Sometimes the unit test for a particular subprogram needs to demonstrate that it handles exceptional conditions raised by the subprograms it calls. However, it can be difficult to contrive these problems at will. PL/SQL conditional

---

1. See *www.google.com/search?q=%22conditional+compilation%22*
2. Readers who are familiar with other preprocessors might appreciate a pointer to the section *How does PL/SQL conditional compilation compare with similar features in other programming environments?* on page 57.
3. Roughly, an assertion is a test — which necessarily takes time — that program state at a particular point in the code is as expected; typically, in PL/SQL, an exception is raised if the test fails.

compilation supports this testing requirement in a simple way (see *Mock objects* on page 34).

- A developer often realizes that more than one approach to the design of a subprogram will result in its correct behavior; sometimes the alternative approaches result in source code versions which are textually largely the same but which differ critically in small areas distributed fairly evenly thought the source. PL/SQL conditional compilation allows all the approaches to be coded in a single source text — while they are being evaluated — and thereby eliminates the risk of carelessly introduced unintended differences (see *Comparing competing implementations during prototyping* on page 37).
- Oracle ISVs sometimes sell applications which provide optional extra functionality for incremental cost. The modular delivery is implemented by optional PL/SQL compilation units which are installed according to what the customer has licensed. PL/SQL's dependency model prevents the core part of the application referring statically to optional components that are not installed. However, the core part of the application should not need re-installation in order to accommodate the installation of a new optional component. This has forced the use of dynamic invocation — which has some drawbacks. Conditional compilation allows a new approach. (see *Component based installation* on page 40).

Before the discussion of the use cases, the section *PL/SQL conditional compilation constructs* on page 5 illustrates the full set of primitives that expose the feature; and the section *How does PL/SQL conditional compilation work?* on page 17 explains the introduction of conditional compilation as a new stage in the compilation pipeline and shows how the programmer can inspect its output.

It is always hard to find illustrative code samples that are, on the one hand, brief and easy to present and are, on the other hand, sufficiently non-trivial that they unequivocally show the benefit of a feature without making large demands on the reader's ability to extrapolate. The code included in the section *PL/SQL conditional compilation use cases* on page 23 errs on the side of triviality. To compensate, the section *Case study: implementing unit testing, assertions, and tracing for a fast cube root body-private helper function* on page 49 is so realistic that the complete, self-contained SQL*Plus scripts occupy about ten pages. Reading this section is optional; but those who do study it will be rewarded by seeing the approaches described in the individual use cases applied in concert to solve a real problem in an effective and highly usable way.

The section *How does PL/SQL conditional compilation compare with similar features in other programming environments?* on page 57 addresses questions that programmers who are familiar with other implementations of conditional compilation might naturally ask.

Unusually, but for very compelling reasons, PL/SQL conditional compilation has been made available in patchsets of releases of Oracle Database earlier than the one that introduced the feature. It is available in the first release of Oracle Database 10*g* from *10.1.0.4* onwards and in Oracle9*i* Database from *9.2.0.6* onwards[4]. However, customers who do not use the new conditional compilation constructs[5] will see no change whatsoever in the way their PL/SQL programs are compiled; this is true for all of the releases that support PL/SQL conditional compilation.

The paper's final section, *The availability of PL/SQL conditional compilation in Oracle Database 10g Release 1 and in Oracle9i Database Release 2* on page 59, explains the restrictions in the feature's functionality in these earlier releases and the rationale for the decision to make the feature so available.

This paper does not attempt to be a reference manual for the feature. Rather, it assumes an understanding of the functionality and the syntax. References to the product documentation are given in *Appendix B: Oracle Database Documentation Library references* on page 68. It does, however, describe some aspects of the feature which — for reasons which will become obvious — are not described in the Documentation Library.

Sadly, but realistically, this paper is likely to have minor spelling and grammar errors. For that reason alone, it is bound to be revised periodically[6]. Other interesting applications of PL/SQL conditional compilation cases might come to our attention and — it is hoped — the paper will be revised to discuss them. Therefore, before settling down to study the paper, readers should ensure that they have the latest copy — for which the URL is given in the page's header.

URLs sometimes change. But this one will always take you to the Oracle Technical Network's PL/SQL Technology Center:

www.oracle.com/technology/tech/pl_sql

Even in the unlikely event that the paper is moved, it will still be easy to find from that page.

---

4. In *10.1.0.4*, the feature is available by default but can be totally disabled by setting the PL/SQL conditional compilation underscore parameter to *"disable condtional compilation"*.

   In *9.2.0.6*, the feature is totally disabled by default but can be made available by setting the PL/SQL conditional compilation underscore parameter to *"enable condtional compilation"*.

   In Oracle Database 10*g* Release 2 and later, the feature cannot be disabled and the PL/SQL conditional compilation underscore parameter is obsolete.

5. Any of the new PL/SQL conditional compilation constructs, if used in a release of Oracle Database that does not support the feature, will cause a compilation error. This guarantees that no compilable extant code will be affected in any way when it is compiled in a release that *does* support the feature.

6. This document's change history is listed at the end. See *Appendix A: Change History* on page 67.

PL/SQL CONDITIONAL COMPILATION CONSTRUCTS

First, for readers who are new to this feature, *Code_1*[7] — so far without an explanation[8] — shows a simple example as a self-contained SQL*Plus script[9,10]. Here, as in all the code examples in this paper, the PL/SQL conditional compilation constructs are emphasized typographically.

```
-- Code_1
alter session set PLSQL_CCFlags = 'Use_IEEE:2'
/
create or replace procedure P is
  -- Notice that $if ... $end interrupts a regular statement.
  n $if    $$Use_IEEE = 0 $then number;
    $elsif $$Use_IEEE = 1 $then binary_float;
    $else $error 'Illegal Use_IEEE: '||$$Use_IEEE $end
    $end
begin
  $if $$Use_IEEE = 0 $then
    n := 1.0;
  $else
    n := 1.0f;
  $end
  Print(n);
end P;
/
SHOW ERRORS
alter procedure P compile
  PLSQL_CCFlags = 'Use_IEEE:0' reuse settings
/
begin P(); end;
/
alter procedure P compile
  PLSQL_CCFlags = 'Use_IEEE:1' reuse settings
/
begin P(); end;
/
```

Code_1 produces this output:

```
...
5/11     PLS-00179: $ERROR: Illegal Use_IEEE: 2
1
1.0E+000
```

The way *n* is rendered in the two versions reflects the rules for the default conversion of, respectively, a *number* and a *binary_float* variable to *varchar2*.

---

7. I don't like to split code illustrations across page boundaries. That's why you'll sometimes see unused white space at the bottom of a page.

8. The meaning of constructs *$if*, *$then*, *$elsif*, *$else*, and *$end* can easily be guessed. Readers who are impatient to understand the object of the test, *$$Use_IEEE*, and how it is defined using the new PL/SQL compilation parameter *PLSQL_CCFlags*, can look ahead to the section *The inquiry directive* on page 9. And those who are impatient to understand the *$error* construct can look ahead to the section *The error directive* on page 12.

9. The schema-level procedure *Print()* — a simple wrapper for *DBMS_Output.Put_Line()*, and effectively a "synonym" for that packaged procedure — is used in the code samples to shorten them.

10. When I write *Some_Proc()* I mean the procedure or function itself. (It might be declared at schema level, at top level in a package, or nested to any deep level in a *declare... begin... end* block.) When I write *Some_Proc* — without the trailing parentheses — I mean the schema level compilation unit itself.

There are three kinds of PL/SQL conditional compilation construct: the *selection directive*, the *inquiry directive*, and the *error directive*. *Code_1* uses all of these. The following sections discuss them in detail and suggest some best-practice principles for their use.

The *selection directive*

The motivating requirement for conditional compilation is that it must select between alternative fragments of source text at compilation time. PL/SQL conditional compilation satisfies this requirement with the *selection directive*. *Code_2* illustrates this.

```
-- Code_2
$if CC_Control.Trace_Level > 0 $then
  Print(Sparse_Collection.Count());
  $if CC_Control.Trace_Level > 1 $then
    declare Idx Idx_t := Sparse_Collection.First();
    begin
      while Idx is not null loop
        Print(Idx||' '||Sparse_Collection(Idx));
        Idx := Sparse_Collection.Next(Idx);
      end loop;
    end;
  $end
$end
```

Code_2 relies on the package *CC_Control*:

```
package CC_Control is
  Trace_Level constant pls_integer := 2;
end CC_Control;
```

The *selection directive* must test a so-called static *boolean* expression. The rules that such an expression must satisfy are defined in the *PL/SQL User's Guide and Reference* book; the evaluation of a static expression must always give the same result unless anything it depends on — a static package constant or the result of an *inquiry directive* (see *The inquiry directive* on page 9) — has been deliberately changed. A static *boolean* expression must be composed using so-called static package constants, literals, or *inquiry directives*. Thus, for example, *d* in the following is not a static package constant:

```
package CC_Control is
  ...
  d constant pls_integer := To_Char(Sysdate, 'J');
end CC_Control;
```

Therefore, the following is not a static *boolean* expression:

```
CC_Control.d > 0 or CC_Control.Trace_Level
```

The *selection directive* uses these special building blocks:

```
$if   $then   $elsif   $else   $end
```

The meaning of each is exactly symmetrical with that of the corresponding run-time *if* building block. (Notice, though, that the *selection directive* ends with just *$end* rather than with *$end $if*.)

The rules for evaluating the static *boolean* expression that the *selection directive* uses are the same rules that PL/SQL uses at run-time. In particular, *null* has its usual significance.

Moreover, when a *selection directive* refers to a static package constant, a dependency is created from the current compilation unit to the package where

the constant is declared. (Such a static package constant must be declared in a compilation unit other than the one where it is used.) Thus, if a static package constant is changed[11], then all compilation units with *selection directives* that use the constant will be invalidated and will therefore be recompiled to use the changed value on their next use.

We[12] recommend that a package that houses a constant that controls conditional compilation should contain nothing but such declarations. (As a corollary, such a package will not have a body.) This best practice principle minimizes consequential invalidations when an element in the package specification is changed.

All these rules — the symmetry with the run-time *if* construct, the way a static *boolean* expression is evaluated, and the way dependencies are set up and guarantee system-wide integrity — can be immediately understood by the PL/SQL programmer. In fact, the static *boolean* expression is evaluated by the identical code in the Oracle executable that would evaluate the same expression at run-time. These properties contribute to making the PL/SQL conditional compilation feature so elegant and easy to use.

The compile-time *$if* construct[13] can always be substituted for the run-time *if* construct — provided that it tests only a static *boolean* expression. Moreover, it can be used in ways that the run-time *if* construct cannot.

- It can be used to select not only between alternative executable statements but also between different *declaration* statements. *Code_26* on page 31 illustrates this.
- It can interrupt a regular PL/SQL statement. *Code_1* on page 5 illustrates this.

Both these differences can be readily understood by realizing that the *selection directive* selects fragments of regular PL/SQL text for compilation "proper".

Code_3 shows another example of the *selection directive* to make the point that source text that is *not* selected — and which the next stage of compilation therefore never sees — need not be syntactically correct.

```
-- Code_3
$if CC_Control.Some_Boolean $then
  Print('Ok. $if kicked in.');
$else
  -- The PL/SQL conditional compilation rules ensure
  -- that the apparently significant $end
  -- is not taken as such in this comment.
  Print(('bad');
$end
```

---

11. Of course, a static package constant can be changed only by editing the source of the package that houses it and by recompiling it.

12. I use "we" in this paper to denote Oracle's PL/SQL Team. In particular "we recommend" indicates that the Team has discussed the issue and is making a consensus recommendation. I use "I" when I'm referring to, for example, a design decision I took regarding one of the code examples or the results I got when I ran it.

13. The term "compile-time *$if* construct" will often be used as an informal synonym for the proper term, *selection directive*, in contexts which discuss the differences and similarities between this and the run-time *if* construct.

Code_3 will compile without error when *CC_Control.Some_Boolean* is *true*; otherwise it will fail to compile[14]. Notice that the presence of what the human would read as the special building blocks *$if* inside the text literal for *Print()* and *$end* within a comment causes no confusion for the PL/SQL compiler. This reflects the fact that the processing of the conditional compilation directives is done by the PL/SQL compiler itself. This is discussed more in *The PL/SQL compilation pipeline* on page 17.

Finally in this section, it is interesting to notice that the trivial construct *$if false $then* can be surprisingly useful for *ad hoc* "commenting out" during the development cycle. Programmers often want to comment out a region of source text before compiling and running a compilation unit. It is easy to do this by with the *C*-style comment syntax by surrounding the region with an opening */** and a closing ***/**. Sometimes, though, the attempt is subverted because the region already uses *C*-style comments — for example, to denote a SQL hint — and the programmer is forced to comment out each line individually with the **--** comment syntax. *Code_4* shows an example.

```
-- Code_4
procedure P is
begin
  ...
  for j in (select /*+ first_rows(10) */ Object_Name
              from All_Objects
              where rownum <= 10)
  loop
    -- Only the first 10 rows are needed because...
    -- The order doesn't mater because...
    Print(j.Object_Name);
  end loop;
  ...
end P;
```

A text-editor with a reasonable keystroke macro feature can help as *Code_5* shows. But the process is still risky, particularly when the time comes to uncomment the commented out region.

```
-- Code_5
procedure P is
begin
  ...
-->>  -- Note to self: re-activate when...
-->>  for j in (select /*+ first_rows(10) */ Object_Name
-->>              from All_Objects
-->>              where rownum <= 10)
-->>  loop
-->>    -- Only the first 10 rows are needed because...
-->>    -- The order doesn't mater because...
-->>    Print(j.Object_Name);
-->>  end loop;
  ...
end P;
```

---

14. Here is the reported error, tied to its source line:

```
Print(('Ok');
PLS-00103: Encountered the symbol ";" when expecting one of the following:
...
The symbol ")" was substituted for ";" to continue.
```

This is typical when the selected source has syntax errors.

Code_6 shows how *$if false $then* solves the problem.

```
-- Code_6
procedure P is
begin
  ...
  -- Note to self: re-activate when...
  $if false $then
    for j in (select /*+ first_rows(10) */ Object_Name
                from All_Objects
                where rownum <= 10)
    loop
      -- Only the first 10 rows are needed because...
      -- The order doesn't mater because...
      Print(j.Object_Name);
    end loop;
  $end
  ...
end P;
```

The *inquiry directive*

The use of static package constants to determine the outcome of a *selection directive* is powerfully beneficial when *selection directives* in many different compilation units all test the same static package constant — presumably to make a choice that has the same meaning at all these sites. This is because changing the constant will guarantee that every compilation unit that refers to it will pick up the new value before it can next be used. If the purpose is to select code on the basis of the release of the Oracle Database in which the unit is to be compiled, then testing a static package constant is definitely the best approach. This is discussed in detail in the section *Spanning different releases of Oracle Database with a single source code corpus* on page 46.

However, if the purpose is to conditionalize the compilation of just a single unit — for example, *Some_Unit* — then it might seem too heavy handed to use a dedicated partner package — for example, *Some_Unit_CC* — just to control the conditionalization. Programmers will prefer a lighter weight approach — one which corresponds roughly to specifying the conditionalization "on the command line". The *inquiry directive* is provided for this purpose; it is used to obtain a value from the compilation environment:

- from a PL/SQL compilation parameter.[15]
- from a user-defined so-called *ccflag*.[16]
- from one of the two predefined *ccflag*s, *PLSQL_Unit* and *PLSQL_Line*.

User-defined *ccflag*s are defined using the value for the new PL/SQL compilation parameter, *PLSQL_CCFlags*. The *All_PLSQL_Object_Settings* view family therefore has a new column *PLSQL_CCFlags* for this parameter[17].

---

15. A PL/SQL compilation parameter is an initialization parameter that affects how PL/SQL compilation is done. Before the advent of conditional compilation, the set was composed of *NLS_Length_Semantics*, *PLSQL_Optimize_Level*, *PLSQL_Code_Type*, *PLSQL_Debug*, and *PLSQL_Warnings*. The value of each such parameter is stored with each compilation unit as part of its metadata and is revealed by the *All_PLSQL_Object_Settings* view family.

16. The *PL/SQL User's Guide and Reference* book uses "flag" where I use *ccflag*. I invented the term *ccflag* for this document to emphasize its specific significance for PL/SQL conditional compilation.

The SQL*Plus script shown in *Code_7* illustrates the definition of the user-defined *ccflag*s *Flag1* and *Flag2* and their use — together with that of the predefined *ccflag*s and of the some PL/SQL compilation parameters — in several *inquiry directives*.

```
-- Code_7
alter session set PLSQL_Warnings = 'enable:all'
/
alter session set PLSQL_Code_Type = native
/
alter session set PLSQL_CCFlags = 'Flag1:10, Flag2:true'
/
create procedure P is
  v varchar2($$Flag1);
begin
  Print('PLSQL_CCFlags:   '||$$PLSQL_CCFlags);
  Print('PLSQL_Code_Type: '||$$PLSQL_Code_Type);
  Print('PLSQL_Unit:      '||$$PLSQL_Unit);
  Print('PLSQL_Line:      '||$$PLSQL_Line);
  Print('Flag1:           '||$$Flag1);
  $if $$Flag2 $then
    Print('Flag2 is true');
  $end
  $if $$Flag3 is null $then
    Print('Flag3 is null');
  $elsif $$Flag3 $then
    Print('Flag3 is true');
  $else
    Print('Flag3 is false');
  $end
end P;
/
begin P(); end;
/
select
  Attribute||Chr(10)||s.Text||E.Text x
  from User_Source s inner join User_Errors e
    using (Name, Type, Line)
  where Name = 'P' and Type = 'PROCEDURE'
/
```

The *inquiry directive* is formed by preceding the *ccflag* whose value it should produce by *$$*. Notice that the *inquiry directive* can be used not only as the object of the *selection directive*'s test; it can be used also where a regular PL/SQL variable or literal might be used. In particular, it may be used where in regular PL/SQL *only* a literal (and not a variable) may be used — for example, to specify the size of a *varchar2* or of a *varray*.

Procedure P*()* compiles without error and produces this output:

```
PLSQL_CCFlags:   Flag1:10, Flag2:true
PLSQL_Code_Type: NATIVE
PLSQL_Unit:      P
PLSQL_Line:      6
Flag1:           10
Flag2 is true
Flag3 is null
```

17. Notice that *PLSQL_CCFlags* can be used in two ways: first, it supports the specification of name-value pairs for the user-defined *ccflag*s; second (and, after all, why not?) it can be used *per se* as the object of an *inquiry directive*. *Code_7* shows both uses. The second use can be valuable as (part of) the innards of an *error directive*.

Notice that *Flag3*, which was not defined, yielded the value *null*. Notice too that a compilation warning is reported. The query against *user_source inner join user_errors* shows this:

```
WARNING
  Print(Print('Flag3:          '||Nvl ($$Flag3, 'null'));
PLW-06003: unknown inquiry directive '$$FLAG3'
```

Moreover, if *P()* is wrapped, then the *PLW-06003* warning is *not* raised.

This behavior was very carefully designed. It allows *selection directives* to be embedded in code that is deployed for production in such a way that the correct production behavior[18] is guaranteed even when an accidental *alter <PLSQL unit>* command is issued without the *reuse settings* clause. If this happens, then it is very unlikely indeed that the current value of *PLSQL_CCFlags* in the compilation environment will specify the same *ccflag*s with the same values as had previously been stored with the compilation unit. Rather, it is very likely that it will *not* mention the *ccflag*s that were used to conditionalize the unit. Recall that a compile-time *$if* construct — in line with the run-time *if* construct — where the test evaluates to *null* makes the same selection as if the test had evaluated to *false*. The designer should, of course, choose an obscure name for each *ccflag* that is to be used in this way to minimize the risk of an accidental collision (or a hacked discovery). For example, *a#b$0_dbg* would be a reasonably safe choice but *debug* would be risky.

Code_8 shows the recommended way to change the values of the *ccflag*s for an extant compilation unit.

```
-- Code_8
alter procedure P compile PLSQL_CCFlags =
 'Flag1:99, Flag2:false, Flag3:true' reuse settings
/
```

The use of *reuse settings* ensures that the values of other PL/SQL compilation parameters are not accidentally changed[19].

Code_9 shows the recommended way to ensure that every user-defined *ccflag* for an extant compilation unit evaluates to *null*.

```
-- Code_9
alter procedure P compile
  PLSQL_CCFlags ='x:null' reuse settings
/
```

This might seem strange. But whether or not the compilation unit uses *$$x*, the effect will be the same[20].

---

18. Of course, this works only when there *is* an appropriate default behavior. The obvious example is when the *selection directive* guards tracing or assertion code that is to be turned off in production.

19. I have noticed that some non-Oracle writers have used this idiom in articles about PL/SQL conditional compilation:

    ```
    alter session set PLSQL_CCFlags = 'Flag1:99, Flag2:false, Flag3:true'
    /
    alter procedure P compile
    /
    ```

    This is obviously risky: it might recompile *P* with a changed value of *PLSQL_Warnings* or *PLSQL_Optimize_Level*.

Finally, consider the apparent paradox shown in *Code_10*.

```
-- Code_10
alter session set
  PLSQL_CCFlags = 'Some_Flag:1, Some_Flag:2, PLSQL_CCFlags:99'
/
begin
  Print($$Some_Flag);
  Print($$PLSQL_CCFlags);
end;
/
```

In fact, *Code_10* compiles and runs without error to produce this output:

```
2
99
```

We saw no reason to disallow a user-defined *ccflag* whose name collides with itself or with that of a PL/SQL compilation parameter or a predefined *ccflag*.[21] Rather, we have a simple rule for resolving the collision: the last-mentioned user-defined *ccflag* "wins".

The *error directive*

The *error directive* allows the programmer to cause a compilation error at will. Suppose a programmer, writing new code, knows that the logic is incomplete at some location in the source code; however, unless special steps are taken, the code will compile and run. Of course its behavior will be incorrect. Suppose too that the programmer — who inhabits the real world — is suddenly diverted from working on this code and wants a forceful reminder about the coding that remains to be done. The *error directive* allows a new approach[22] as *Code_11* shows.

```
-- Code_11
procedure P is
begin
  ...
  $error '
    Note to self:
    Need to detect and handle "x is null"
    at line '||$$PLSQL_Line||' in unit '||$$PLSQL_Unit
  $end
  ...
end P;
```

The "argument" for the *error directive* must be a static *varchar2* expression. It can be composed using *varchar2* literals, the concatenation operator, and static

---

20. At the time of writing, Bug #4537156 prevents the more obvious approach:
alter procedure P compile PLSQL_CCFlags =" reuse settings

21. PL/SQL allows name collisions in other contexts and there are always rules for resolving them. For example, an inner *declare... begin... end* block can declare a variable whose name collides with one that is declared in a surrounding *declare... begin... end* block. For the *inquiry directive*, the rule is based on a search order. This is documented in the *PL/SQL User's Guide and Reference* book.

22. Inserting, for example, *Raise_Application_Error();* with a suitable message text would have a similar effect; but the reminder would be delayed until run-time — and might be substantially delayed until a particular code-path brought the point of execution to the location in question.

varchar2 functions. Currently, the repertoire of static *varchar2* expressions is very limited. Here are some examples:

```
'Code type is '||$$PLSQL_Code_Type

'Optimize level is '||$$PLSQL_Optimize_Level

-- Pkg.n is a static pls_integer constant
'Pkg.n is '||To_Char(Pkg.n)
```

The *To_Char()* built-in with just one actual argument of datatype *pls_integer* is a static function. (In the second example — with *$$PLSQL_Optimize_Level* — it is invoked implicitly.) Surprisingly, the SQL*Plus script shown in *Code_12* causes *PLS-00178: a static character expression must be used.*

```
-- Code_12
create package Pkg is
  v constant varchar2(1) := 'x';
end Pkg;
/
create procedure P is
begin
  $error Pkg.v $end
end P;
/
SHOW ERRORS
```

The explanation is very simple. The current rules are conservative. Operations involving stringy datatypes are, in general, affected by environment parameters like *NLS_Sort* — and the current implementation favors caution over flexibility.

The *error directive* is particularly useful for trapping "case not found" in a *selection directive* as *Code_13* shows.

```
-- Code_13
package CC_Lov is
  Red    constant pls_integer := 1;
  Blue   constant pls_integer := 2;
  Green  constant pls_integer := 3;
  Color  constant pls_integer := 0;
end CC_Lov;

procedure P is
begin
  $if CC_Lov.Color = CC_Lov.Red $then
    Print('Handling red');

  $elsif CC_Lov.Color = CC_Lov.Blue $then
    Print('Handling blue');

  $elsif CC_Lov.Color = CC_Lov.Green $then
    Print('Handling green');

  $else
    $error
      'Illegal value for CC_Lov.Color: '||CC_Lov.Color
    $end
  $end
end P;
```

With *CC_Lov.Color = 0* as shown, compilation unit *P* fails to compile with *PLS-00179: $ERROR: Illegal value for CC_Lov.Color: 0.* (The intended *PLS-00179* will also be caused if *CC_Lov.Color* is *null*.)

Suppose that procedure P in *Code_13* were rewritten to use an *inquiry directive* as *Code_14* shows.

```
-- Code_14
procedure P is
begin
  $if $$Color = CC_Lov.Red $then
    Print('Handling red');

  $elsif $$Color = CC_Lov.Blue $then
    Print('Handling blue');

  $elsif $$Color = CC_Lov.Green $then
    Print('Handling green');

  $else
    $error
      'Illegal value for Color: '||$$Color
    $end
  $end
end P;
```

And suppose that *P* were then carelessly recompiled thus:

```
alter procedure P compile
  PLSQL_CCFlags = 'Color:true' reuse settings
```

The intended *"Illegal value for Color"* error message would not be caused; rather, the less helpful compilation errors *PLS-00174: a static boolean expression must be used* and *PLS-00306: wrong number or types of arguments in call to '='* would occur. This simply represents a limitation: there is no construct to test the datatype of the value that an *inquiry directive* produces[23].

Choosing between an *inquiry directive* and a static package constant

We recommend controlling conditional compilation with an *inquiry directive* when the normal behavior of the compilation unit can be defined when it has the value *null* — and when a *not null* value is never expected to be needed in the production environment. Using a *not null* value to turn on (a specified level of) tracing[24] is an excellent example. (If correct behavior is defined by a *not null* value and especially if the conditional compilation is wrapped, there's always a risk that an accidental *alter... compile* command that omits *reuse settings* will subvert the intended behavior.) Therefore, using an *inquiry directive* is a perfect fit for isolating diagnostic code that will be used deliberately in each successive revision of the code in the development shop but which will be used only as a last resort in the production environment. The following use cases illustrate this paradigm: *Latent self-tracing code* on page 23; *Latent assertions* on page 24; *Unit testing of subprograms declared only in a package body* on page 27; *Mock objects* on page 34; and *Comparing competing implementations during prototyping* on page 37.[25]

---

23. A *ccflag* is — perhaps surprisingly — not of a datatype *per se*. However, the syntax check for the value of the *PLSQL_CCFlags* parameter ensures the text that each *ccflag* specifies is either *true*, *false*, *null*, or represents a legal *pls_integer* literal. Conditional compilation replaces each *inquiry directive* with the literal that it denotes; then this turns out to be legal or not — in the subsequent compilation stages — according to its context.

24. Conceivably, tracing may be turned on as an emergency diagnostic measure if a bug manifests in the production environment. This would be done manually under the guidance of a support engineer.

We recommend controlling conditional compilation with a static package constant in the opposite situation: when two or more different conditionalizations are expected routinely to be used in the production environment. The nature of the requirement that drives this will typically mean that two or more compilation units are controlled by the same static package constant; of course, a consistent response to conditionalization must be guaranteed across all these units. The requirement will also typically imply that the conditionalization should be triggered by an event — for example, licensing and installing a new software component — and that the correct response should be guaranteed without manual intervention. The following use cases illustrate this paradigm: *Component based installation* on page 40 and *Spanning different releases of Oracle Database with a single source code corpus* on page 46.

The *DBMS_DB_Version* package

The *DBMS_DB_Version* is supplied with Oracle Database. It exposes static package constants that specify the current version and release[26]. The source is listed in *Appendix C: The source code of the DBMS_DB_Version package in 10.2, 10.1, and 9.2* on page 69[27]. Programmers should test the *DBMS_DB_Version* constants in *selection directives* that guard code that is legal only in a new release of Oracle Database and that will therefore fail to compile in an earlier release.

This package is valuable precisely because it is *supplied* and because Oracle Corporation, therefore, will ensure that its constants will differ appropriately from release to release. It will therefore enable the *automatic* selection of the appropriate release-specific code. The benefit is obviously felt when release-conditionalized code is newly installed. A moment's thought shows that the benefit is felt also when release-conditionalized code is part of a deployed system and when the populated Oracle Database is upgraded to a new release[28].

Suppose that a compilation unit *U* containing a *selection directive* that tests a *DBMS_DB_Version* constant is deployed in production in, for example, a *Release N* environment. And suppose that an under-the-feet upgrade is made to *Release N+1*. Then — because the compilation unit depends on the *DBMS_DB_Version* package and because the upgrade will recompile this package to reflect the new release — the compilation unit *U* will be invalidated. Therefore, when *U* is next used, it will be recompiled and will therefore start *automagically*[29] to use the (presumably) newer and more efficient implementation

25. The same effect could be achieved by using a package specification that exposes a single static constant and by shipping it wrapped. This approach, though, is decidedly less convenient — and would be compromised if a future release of Oracle Database exposed information about package variables in metadata.

26. *DBMS_DB_Version* has no package body and the package specification exposes *only* these static package constants.

27. See also the section *The availability of PL/SQL conditional compilation in Oracle Database 10g Release 1 and in Oracle9i Database Release 2* on page 59.

28. We understand from several ISVs and customers with mission critical applications developed in house that such upgrading "under the feet" of a deployed system is common practice.

29. See *en.wiktionary.org/wiki/automagic*: *...done automatically in such a clever way that the result looks like magic... It is drawn from the adage (often called Clarke's third law) that any sufficiently-advanced technology is indistinguishable from magic.*

that has been provided in a previously unselected leg of the *selection directive* "compound statement".

A code example is given later (see *Spanning different releases of Oracle Database with a single source code corpus* on page 46).

HOW DOES PL/SQL CONDITIONAL COMPILATION WORK?

The section explains how conditional compilation fits into the PL/SQL compilation pipeline and how the programmer can inspect the output of this new stage. It also addresses what might seem to be a surprising question: is there really much difference between the effect of compile-time *$if* construct and that of the run-time *if* construct when the object of the test is a static *boolean* expression? (The question arises because the optimizing PL/SQL compiler can recognize when the outcome of the test for a run-time *if* construct is known at compile time and can eliminate the code that will not be executed.)

The PL/SQL compilation pipeline

PL/SQL compilation proceeds in distinct stages. Each stage completes before the next begins. Conditional compilation introduces a new early stage. This stage intercepts the new PL/SQL conditional compilation constructs (the *inquiry directive*, the *selection directive*, and the *error directive*). The new constructs rely on these building blocks[30,31]:

```
$if $then $elsif $else $end $error $$<any_simple_identifier>
```

The stage either replaces the PL/SQL conditional compilation constructs with fragments of regular PL/SQL or drops them completely. The rest of the compilation process then proceeds as if the PL/SQL conditional compilation feature did not exist[32].

Using the *DBMS_Preprocessor* package to see the conditional compilation output

The *DBMS_Preprocessor* package satisfies this requirement from the functional specification for PL/SQL conditional compilation:

> The post-processed source should be available for inspection except for wrapped sources. (This requirement asks that the text the compiler actually processes after PL/SQL conditional compilation should also be visible. This is a crucial aid to debugging.)

It has two modes of operation:

- It can retrieve the source text of an extant compilation unit — denoted by *schema*, *type*, and *name* — and process it using the value of *PLSQL_CCFlags* that was stored with the compilation unit (exposed by the *All_PLSQL_Object_Settings* view family).
- It can take ephemeral source text — which exists only in a PL/SQL variable in the program that calls the *DBMS_Preprocessor* subprogram — and process it

---

30. Notice that the new building blocks would be illegal in PL/SQL source before the advent of PL/SQL conditional compilation. Source text containing any of these would cause the rather uninformative compilation error *PLS-00103: Encountered the symbol "$" when expecting one of the following.*

31. Here, <*any_simple_identifier*> stands for any identifier that would be legal in a regular PL/SQL context (for example, as the name of a variable) without using double quotes.

32. The risk analysis that supported the decision to make this feature available in *10.1.0.4* and in *9.2.0.6* rests on this understanding.

using the value of *PLSQL_CCFlags* from the current environment. (There are overloads in this mode — corresponding to those for *Dbms_Sql.Parse()* — for "small" and "large" amounts of source text.)

Each mode has two sub-modes:

- to return the post-processed source text into a PL/SQL variable.
- to output the post-processed source text via *DBMS_Output* subprograms.

For an input source text that compiled (or would compile) without error, then the post-processed source text is derived simply by replacing all the structural parts of each *selection directive* and the unselected source text therein with whitespace and by replacing each *inquiry directive* that occurs in selected regular PL/SQL text by the literal that the inquiry produced. The requirement for compilation without error in this statement of the rule means that no occurrences of the *error directive* will survive in the post-processed source.

Suppose that procedure *P*, created with *Code_1* on page 5, is in place and is owned by *Usr*. (Recall that the last action was to compile it with *Use_IEEE:false*.) The SQL*Plus script shown in *Code_15* reveals the current state.

```
-- Code_15
select PLSQL_CCFlags from User_PLSQL_Object_Settings
  where Type = 'PROCEDURE' and Name = 'P'
/
select Text from User_Source
  where Type = 'PROCEDURE' and Name = 'P'
/
```

Code_15 produces the following output:

```
Use_IEEE:1
```

and:

```
procedure P is
  n
    -- To do: implement a variant using binary_double
    $if    $$Use_IEEE = 0 $then number;
    $elsif $$Use_IEEE = 1 $then binary_float;
    $else $error 'Illegal Use_IEEE: '||$$Use_IEEE $end
    $end
begin
  $if $$Use_IEEE = 0 $then
    n := 1.0;
  $else
    n := 1.0f;
  $end
  Print(n);
end P;
```

Notice that the *actual* source text that the programmer submitted using the *create or replace* SQL statement — with its conditional compilation directives in place — is stored in the catalog structures that are exposed by the *All_Source* view family. *Code_16* shows how to display the code that the conditional compilation stage produced and passed on to the rest of the compilation process.

```
-- Code_16
begin
  DBMS_Preprocessor.Print_Post_Processed_Source(
    Schema_Name => 'USR',
    Object_Type => 'PROCEDURE',
    Object_Name => 'P');
end;
```

Code_16 produces the following output:

```
procedure P is
  n
    -- To do: implement a variant using binary_double
                                  binary_float;

begin


    n := 1.0f;

  Print(n);
end P;
```

Code_17 shows how to display the alternative version.

```
-- Code_17
alter procedure P compile
  PLSQL_CCFlags = 'Use_IEEE:0' reuse settings
/
begin
  DBMS_Preprocessor.Print_Post_Processed_Source(
    Schema_Name => 'USR',
    Object_Type => 'PROCEDURE',
    Object_Name => 'P');
end;
/
```

Code_17 produces the following output:

```
procedure P is
  n
    -- To do: implement a variant using binary_double
                                  number;


begin

    n := 1.0;


  Print(n);
end P;
```

The abundance of whitespace may seem surprising. It is the result of the straightforward specification of the behavior of conditional compilation (when the output it produces compiles without error):

- it replaces each construct *$if*, *$then*, *$elsif*, *$else*, and *$end* with the number of spaces that the construct occupies;
- it replaces the whole of the static *boolean* expression with a corresponding number of spaces;
- it replaces unselected source text items with a corresponding number of spaces — so even unselected comments are replaced with whitespace;
- it leaves selected source text items at (almost) their original locations;
- it replaces each *inquiry directive* that survives in selected source text with the text it denotes[33] and surrounds this text with one leading and one trailing

whitespace[34]. (This is the reason for the caveat "almost" in the preceding bullet.)

The purpose of these rules is easily explained. PL/SQL compilation errors in the surviving source text that the rest of the compilation process sees will be reported — in the normal way — against line and column numbers in that text. These must correspond as closely as possible to what the programmer wrote. (As mentioned, this text is exposed exactly as the programmer wrote it by the *All_Source* view family.)

For an input source text that compiled (or would compile) with errors, then the post-processed source text is typically only partly produced and the text of the compilation error messages is appended. *Code_18* illustrates this.

```
-- Code_18
alter procedure P compile
  PLSQL_CCFlags = 'Use_IEEE:2' reuse settings
/
begin
  DBMS_Preprocessor.Print_Post_Processed_Source(
    Schema_Name => 'USR',
    Object_Type => 'PROCEDURE',
    Object_Name => 'P');
end;
/
```

Code_18 produces the following output:

```
procedure P is
  n
    -- To do: implement a variant using binary_double


ORA-06550: line 6, column 11:
PLS-00179: $ERROR: Illegal Use_IEEE: 2
```

Choosing between the compile-time *$if* construct and the run-time *if* construct

Oracle Database 10*g* Release 1 introduced the optimizing PL/SQL compiler. This radical change with respect to the previous releases of Oracle Database is described in several technical whitepapers published on the Oracle Technical Network.[35]

The compiler is able to recognize when the outcome of the test for a run-time *if* construct is known at compile time and it tries to eliminate, at compile time, the code that will not be executed. When this happens, the performance benefit of the compile-time *$if* construct (smaller and faster run-time code) is delivered also

---

33. As mentioned earlier (see *The error directive* on page 12), a *ccflag* is not of a datatype *per se*.

34. This too may seem surprising. But suppose the *ccflag x* has been defined as *32676*, which uses five character positions. The *inquiry directive $$x* cannot be replaced without upsetting the length of the source text line where it occurs. And if the surrounding spaces were not added, subsequent PL/SQL compilation errors could occur in certain corner cases.

35. Start here: *www.oracle.com/technology/tech/pl_sql/htdocs/new_in_10gr1.htm#faster*

by the run-time *if* construct. The SQL*Plus script shown in *Code_19* demonstrates this.

```
-- Code_19
alter session set PLSQL_Optimize_Level = 2
/
alter session set PLSQL_Warnings = 'enable:all'
/
create or replace package CC_Control is
  Tracing constant boolean := true;
end CC_Control;
/
create or replace procedure P is
begin
  Print('P');
  $if CC_Control.Tracing $then
--if CC_Control.Tracing then
    Print('The first of 1000 such lines');
    ...
    Print('The last line.');
  $end
--end if;
end P;
/
SHOW ERRORS
select Source_Size, Code_Size
  from User_Object_Size
  where Name = 'P' and Type = 'PROCEDURE'
/
```

The 1,000 *Print()* statements are used as a simple device to simulate voluminous source code that implements tracing. The alternative approaches — using a compile-time *$if* construct, as shown, or using a run-time *if* construct by uncommenting it and by commenting out the compile-time *$if* construct — are deliberately both in place so that toggling between the two will cause no change in the observed value of *Source_Size*. The script is run four times to cover all combinations of *CC_Control.Debug* (*true* and *false*) and of the kind of the *if* test (*compile-time* and *run time*). I recorded the following results:

```
Compile-time IF & Tracing=true
      68006      52575

Compile-time IF & Tracing=false
      68006        298

Run-time IF     & Tracing=true
      68006      52575

Run-time IF     & Tracing=false
      68006        298
```

In this test, the optimizing compiler succeeded in eliminating the code that it could prove would never be executed. (With warnings turned on as shown, the *SHOW ERRORS* SQL*Plus command showed *PLW-06002: Unreachable code*.) If the test is repeated using the run-time *if* construct with just *Tracing boolean := false;* (without the *constant* keyword), then the bigger value of *Source_Size* is seen. Of course, here the optimizer cannot prove that *CC_Control.Tracing* will never be *true*. For the same reason, the attempt to compile P using the compile-time *$if* construct fails when *CC_Control.Tracing* is not declared as a *constant*.

Notice, though, that the run-time *if* construct carries no guarantee to detect and to eliminate unreachable code — notwithstanding the fact that the benefit is *usually* delivered. (For example, if *Code_19* is run in an environment where *PLSQL_Optimize_Level = 0*, then the unreachable code is not eliminated and the warning *PLW-06002: Unreachable code* is not given.) The compile-time *$if*

construct is guaranteed — by the definition of its syntax and semantics — to deliver its benefit. This difference is crucial.

The effort of analyzing an extant code corpus and replacing every run-time *if* construct that tests a static *boolean* expression with the corresponding compile-time *$if* construct is very unlikely to deliver a measurable performance improvement. Nevertheless, if time does allow a deliberate review of extant code, then this is one excellent check-list item: use the *constant* keyword in the declaration of every variable that the code does not change. The worst that can happen if *constant* is used inappropriately is a compilation error; and conversely, if using *constant* does not cause a compilation error, its use can never be harmful — and might be beneficial.

However, in new projects, we strongly recommend a conscious choice. If you know that the condition you are testing can change only by deliberately editing the source text — as is the case with *static constants* — then the compile-time *$if* construct is overwhelmingly the best choice. PL/SQL now provides syntax to let you express that knowledge explicitly. Be sure to make a conscious choice between using a static package constant or using an *inquiry directive* in the *selection directive*'s boolean expression (see *Choosing between an inquiry directive and a static package constant* on page 14)[36].

36. A run-time *if* construct that tests an expression composed only of *inquiry directives* and literals should cause concern. If the expression properly expresses the intended test, then the compile-time *$if* construct should be used instead.

PL/SQL CONDITIONAL COMPILATION USE CASES

This section presents a series of use cases. For each use case, first the problem is stated and then the use of PL/SQL conditional compilation to solve the problem is illustrated. For some of the use cases, conditional compilation allows a better solution than was possible before its advent. For others it allows a solution where previously none existed. The uses that are easiest to understand are discussed first. The order of presentation is unrelated to the potential value.

Latent self-tracing code

The problem

Programmers often need to trace the execution of their PL/SQL compilation units. They can choose between two radically different approaches to do this.

- They can adorn their code with a multitude of "print" statements (typically using the *DBMS_Output* package or the *Utl_File* package) to say "I got here" and to display selected values that characterize the current state of the world.
- They can leave their code untouched and use an interactive debugger[37].

Of course, they can combine the approaches — as most programmers typically do. A notable advantage of the "print" approach is that is can produce machine readable output that can be archived and used in mechanical regression testing. I draw on a further advantage in my case study (see *Case study: implementing unit testing, assertions, and tracing for a fast cube root body-private helper function* on page 49): it can show the evolution of a value of interest in a single report. The disadvantage of the "print" approach has been that it causes a dilemma when the code is deployed in production and when the tracing needs to be suppressed. It is uncomfortable to delete all the tracing code because — sadly, but realistically — it might be needed to diagnose bugs that manifest first in the production environment; and it is uncomfortable to surround all the tracing code with run-time *if* constructs. Performance considerations aside, this second approach might require static reference to objects (for example, helper subprograms and schema-level tables or directory objects for logging) that should not be present in the production environment.

37. Oracle9*i* Database Release 2 introduced server-side support for interactive debugging of PL/SQL subprograms using the industry-standard JDWP protocol. At the same time, JDeveloper introduced a graphical interface to exploit this. You can set and remove breakpoints, step into or step over subprograms, and display the values of all the variables in scope when execution is paused at a breakpoint. This is especially labor saving when the variable is not scalar (for example, a collection of collections of records); writing the "print" code to show the same information would take some effort.

Using PL/SQL conditional compilation to solve the problem

Code_20 shows the typical idiom for effectively removing tracing code from the production environment while leaving it in place, latent but ready to be turned back on when needed.

```
-- Code_20
$if $$tracing $then
  declare Idx Idx_t := Sparse_Collection.First();
  begin
    while Idx is not null loop
      Print(Sparse_Collection(Idx));
      Idx := Sparse_Collection.Next(Idx);
    end loop;
  end;
$end
```

In this example, the extra declaration that is needed to support the tracing can be made within the one region of selected text. However, sometimes programmers find that a dedicated procedure is useful to format trace output in the same way at many different tracing sites. Notice that conditional compilation allows the definition of such a helper procedure to be compiled only when *$$tracing* is true. Even if the optimizer manages to eliminate code that will never execute and then — presumably in a second pass — to eliminate the definitions of subprograms that are never called, the difference between the run-time *if* construct and the compile-time *$if* construct is very significant: the latter allows the programmer explicitly to say what is intended: that under specific circumstances, specific subprograms should not become part of the run-time code. Real examples of this are shown later (see *Case study: implementing unit testing, assertions, and tracing for a fast cube root body-private helper function* on page 49).

Latent assertions

The problem

The notion of an assertion is well established in the general discussion of programming practice; it is programming language agnostic. Wikipedia gives a very clear account of the subject[38]. Here are some extracts from the article.

> "...an assertion failure ...indicates a possible bug in the program."
>
> "...an assertion failure ...[usually] halts the program's execution immediately."
>
> "Programmers add assertions to the source code as part of the development process. They are intended to simplify debugging and to make potential errors easier to find."
>
> "Assertions can also be a form of documentation: they can describe the state the code expects to find before it runs (its preconditions), and the state the code expects to result in when it is finished running (postconditions). Assertions are also sometimes placed at points the execution is not supposed to reach."
>
> "The advantage of using assertions rather than comments is that assertions are checked for validity every time the program is run; if the assertion no longer holds, the programmer will be notified. This prevents the code from getting out of sync with the assertions (a problem that can occur with comments)."

38. See *en.wikipedia.org/wiki/Assertion_(computing)*

> "Assertions should be used to document logically impossible situations — if the 'impossible' occurs, then something fundamental is clearly wrong. This is distinct from error handling."
>
> "Assertions can be enabled or disabled, usually on a program-wide basis. If assertions are disabled, assertion failures are ignored. Since assertions are primarily a development tool, they are often disabled when the program is released. Because some versions of the program will include assertions and some will not, it is essential that the presence of assertions does not change the meaning of the program. In other words, assertions ought to be free of side effects."
>
> "The removal of assertions from production code is almost always done automatically. It usually is done via conditional compilation."

The point, of course, is that assertions incur a run-time cost. But to remove them all after system testing is done and before the application goes live would compromise the ability of the debugging team to diagnose bugs first seen in the production environment. This is why the last bullet mentions conditional compilation. PL/SQL programmers can now implement assertions so that they deliver their intended benefit without incurring a cost in production. An old best practice is now newly available in PL/SQL. Here is another extract from the Wikipedia article:

> "Some people, however, object to the removal of assertions by citing an analogy that the execution of a program with assertions in development stage and without [them in production] is like practicing swimming in a pool with a lifeguard and then going swimming in the sea without a lifeguard."

My view on this is that there are different kinds of assertions. Here is an example:

- On the one hand, a subprogram *Public_Utility()* which is published and documented for general use by programs that are not yet written when *Public_Utility()* is released had better not suffer the removal for production of the assertion that confirms that the actual arguments with which it is called conform to its specification.
- On the other hand, a subprogram *Body_Private_Helper()* which cannot be called except from sites in a single package body and which therefore can be policed in a single unit of editing — the package body's source file — may well lose the corresponding assertion for production.

Using PL/SQL conditional compilation to solve the problem

Suppose that a helper function is needed in a package body to calculate the area of a triangle given the length of each of its sides. The general approach[39] can be simplified because — supposedly — the triangles will always be right-angled[40]. The function had better start with an assertion that the values of the formal

---

39. Pick one of the sides, *s1*; calculate the angle, *theta*, between *s1* and the side, *s2*, at one of its ends; determine the height, *h*, from the *sin* of *theta* and the length of *s2*; compute the area by halving the product of *h* and the length of *s1*.

40. Find the two shortest sides; compute the area by halving their product.

parameters do indeed specify a right-angled triangle. The simplest version of the guard for the assertion looks like this:

Code_21[41] shows a simple approach to implementing the assertion.

```
-- Code_21
function Area(
  s1 in Size_t, s2 in Size_t, s3 in Size_t)
  return Size_t
is
  epsilon  constant Size_t := 0.000001;
  hyp      constant Size_t := Greatest(s1, s2, s3);
  a        constant Size_t := Least(s1, s2, s3);
  b        constant Size_t := case
                                when hyp=s1 and a=s2 then s3
                                when hyp=s1 and a=s3 then s2
                                when hyp=s2 and a=s1 then s3
                                when hyp=s2 and a=s3 then s1
                                when hyp=s3 and a=s1 then s2
                                when hyp=s3 and a=s2 then s1
                              end;
begin
  $if $$Asserting $then
    if not Abs(hyp*hyp - (a*a + b*b)) < epsilon then
      Raise_Application_Error(-20000,
        'hyp*hyp='||hyp*hyp||'; a*a + b*b = '||(a*a + b*b));
    end if;
  $end
  return 0.5*a*b;
end Area;
```

Suppose that an ISV ships an application that includes many package bodies[42]. And suppose that each package body has a number of assertions that are guarded by a *selection directive*. It is likely that a single switch would be preferred to turn on or turn off the assertions for all the compilation units that implement the application. In that case, it would be useful for the *selection directive* that guards each assertion to test the same static package constant — for example, *Assertion_CC.Asserting*. The installation scripts would include two alternative scripts to create the *Assertion_CC* package: one would set *Asserting* to *false* for normal use; and the other would set it to *true* for exceptional use. It would be sensible to obfuscate these two scripts and the compilation units they control.

Notice that the two approaches — using an *inquiry directive* or using a static package constant — can be combined. *Code_22* shows how the assertion shown in *Code_21* could be guarded.

```
-- Code_22
$if $$Asserting or Assertion_CC.Asserting $then
  if not Abs(hyp*hyp - (a*a + b*b)) < epsilon then
  ...
$end
```

The *inquiry directive* would be used to turn the assertions on or off for an individual compilation unit; and the static package constant would be used to turn them on or off for the whole application.

---

41. You may wonder why I do not simply use *Heron's formula*.The reason is simple: I had forgotten about it until a colleague reminded me. I beleive that — with some indulgence — my example still works well.

42. The same considerations apply when an application that is developed in house is deployed for production.

Unit testing of subprograms declared only in a package body

The problem

Packages usually have subprograms that are defined only in the body. They are deliberately not exposed in the package specification because they have no meaning except as specific helpers for the implementation in the package body. Many development shops require that *every* subprogram — and not just those exposed by the package specification — be tested *explicitly*; they hold that the implicit testing that body-private subprograms would receive — by limiting formal unit testing to just the exposed subprograms that call them — is insufficient.

How, then, can a unit test be written for such body-private subprograms?

The naïve approach is to declare every top-level body-subprogram in the package specification and to implement the testing in subprograms that are defined outside of the package. We know of customers who have done this — and they report difficulty in policing the ordinary use of the would-be private subprograms from other compilation units[43]. This has been particularly troublesome when the subprogram seems to be generically usable but when its design rests on the assumption of simplifying invariants that can be guaranteed only when all invocations are from within the package itself.

The alternatives, before the advent of PL/SQL conditional compilation, all have drawbacks.

(1) *"Wrapper" package and tightly disciplined ownership rules.* The "real" package is created in a schema dedicated to hold just that package and the compilation units that implement the unit tests. Every package body subprogram is declared in the package specification. A wrapper package is written to expose only those subprograms that the outside world should see and the *execute* privilege is granted on just the wrapper to other users. The SQL*Plus

43. Apparently, *"if you can describe it, then you can use it"* trumps naming conventions, comments, or external documentation.

script shown in *Code_23* sketches the approach. In this example, *Pkg_* is the "real" package and *Pkg* is the wrapper.

```
-- Code_23
create package Usr.Pkg_ is
  procedure P1(...);
  procedure P2(...);
  ...
  procedure Helper1(...);
  procedure Helper2(...);
  ...
end Pkg_;
/
create package body Usr.Pkg_ is
  ...
end Pkg_;
/
create package Usr.Pkg is
  -- Notice that this exposes no helpers.
  procedure P1(...);
  procedure P2(...);
  ...
end Pkg;
/
-- List the intended clients.
grant execute on Usr.Pkg to ...
/
create package body Usr.Pkg is
  procedure P1(...) is begin Pkg_.P1(...); end P1;
  procedure P2(...) is begin Pkg_.P2(...); end P2;
  ...
end Pkg;
/
create procedure Usr.Run_The_Tests(...) is
begin
  Pkg_.P1(...);
  Pkg_.P2(...);
  ...
  Pkg_.Helper1(...);
  Pkg_.Helper2(...);
  ...
end Run_The_Tests;
/
begin Usr.Run_The_Tests(...); end;
/
```

The convention that this approach rests on would require some effort to document and some discipline to enforce[44].

44. Of course, there would be a performance penalty too from the extra subprogram invocation. A feature planned for next major release of Oracle Database after *10.2*, intra-unit and inter-unit inlining, may remove this concern.

(2) *Implement the unit testing inside the package body itself* and expose this by a single *Run_The_Tests()* procedure. The SQL*Plus script shown in *Code_24* sketches the approach.

```
-- Code_24
create package Pkg is
  procedure P1(...);
  procedure P2(...);
  ...
  procedure Run_The_Tests(...);
end Pkg;
/
create package body Pkg is
  procedure P1(...)      is ... end P1;
  procedure P2(...)      is ... end P1;
  ...
  procedure Helper1(...) is ... end Helper1;
  procedure Helper2(...) is ... end Helper2;
  ...

  procedure Run_The_Tests(...) is
  begin
    P1(...);
    P2(...);
    ...
    Helper1(...);
    Helper2(...);
    ...
  end Run_The_Tests;
end Pkg;
/
begin Pkg.Run_The_Tests(...); end;
/
```

This approach has the disadvantage that, in the production environment, packages would be rather bigger than their purpose required and that the package specification would expose a mysterious testing procedure. There might be other problems to solve if the tests require various schema objects to be in place that would not be present in the production environment.

Using PL/SQL conditional compilation to solve the problem

"I described the challenge we faced with unit testing our package-body-private subprograms to Bryn before I knew that PL/SQL conditional compilation was on the way. Testing is critical to us and I'm excited to see the new possibilities that this feature gives us — both for ordinary unit testing and for the PL/SQL equivalent of mock objects."

— Nick Strange
Principal Architect
Fidelity Brokerage Company Technology
www.fidelity.com

PL/SQL conditional compilation can be used to support each of these two approaches (selective exposure of body-only subprograms and encapsulation of the tests inside the package body) and to overcome the disadvantages that each — as so far described — has.

(1) *Conditional exposure of the helper subprograms.* The most obvious way to implement this is to surround the direct declaration of each helper in the package specification with a *selection directive*. This would require that the package specification is wrapped and that the *selection directive* tests a *ccflag* with an unguessable name — relying on the behavior that if such a *ccflag* is not defined, then it evaluates to *null*. However, it is common to rely on the package specification and the comments it contains to document a package and so wrapping could be undesirable. Moreover, compiling and recompiling the package specification to expose and to hide the helper subprograms, even in the development shop, might hurt developer productivity because of

the invalidations this would cause. *Code_25* sketches an approach that avoids these drawbacks.

```
-- Code_25
-- Don't wrap this
create package Pkg is
  procedure P1(...);
  procedure P2(...);
  ...
  procedure Helper1_(...);
  procedure Helper2_(...);
  ...
end;
/
-- Wrap this
create package body Pkg is
  procedure P1(...)      is ... end P1;
  procedure P2(...)      is ... end P2;
  ...
  procedure Helper1(...) is ... end Helper1;
  procedure Helper2(...) is ... end Helper2;
  ...

  procedure Helper1_(...) is
  begin
    $if $$Testing $then
      Helper1(...);
    $else
      raise Program_Error;
    $end
  end Helper1_;

  procedure Helper2_(...) is
  begin
    $if $$Testing $then
      Helper2(...);
    $else
      raise Program_Error;
    $end
  end Helper2_;
  ...
end Pkg;
/
create procedure Usr.Run_The_Tests(...) is
begin
  Pkg.P1(...);
  Pkg.P2(...);
  ...
  Pkg.Helper1_(...);
  Pkg.Helper2_(...);
  ...
end Run_The_Tests;
/
alter package Pkg compile body
  PLSQL_CCFlags = 'Testing:true' reuse settings
/
begin Usr.Run_The_Tests(...); end;
/
```

Notice how *Code_25* is derived from *Code_23*. It would still take some effort to keep the list of formal parameters for each subprogram like *Helper1_()* in step with that of the corresponding *Helper1()*. And it might be appropriate to use obscure names for identifiers like *Helper1_*. (Presumably, a real example would use *Raise_Application_Error()* with a useful error message.) Nevertheless, the approach is substantially easier to maintain than the approach — without conditional compilation — that inspired it.

(2) *Implement the unit testing inside the package body itself* and expose this by a single *Run_The_Tests()* procedure. The SQL*Plus script shown in *Code_26* sketches the approach.

```
-- Code_26
-- Don't wrap this
create package Pkg is
  procedure P1(...);
  procedure P2(...);
  ...
  procedure Run_The_Tests(...);
end;
/
-- Wrap this
create package body Pkg is
  procedure P1(...)      is ... end P1;
  procedure P2(...)      is ... end P2;
  ...
  procedure Helper1(...) is ... end Helper1;
  procedure Helper2(...) is ... end Helper2;
  ...

  procedure Run_The_Tests(...) is
  begin
    $if $$Testing $then
      P1(...);
      P2(...);
      ...
      Helper1(...);
      Helper2(...);
      ...
    $else
      raise Program_Error;
    $end
  end Run_The_Tests;
end Pkg;
/
alter package Pkg compile body
  PLSQL_CCFlags = 'Testing:true' reuse settings
/
begin Pkg.Run_The_Tests(...); end;
/
```

Notice how *Code_26* is derived from *Code_24*. Again, it might be appropriate to use an obscure name for *Run_The_Tests()*. Notice that all of the code responsible for the unit testing is removed for production; yet, by keeping it in the same editing unit as the subprograms it tests, it provides an excellent form of documentation for the programmer — and especially for someone other than the author who later might need to maintain the code. This technique is illustrated in a working example later (see *Case study: implementing unit testing, assertions, and tracing for a fast cube root body-private helper function* on page 49).

The choice between conditionally exposing the helper subprograms in order to write the tests outside the package and writing the tests inside the package itself will possibly depend on the scale of the project and on the number of developers involved. And different development shops will probably have different preferences. The second approach does have some practical advantages.

- When the tests are written inside the package body, they can manipulate package globals that are declared only there. This would allow a simple, direct way to set up the preconditions that the specification for a particular test might require.

- Inner subprograms, at an arbitrarily deep level of nesting, can be unit tested. The technique may not be immediately obvious.

(3) *Unit testing of deeply nested inner subprograms.* The essential point is that an inner subprogram is not visible in an outer scope; therefore, its unit test must be written at a deeply nested site — guarded, of course, by a *selection directive* — and a conditionally compiled mechanism must provide a path to invoke the test from outside of the package. *Code_27* shows how the plan starts[45].

```
-- Code_27
procedure Inner_2(...) is
begin
  ...
end Inner_2;

$if $$Testing $then
  procedure Test_Inner_2(...) is
  begin
    ...Set up the conditions to call Inner_2
    Inner_2(...);
    ...Check that Inner_2 behaved according to spec
  end Test_Inner_2;
$end
```

Of course, as it name suggests, *Inner_2()* is nested inside *Inner_1()* as *Code_28* shows.

```
-- Code_28
procedure Inner_1(...) is

  procedure Inner_2(...) is ... Inner_2;

  $if $$Testing $then
    procedure Test_Inner_2(...) is ... Test_Inner_2;
  $end

begin
  $if $$Testing $then
    Test_Inner_2(...);

  $else
    ...Normal operations of Inner_1()
    Inner_2(...);
    ...More normal operations of Inner_1()
  $end
end Inner_1;
```

Notice that *Inner_1()* has two alternative bodies: the normal one that its requirements specification determines and a trivial one that merely invokes the test of its inner subprogram — or of *all* its inner subprograms, should there be several.

45. I prefer to encapsulate the unit test as a subprogram. This both is self-documenting and provides an obvious "granule" for conditionalization.

Of course — again, as *its* name suggests — *Inner_1()* is further nested in *Helper()* as *Code_29* shows.

```
-- Code_29
procedure Helper(...) is

  procedure Inner_1(...) ... end Inner_1;

begin
  $if $$Testing $then
    Inner_1(...);

  $else
    ...Normal operations of Helper()
    Inner_1(...);
    ...More normal operations of Inner_1()
  $end
end Helper;
```

Notice that the invocation of *Inner_1()* when *Testing* is *true* does not test *Inner_1()*; rather, it follows the prepared invocation route to call *Test_Inner_2()*.

To keep this example tractable, *Helper()* is a top-level, body-private subprogram. *Code_30* shows the last step in the invocation route to the outside world.

```
-- Code_30
package body Pkg is
  procedure Helper(...) is ... end Helper;

  procedure Main(...) ... Main;

  procedure Run_The_Tests(...) is
  begin
    $if $$Testing $then
      Helper(...);
    $else
      Raise_Application_Error(-20000, 'Testing diabled');
    $end
  end Run_The_Tests;
end Pkg;
```

This might seem to be rather complex. But consider the alternatives: either, *(a)*, the ambition level to unit test inner subprograms is simply abandoned in favor of the assertion that they receive adequate testing implicitly as a consequence of the unit testing of the top-level subprograms that depend on the inner helpers; or, *(b)*, the source of the to-be-tested inner subprogram is copied — temporarily — to establish it at top level in the package body; or, *(c)*, abandon the attempt to use inner subprograms and use only subprograms that are declared at top level in a package.

The disadvantage of *(a)* is obvious; you may as well say that *all* subprograms receive sufficient testing in normal use and that unit testing is unnecessary.

The disadvantage of *(b)* is that it is not in general viable without a great deal of re-architecture: an inner subprogram often depends on items — both variables and subprograms — that are visible within its own body but not at the next outer scope. The quantity of temporary source code relocation that is needed to make the test possible will be so great that the test it enables is no longer a reliable test of what should be tested.

> The disadvantage of *(c)* will seem trivial to those who are not convinced by "theoretical" software engineering best practices. They will need to invent distinguishable — and therefore often uncomfortably long — identifiers. Sometimes this will be necessary to avoid compilation error. Sometimes it will result from "moral" pressure (you can hardly name a subprogram *Check_Inputs()* when it is declared very far textually from its use, even when this name would be unique). Sometimes, you will "reuse" a global variable — safely, but confusingly for one who later must maintain the code — because it has the right name and datatype. And sometimes, you will reuse a global variable dangerously and descend into side-effect hell.

This has been a rather lengthy section. My aim has been to explain a number of techniques and to discuss their properties. I recognize that there is no universally applicable best approach. But I am convinced that knowing about these techniques and understanding their properties will enable developers to design better approaches to unit testing than were possible before the advent of PL/SQL conditional compilation.

Mock objects

The problem

The term *mock object* is borrowed in this paper from object-oriented programming in general — and from Java development in particular — as a mnemonic for the use case described in this section. It denotes a paradigm for unit testing. Again, Wikipedia has a useful account of the topic[46]. Here is a short extract.

> "In tests, a mock object behaves exactly like a real object with one crucial difference: the programmer will hard-code return values for its methods..."

Consider the following subprogram dependency in a PL/SQL application.

- The function *Pkg.Customer()* takes the unique identifier of a customer and returns a record representing the customer's information. Assume that the record has a complex structure; for example, one of its fields might be a collection of object types that represent purchases that have been made and are pending.
- The procedure *Pkg.Process_Customer()* iterates over a list of customer identifiers and for each it invokes *Pkg.Customer()* and analyzes the return records to look for specified patterns and to take appropriate action. For example, it might test if the set of purchase records includes more than a specified number of novels written by one of a group of authors; a customer for whom this test is positive should receive an email announcing the availability of a new novel by one of the authors in the group — provided that this novel has not already been purchased.
- The function *Pkg.Customer()* can raise some documented exceptions. For example, it might raise *No_Data_Found* or *Too_Many_Rows*. Presumably *Pkg.Process_Customer()*, because it takes responsibility for generating a list of valid customer identifiers, would handle *No_Data_Found* with an assertion.

46. See *en.wikipedia.org/wiki/Mock_Object*

Presumably, too, it would regard *Too_Many_Rows* as unexpected — but not logically impossible. (This would be an indication of corrupt data stemming, probably, from the accidental dropping a of a unique index.) It would probably handle this by logging the occurrence of the problem and continuing with the processing of the remaining customer records.

How is the unit testing for *Pkg.Process_Customer()* to be orchestrated? It must be fed, in response to successive invocations of *Pkg.Customer()*, at least one record whose characteristics trigger each of the specified *Pkg.Process_Customer()* actions — not only the routine actions, but the special actions in response to exceptions.

It can be tricky and time-consuming to contrive the persistent data in the test environment so that *Pkg.Customer()* will return the required representative set of records.

Using PL/SQL conditional compilation to solve the problem

Conditional compilation lets developers meet the same unit testing goal cheaply and without uncertainty by conditionally including code in the *Pkg.Customer()* that replaces its normal implementation and returns a set of hard-coded customer

records with exactly the characteristics required by the unit test specification for *Pkg.Process_Customer()*. *Code_31* sketches the approach.

```
-- Code_31
package body Pkg is
  type Customer_t is record(...);

  $if $$Mocking_Customer $then
    Counter pls_integer := 0;
  $end

  function Customer(Id in integer) return Customer_t is

    $if $$Mocking_Customer $then
      function Mock_1(Id in integer) return Customer_t is
        This_Customer Customer_t;
      begin
        ... Mock up the first case
        return This_Customer;
      end Mock_1;

      function Mock_2(Id in integer) return Customer_t is
        This_Customer Customer_t;
      begin
        ... Mock up the second case
        return This_Customer;
      end Mock_2;
      ...
    $end

  begin
    $if $$Mocking_Customer $then
      Counter := Counter + 1;
      case Counter
        when 1 then return Mock_1(Id);
        when 2 then return Mock_2(Id);
        ...
        when n then raise No_Data_Found;
        else        raise Too_Many_Rows;
      end case;
    $else
      declare This_Customer Customer_t;
      begin
        ... The "real" (non-mock) code to retrieve
        ... this customer from the persistent data.
        return This_Customer;
      end;
    $end
  end Customer;

  procedure Process_Customer is
    type Customer_Ids_t is table of integer
                             index by pls_integer;
    Customer_Ids Customer_Ids_t;
    This_Customer Customer_t;
    function Valid_Customer_Ids return Customer_Ids_t is
      Ids Customer_Ids_t;
    begin
      ...
      return Ids;
    end Valid_Customer_Ids;
  begin
    Customer_Ids := Valid_Customer_Ids();
    for j in 1..Customer_Ids.Last() loop
      This_Customer := Customer(Customer_Ids(j));
      ... Process this customer
    end loop;
  end Process_Customer;
end Pkg;
```

Some of my colleagues in Oracle's PL/SQL Team think that the compile-time *$if* construct has a significantly different *feel* than does its run-time cousin. I prefer

to see the compile-time *$if* construct and the run-time *if* construct as not nearly so different as they seem[47] to be on first consideration.

Some would prefer to write *$if $$Mocking_Customer $then* only once in the example and to have two completely separate regions of code — one for when *Mocking_Customer* is *true*, for one for when it is not. At the very least, that approach would require that the text *function Customer(Id in integer) return Customer_t is* and the matching text *end Customer* be repeated. Possibly other code, not shown in my example, would need to be repeated. I was taught that such repetition is an evil to be avoided at all costs. Others argue against a proliferation of tests of the same condition. You will have to resolve this aesthetic difference for yourself — and different cases may lead to different resolutions.

Comparing competing implementations during prototyping

The problem

A developer will often realize that two design alternatives will result in the same required behavior. For example, a collection might be implemented as an *index-by-plsql_integer table* or as a *nested table*. Frequently, the best way to choose between these alternatives is to code both and to compare them — for performance, for source code size, and for aesthetic appeal.

Suppose that code in question is created using a single text file and that this file will be the developer's eventual deliverable. Without conditional compilation, the developer will, during this prototyping phase, have two files; each will implement one of the alternatives. Of course, the nature of this scenario means that these competing files will be substantially textually identical but will differ in small critical — but widely distributed — spots. Usually one file is derived from the other by making a series of small edits. It often happens that, after having derived the second file, the need arises to make changes in the code that is common to both. This is especially the case when the competing alternatives need to coexist, while the dedcision on the favorite is pending, but progress must nevertheless be made on the implementation project. This duplication of work is not only time-consuming; even worse, it is error prone.

Using PL/SQL conditional compilation to solve the problem

PL/SQL conditional compilation solves the problem by allowing the competing implementations to coexist in the same text file. Typically the volume of conditional code will be very much smaller than that of the file. This gives you, effectively, an in-place, editable *diff* view.

I mentioned (see *Introduction* on page 2) that it is very hard to design an illustration that is both realistic and short. This is especially true for this use case because the defining characteristic of a real example is that it will have large quantities unconditional code. The scenario I invented for my illustration here will tax your imagination.

Suppose that you need to populate a scalar collection — say, of *varchar2*s — with data that is produced programatically. Then later you need to derive a new

47. Some of my colleagues remind us that compilation can be seen as an optimization — and is not an essential concept for understanding the meaning of a program. In that context, the difference between the compile-time *$if* construct and the run-time *if* construct is that the former can be used where the latter is syntactically illegal: to surround declarations and to interrupt regular statements.

collection as a subset of the starting collection. Both the source collection and the target collection should be of the same datatype so that various helper subprograms can take each as an actual parameter. You realize that there are two approaches to implement the subset derivation.

- You can iterate programatically over the collection elements and test each using a run-time *if* construct. It the test succeeds, then you increment a counter and copy from the current source element to the next target element. For this approach, the preferred collection datatype is the *index-by-plsql_integer table* because it excuses you from the chore of dealing with initialization and extension.
- You can express the test in the *where* clause of a SQL *select* statement that uses the source collection — using the *table* operator — in the *from* list and uses the target collection as the destination for *bulk collect into*. For this approach, the collection datatype must be the *nested table* because it must be declared at schema level so that the *select* statement can understand it[48]. *Code_32* assumes that this type exists at schema level:

```
type Names_Nested_Table_t is table of varchar2(30)
```

48. A proposed project for the next major release of Oracle Database after *10.2* would remove this obstacle and would allow SQL within a PL/SQL program to select from a *index-by-plsql_integer table*.

Code_32 shows how these two alternative approaches can coexist in the same source text.

```
-- Code_32
procedure P is

  $if $$Alt = 1 $then
    type Names_IBI_Tab_t is table of varchar2(30)
      index by pls_integer;
    Source_Rows Names_IBI_Tab_t;
    Target_Rows Names_IBI_Tab_t;
    subtype Rows_t is Names_IBI_Tab_t;
  $elsif $$Alt = 2 $then
    Source_Rows Names_Nested_Tab_t := Names_Nested_Tab_t();
    Target_Rows Names_Nested_Tab_t;
    subtype Rows_t is Names_Nested_Tab_t;
  $else
    $error 'Alt must be 1 or 2' $end
  $end

  procedure Use_The_Data(Rows in Rows_t) is
    n pls_integer := Rows.First();
  begin
    while n is not null loop
      n := Rows.Next(n);
      ...
    end loop;
  end Use_The_Data;
begin
  declare n pls_integer := 0; x varchar2(30);
  begin
    while <some condition> loop
      ... compute a new "x"
      n := n + 1;
      $if $$Alt = 2 $then
        Source_Rows.Extend();
      $end
      Source_Rows(n) := x;
    end loop;
  end;

  Use_The_Data(Source_Rows);

  -- Derive Target_Rows as a subset of Source_Rows.
  $if $$Alt = 1 $then
    declare n pls_integer := 0;
    begin
      for j in 1..Source_Rows.Last() loop
        if Source_Rows(j) like 'X%' then
          n := n + 1;
          Target_Rows(n) := Source_Rows(n);
        end if;
      end loop;
    end;

  $elsif $$Alt = 2 $then
    select Column_Value
      bulk collect into Target_Rows
      from Table(Source_Rows)
      where Column_Value like 'X%';
  $end

  Use_The_Data(Target_Rows);
  Print(Target_Rows.Count()||' from '||Source_Rows.Count());
end P;
```

As presented, there are 28 lines of common code, 14 lines that are selected when *$$Alt* evaluates to 1, and 8 lines that are selected when *$$Alt* evaluates to 2. You must imagine that the elided code — doing something in *Use_The_Data()*, computing <*some condition*>, and *computing a new "x"* — is voluminous. The real program would also contain self-timing code and assertions — both, of course

guarded by *selection directives* — which would increase the common code volume further.

This use case is unique among those discussed in this paper because it is very unlikely that the loser of the two competing alternatives would survive into the deliverable code; its lifetime may well be only a few days — but the effectiveness of PL/SQL conditional compilation in increasing developer productivity is not diminished by this. Of course, therefore, it is natural that the *selection directive* should test an *inquiry directive* rather than a static package constant.

Notice the similarity between this case and the case discussed in the section *Spanning different releases of Oracle Database with a single source code corpus* on page 46. In both cases, large regions of code in the conditionalized compilation unit are likely to be unconditional. And, quite possibly, the same identifier — used in various constructs in the common code — will denote a variable whose datatype is conditionally defined. However, the intent is dramatically different in the two cases: here, the aim is to compare two approaches and reject one; there, the aim is to provide alternative approaches — of indisputable difference in attractiveness — for differently endowed environments. Therefore, here, the *inquiry directive* is preferred; and, there, the static package constant is preferred.

Finally in this section, notice that the characteristics of the scenario described here are very similar to those which are met in bug fixing. Very commonly, in bug fixing, the fix is implemented as several small local changes scattered over a wide area — possibly over several compilation units. It can be useful, during the verification phase of the fix, to toggle quickly between the unfixed and the fixed regimes to repeat old and newly invented tests — not least to ensure that the fix does not cause performance degradation in those tests that run without error in both regimes.

Component based installation

The problem

Oracle ISVs sometimes sell applications which provide optional extra functionality for incremental cost. The modular delivery is implemented by PL/SQL compilation units (and other schema objects and instance data) which are installed, or not, according to what the customer has licensed. The core part of the application must — somehow — be able to invoke these optional components when they are installed and must be viable when they are not installed. However, the core part of the application should not need re-installation, in a modified form, in order to accommodate the installation of a new optional component. (To do so would be to contradict the notion of component based installation.)

Ordinary run-time conditional logic — by taking advantage of configuration data that is maintained by the install scripts for the optional components — can invoke operations supported by an optional component when it is installed and avoid such invocations when it is not. However, PL/SQL's compile-time dependency model prevents the core part of the application from referring statically to elements that are not installed — even if the code that would make such references is never invoked at run time.

Here are three solutions which might be used before the advent of PL/SQL conditional compilation; each has some uncomfortable drawbacks.

(1) *Invoke the optional elements dynamically.* This has two drawbacks. Firstly, the approach puts some strain on the subsystem than handles the parsing and sharing of cursors. And secondly, the formal parameters (including the return value for a function) of a subprogram that is invoked dynamically must be of SQL datatypes; for example, a *boolean* formal is not allowed[49]. In addition, the syntax is more elaborate than that for the equivalent static invocation — and so there is some cost in developer productivity. *Code_33* shows such a dynamic invocation.

```
-- Code_33
declare Answer char(1);
begin
  execute immediate
    'begin :r := Red.X_Exists(:a); end;'
    using out Answer, in a;
  if Answer = 'Y' then
    ...
  end if;
end;
```

Here, and in the following examples in this section, the identifiers *Red*, *Green*, *Blue*, *Yellow*, *Cyan*, and *Magenta* are used to denote optionally installable compilation units. Had it not been necessary to invoke *Red.X_Exists()* dynamically, then it could have been specified to return a *boolean*; the intent of *Code_33* could have been achieved — using one line instead of six — as shown in *Code_34*.

```
-- Code_34
if Red.X_Exists(a) then
  ...
end if;
```

The fact that the dynamic invocation approach sacrifices compile-time checking and dependency creation (it is, of course, precisely this that allows the approach to succeed) may be felt to be too large a price to pay. A further consequence is that a development shop cannot build dependency documentation automatically; this is critical for impact analysis when considering certain kinds of changes during successive development cycles

(2) *Supply the optional components as packages with no bodies.* This approach removes the need to use the dynamic invocation shown in *Code_33* for elements in optional packages. And, provided that the run-time logic ensures that an optional component that is not installed is never invoked, then the missing bodies will never cause an error. A customer might attempt directly to invoke a subprogram in such a package without a body; the result would be *ORA-04067* — which certainly is readily understandable. However, these bodiless "stub" packages would still have a footprint — most conspicuously by the presence of unwanted schemas if the high level architecture has dedicated a separate owner for each component. Further, the stubs would show up in the *All_Objects* view family — and in all the views that reflect the existence of PL/SQL compilation units — and would seem, in response to

---

49. It is possible that a future release of Oracle Database will remove this limitation and allow the dynamic invocation of PL/SQL subprograms that have formal parameters with PL/SQL datatypes.

the SQL*Plus command *Describe*, to be normally viable. All this felt by some ISVs to be an intolerable drawback; rather than installing only the licensed components, the customer would install *all* components — some useful and others deliberately broken yet highly visible.

(3) *Use virtual invocation based on a hierarchy of object types.* A full explanation of this approach would be rather lengthy[50]. Briefly, an optional component that would have been the package *Red* is described instead as a *not final* object type *Red_Super*. And what would have been subprograms in the package are described as *not final member* subprograms in the type. Then the component itself is implemented as the type *Red under Red_Super*. When *Red* is installed, its member subprograms override those of its supertype. The implementation, in the body of *Red*, of the *member* methods would probably be wrappers for subprograms that did the real implementation in the package *Red_Implementation*[51]. This approach is very powerful when there must be two or more distinguishable implementations of an interface[52] in a single database. However, when there must be never more than one implementation in the same database, it adds little to approach *(2)*. (It would, however, solve the dedicated schema problem: the supertypes could be installed in the core schema and the subtypes and partner packages for each component could be installed in their own schemas.) Otherwise, this approach, suffers from all the disadvantages of approach *(2)* — and suffers further because of its added complexity.

Using PL/SQL conditional compilation to solve the problem

PL/SQL conditional compilation allows the references to optional components to be guarded by *selection directives*. The following example shows a scheme where such a *selection directive* tests a static package constant that reflects the presence or absence of a particular component. A PL/SQL procedure — called as part of the installation process for an optional component — recreates the package that exposes these static package constants appropriately.

The core part of the application is modeled by the package *Core* in *Code_35*.

```
-- Code_35
package Core is
  procedure Choose_Action(Choice in varchar2);
end Core;
```

50. I intend to publish a whitepaper on OTN on this single topic.

51. You might fear a performance problem. In fact, especially in Oracle Database 10*g* Release 2 following some improvements in the "plumbing" for virtual invocation of subtype methods, the overhead added by this approach is trivial.

52. The term interface is borrowed from Java and implies here only loosely that language's meaning.

The optional components are modeled by the packages *Red*, *Green*, *Blue*, *Yellow*, *Cyan*, and *Magenta*, each of which exposes the single procedure *Main()*. *Code_36* shows the body of *Red*.

```
-- Code_36
package body Red is
  procedure Main is
  begin
    Print('Red');
  end Main;
end Red;
```

The system also has a component *Mandatory* — with the same shape — which is always installed. *Code_37* shows the body of *Core*.

```
-- Code_37
package body Core is
  procedure Choose_Action(Choice in varchar2) is
  begin
    case InitCap(Choice)
      -- Mandatory is always installed.
      when 'Mandatory' then Mandatory.Main();

      $if Cpts_CC.Red_Installed $then
        when 'Red' then Red.Main();
      $end

      $if Cpts_CC.Blue_Installed $then
        when 'Blue' then Blue.Main();
      $end
      ...
      $if Cpts_CC.Magenta_Installed $then
        when 'Magenta' then Magenta.Main();
      $end
    end case;
  exception when Case_Not_Found then
    Print('Component '||Choice||' is not installed.');
  end Choose_Action;
end Core;
```

As mentioned, the design must include some metadata that specifies which components have been installed. For this example, it is sufficient to let the *User_Objects* view model this. *Code_38* shows the procedure *Maintain_Cpts_CC()* — which is required to recreate the package *Cpts_CC* to reflect the current state of the component installation metadata. *Maintain_Cpts_CC()* depends on the definition of the schema-level type *Object_Names_t*, thus:

```
type Object_Names_t is table of varchar2(30)
```

The package *Cpts_CC*, maintained by *Maintain_Cpts_CC()*, exposes the static package constants controlling the conditionalization of package body *Core*.

```
-- Code_38
procedure Maintain_Cpts_CC is
  Component_Names constant Object_Names_t := Object_Names_t(
    'RED', 'GREEN', 'BLUE', 'YELLOW', 'CYAN', 'MAGENTA');

  subtype Line_t  is varchar2(80);
  type Lines_T    is table of Line_t index by pls_integer;
  Newline         constant char(1) := Chr(10);
  First_Line      constant Line_t :=
    'create or replace package Cpts_CC is';
  Last_Line       constant Line_t := 'end Cpts_CC;';
  Ddl             varchar2(32767) := First_Line||Newline;
begin
  for j in (select Object_Name, 1 Installed
              from User_Objects
              where Object_Type = 'PACKAGE'
              and Status = 'VALID'
              and Object_Name in (
                select Column_Value from
                  Table(Component_Names))
            union
            select c Object_Name, 0 Installed from (
              select Column_Value c
                from Table(Component_Names)
                where Column_Value not in (
                  select Object_Name from User_Objects
                    where Object_Type = 'PACKAGE'
                    and Status = 'VALID')))
  loop
    Ddl := Ddl||'  '||
      Rpad(Initcap(j.Object_Name||'_Installed'), 20)||
      'constant boolean := '||
      case j.Installed
        when 1 then 'true;'
        when 0 then 'false;'
      end||
      Newline;
  end loop;
  Ddl := Ddl||Last_Line||Newline;
  execute immediate Ddl;
end Maintain_Cpts_CC;
```

The possibly daunting *select* statement first restricts the *User_Objects* to those optional components that *have* been installed: this set is used to set the corresponding static package constants to *true*. Then it restricts the list of optional components to those that *have not* been installed: this set is used to set the corresponding static package constants to *false*. When only *Red* is installed, then the *execute immediate Ddl* statement creates the package *Cpts_CC* as shown in *Code_39*[53].

```
-- Code_39
package Cpts_CC is
  Blue_Installed      constant boolean := false;
  Cyan_Installed      constant boolean := false;
  Green_Installed     constant boolean := false;
  Magenta_Installed   constant boolean := false;
  Red_Installed       constant boolean := true;
  Yellow_Installed    constant boolean := false;
end Cpts_CC;
```

53. A real-world implementation would probably use *DBMS_DDL.Create_Wrapped* (new in Oracle Database 10*g*), rather than *execute immediate*; then the *All_Source* view family would expose obfuscated source text for the package *Cpts_CC*.

Suppose that when only *Red* has been installed, the block shown in *Code_40* is run.

```
-- Code_40
begin
  Core.Choose_Action('Mandatory');
  Core.Choose_Action('Red');
  Core.Choose_Action('Blue');
  Core.Choose_Action('Magenta');
end;
```

It will produce this output:

```
Mandatory
Red
Component Blue is not installed.
Component Magenta is not installed.
```

Suppose that then *Blue* and *Magenta* are installed and that — as the last step in the install process the block shown in *Code_41* is run.

```
-- Code_41
begin Maintain_Cpts_CC(); end;
```

The approach can be made more reliable by using a DDL trigger to invoke *Maintain_Cpts_CC()*. A DDL trigger is not allowed to do DDL (except in a small number of restricted ways that do not include create or replace on a PL/SQL compilation unit). The *DBMS_Scheduler* package allows the restriction to be overcome by running a job in a different session to call *Maintain_Cpts_CC()*. *Code_42* shows a procedure that encapsulates the calls to the *DBMS_Scheduler* subprograms.

```
-- Code_42
procedure Submit_Run_Maintain_Cpts_CC is
  -- The DBMS_Scheduler subprograms commit
  -- DML to their metadata tables.
  -- But a trigger must not commit.
  pragma Autonomous_Transaction;
begin
  declare
    Job_Doesnt_Exist exception;
    pragma Exception_Init(Job_Doesnt_Exist, -27475);
  begin
    Sys.DBMS_Scheduler.Drop_Job(
      Job_Name          => 'RUN_MAINTAIN_CPTS_CC',
      Force             => true);
  exception when Job_Doesnt_Exist then null; end;
  Sys.DBMS_Scheduler.Create_Job(
    Job_Name            => 'RUN_MAINTAIN_CPTS_CC',
    Job_Type            => 'STORED_PROCEDURE',
    Job_Action          => 'MAINTAIN_CPTS_CC',
    Number_Of_Arguments => 0,
    Enabled             => false,
    Auto_Drop           => true);
  Sys.DBMS_Scheduler.Run_Job(
    Job_Name            => 'RUN_MAINTAIN_CPTS_CC',
    Use_Current_Session => false);
end Submit_Run_Maintain_Cpts_CC;
```

Code_43 shows the DDL trigger.

```
-- Code_43
trigger On_DML_Trg
  after create or drop or alter on schema
declare
  The_Event constant varchar2(30) := Ora_SysEvent();
  The_Owner constant varchar2(30) := Ora_Dict_Obj_Owner();
  The_Type  constant varchar2(30) := Ora_Dict_Obj_Type();
  The_Name  constant varchar2(30) := Ora_Dict_Obj_Name();
begin
  -- Submit_Run_Maintain_Cpts_CC() will recompile the
  -- Cpts_CC package. That recompilation need not
  -- and must not invoke Submit_Run_Maintain_Cpts_CC(),
  -- else endless recursion.
  if not (The_Owner = 'USR' and
          The_Type  = 'PACKAGE' and
          The_Name  = 'CPTS_CC')
  then
    Submit_Run_Maintain_Cpts_CC();
  end if;
end On_DML_Trg;
```

Notice that *The_Event* is not used in this illustration. However, a real world implementation would probably use more elaborate logic to call *Submit_Run_Maintain_Cpts_CC()* only whena known component is installed or de-installed.

Following the *create or replace* DDL for *Blue* and *Magenta* — and with no further intervention — the package *Cpts_CC* will now have the constants *Blue_Installed* and *Magenta_Installed* set to *true*. Therefore, if the block shown in *Code_40* is run again, it will now produce this output:

```
Mandatory
Red
Blue
Magenta
```

In a real application with such optional components, there would probably be several compilation units that would need to include *selection directives* like those shown in *Code_37* on page 43. Then the automagic response — causing all such compilation units to be recompiled with an appropriate new conditionalization following the installation of new components — would be particularly valuable.

Spanning different releases of Oracle Database with a single source code corpus

The problem

Oracle's Applications Division provided the use case that motivated the introduction of PL/SQL conditional compilation. They wanted their code to be able to span different releases of Oracle Database, using the latest features in the latest release and using a fallback in earlier releases.

Historians of Oracle Database might be interested to know that the introduction of PL/SQL conditional compilation was motivated by a request from Oracle's Applications Division for a solution to the problem described in this section. They face a programming challenge that is very commonly encountered by all Oracle ISVs. They need to support the PL/SQL they deliver not only on the latest release of Oracle Database but also on earlier releases. The argument goes like this.

- Typically, each new release of Oracle Database introduces new functionality in PL/SQL and in SQL along with new syntax for it. Code which takes advantage of a new feature will fail to compile in earlier releases because of the new syntax.

- New features deliver a benefit — usually improving performance and sometimes enabling more compact and reliable programming.
- Oracle's Applications Division, like most ISVs, maintains only a single source code corpus. ISVs typically do not branch derived versions for specific releases of Oracle Database.
- Therefore, PL/SQL code — and the SQL it contains — is written to compile in the *earliest* release of Oracle Database that the ISV supports.
- Therefore, new features — which have been energetically requested by the ISV and which would improve performance and functionality for the ISV's customers — are not taken advantage of until several releases of Oracle Database after their introduction.
- Therefore, customers who *do* use the latest release of Oracle Database are penalized by the procrastination of those who do not.

Using PL/SQL conditional compilation to solve the problem

PL/SQL conditional compilation allows a single source code corpus to be viable in several different releases of Oracle Database so that the code can use the latest features in the latest release and can provide fallback implementations for earlier releases. It manages this precisely because, as was pointed out in the discussion of *Code_3* on page 7, unselected source text need not be compilable.

Oracle Database 10*g* Release 1 introduced new functionality — exposed by the new syntax *indices of* and *values of* as part of the *forall* statement — to allow the collection that specifies the intended DML to be sparse. This functionality was specifically requested by Oracle's Applications Division and other users have welcomed it[54]. It delivers a performance benefit to the user of the application and a productivity benefit to its developer.

54. Oracle's *Ian.Neall* wrote "We are now using sparse arrays a great deal... *indices of* is such a powerful feature and I'm delighted to have this."

Code_44[55] shows how a PL/SQL conditional compilation *selection directive* is used to select the ideal implementation when the release of the Oracle Database supports it and a less attractive fallback when it does not.

```
-- Code_44
$if DBMS_DB_Version.Ver_LE_9_2 $then

  Print('Fallback. Selected in 9.2 and earlier.');
  declare k Index_t := 1; j Index_t := Sparse.First();
  begin
    while j is not null loop
      Dense(k) := Sparse(j);
      j := Sparse.Next(j);
      k := k + 1;
    end loop;
  end;

  forall j in Dense.First..Dense.Last()
    insert into Tbl values Dense(j);

$else

  Print('Ideal. Selected in 10.1 and later.');
  forall j in indices of Sparse
    insert into tbl values Sparse(j);

$end
```

If this fragment is compiled on Oracle9*i* Database Release 2[56], then at run-time it will announce that the fallback has been selected and will execute the old style loop to derive a dense collection for the bulk insert. If it is compiled on Oracle Database 10*g* Release 1 or later, then it will announce its ideal quality and will use the new *indices of* syntax to insert the sparse collection directly without needing the explicit programming effort — and run-time cost — to compact the data.

Notice the crucial difference between the power of the compile-time *$if* construct and the regular run-time *if* construct. When *Code_44* is conditionally compiled on Oracle9*i* Database Release 2, the rest of the compilation pipeline sees *only* the code appropriate for that release; the other code — which would not compile — vanishes. If this were not so, then this release of Oracle Database would suffer compilation errors on the new code.

Recall the explanation given in the section *The DBMS_DB_Version package* on page 15. If a compilation unit containing the construct shown in *Code_44* were deployed in production in a Oracle9*i* Database Release 2 environment and if an upgrade were made to Oracle Database 10*g* (at either Release 1 or Release 2), then the compilation unit would be invalidated. On its next use, it would be recompiled and would therefore start automagically to use the newer and more efficient implementation for the bulk insert.

---

55. The full context for *Code_44* is listed in *Appendix D: Self-contained SQL*Plus script from which Code_44 is an extract* on page 70.

56. Remember that PL/SQL conditional compilation has been made available in patchsets of releases of Oracle Database earlier than the one that introduced the feature. See *The availability of PL/SQL conditional compilation in Oracle Database 10g Release 1 and in Oracle9i Database Release 2* on page 59.

CASE STUDY: IMPLEMENTING UNIT TESTING, ASSERTIONS, AND TRACING FOR A FAST CUBE ROOT BODY-PRIVATE HELPER FUNCTION

This section first presents the problem that is the basis of the study and then it explains the design of the algorithm for the function that is required. Next, it sets out the requirements for the unit tests. All this is to motivate the actual PL/SQL implementation — and the next section shows how PL/SQL conditional compilation is used to achieve this effectively. This section is followed by a discussion of the test results. Because this is a relatively lengthy case study, it warrants its own conclusion.

Introduction to the case study

Imagine that during the implementation of a particular package body the need arises to find the cube root of a number[57] resulting from a computation within the package body. The designer can be sure what the range of values is; for this example, assume that the number whose cube root is needed lies between 1.0 and 32767.0 and is never *null*[58].

It is very easy to write a functionally correct version of the required helper, as *Code_45* shows.

```
-- Code_45
function Slow_Cbrt(n in Math_Real) return Math_Real is
  Three constant Math_Real :=  3.0;
begin
  return Exp(Ln(n)/Three);
end Slow_Cbrt;
```

Suppose, though, that — having used the *Code_45* implementation — a performance profiling exercise[59] has shown that, in executing the package's exposed subprograms, a considerable amount of time is spent in the *Slow_Cbrt()* function. Moreover, the private helper cube root function needs only to return an approximate result; the requirement is that the computed cube root *R_Fast* of the input value *N* must satisfy this inequality for all values of *N* within the specified range:

Abs(R_Fast - R_Slow)/R_Slow <= 0.03

---

57. SQL does not provide a *Cbrt* builtin function; nor does any supplied PL/SQL package expose this functionality.

58. This declaration succeeds:

    ```
    subtype My_Int is pls_integer range 1..32767;
    ```

    But this declaration fails with *PLS-00572: improper constraint form used*:

    ```
    subtype My_Real is binary_float range 1.0f..32767.0f;
    ```

    Enhancement request 4676454 asks to remove this restriction.

59. We hope, for the next major release of Oracle Database after *10.2*, to productize a hierarchical performance profiling tool for PL/SQL and the SQL that it invokes. The tool — in prototype form — has already been used to effect by development teams inside Oracle Corporation. Here, the tool would show the time spent in the exposed procedure *P()* and the breakdown of the time in *P()* to the self time and the time in its callees — in this example *Slow_Cbrt()*. Should it be interesting, the user could drill down to a corresponding breakdown for the self time in *Slow_Cbrt()* and the time in *its* callees — *Ln()* and *Exp()*.

R_Slow is the result returned by *Slow_Cbrt()* and is presumably the exact cube root to the limit of the precision of the datatype. This motivates the attempt to implement a special *Fast_Cbrt()* helper function; it needs to deal only with inputs in the range 1.0 to 32767.0 and it needs only three percent accuracy[60].

The design of the *Fast_Cbrt()* algorithm

The design of the algorithm will probably be familiar to most readers:

- Choose an approximate starting value for the root.
- Improve the approximate value repeatedly using the Newton-Raphson method until the next approximation is sufficiently close to the previous approximation.

The mathematical basis for this approach is explained in *Appendix F: The Newton-Raphson formula for improving an approximation for the cube root of a number* on page 72.

Real implementations for various mathematical functions which use this scheme are common and all face the same design challenges: how to choose a good starting approximation and how to make the iteration maximally efficient. The implementation of *Fast_Cbrt()* tries two different approaches:

- The first is naïve: it gives no care to the choice of the starting approximation (it merely divides the input by one hundred) and then it iterates slavishly until a tolerance is satisfied. I will refer to this as *Alt=1*. (You can guess that this corresponds to the value of the *ccflag* that selects this alternative.)
- The second is careful about the choice of the starting approximation (it divides the input range logarithmically into intervals and uses a table that holds the pre-calculated cube root for each interval) and then it applies the Newton-Raphson improvement just once. I will refer to this as *Alt=2*.

The requirements for the unit tests

The testing must address two dimensions: correctness and — because it was this that motivated the project — performance relative to *Slow_Cbrt()*.

- The correctness test must select closely-spaced input values over the whole specified range and must, for each, obtain the cube root using both *Fast_Cbrt()* and *Slow_Cbrt()* and compute the fractional error — *Abs(*R_*Fast* - R_*Slow)*/R_*Slow*. The correctness test must deliver the maximum value for this error[61]. Of course, the correctness test should be run with all assertions

---

60. I should say right away the task for this case study is — these days — slightly artificial. Rather than keep you in suspense, I will tell you now that when *Slow_Cbrt()* is implemented using the *number* datatype, then the *Fast_Cbrt()* that this case study examines is, on average, a *factor of twenty* faster than *Slow_Cbrt()*. However, when the datatype is changed to *binary_float* throughout — as would be natural for a low accuracy cube root calculation — then *Fast_Cbrt()* is faster than *Slow_Cbrt()* by a factor of "only" somewhere between 2.5x and 1.3x. (The result depends on the platform.) Nevertheless, the task of implementing *Fast_Cbrt()* provides a very good exemplar for the techniques that this paper presents. For that reason, I decided to keep to my plan and to use it in this section.

61. A real world test would probably derive measures of central tendency — for example, the mean and the standard error. I decided that this would have complicated my code with no pedagogical gain.

enabled. It might be useful to enable tracing during this test to generate a machine readable log for use in regression testing.

- The performance test must exercise *Slow_Cbrt()* with enough different input values that the total time is measurable to reasonable precision and must then exercise *Fast_Cbrt()* with the same set of input values. Because here it is computation time — and not data access time — that is significant, the *DBMS_Utility.Get_CPU_Time()* function[62], with a precision of one centisecond, is the natural choice for supporting the timing measurements. The performance test should be run with assertions and tracing disabled.

Discussion of the PL/SQL implementation

This section begins with an overview that provides a guide to the code and shows the *Alt=1* and *Alt=2* implementations as the compiler sees them when they are conditionalized for production deployment. It then discusses how the various conditional compilation constructs have been used.

Overview

The code is presented in *Appendix G: SQL*Plus scripts for the case study* on page 73 and is also available for download as explained in that appendix. At the highest level, the approach is unremarkable.

- The first script (*Setup.sql* on page 74) creates a brand-new Oracle user to own the schema objects that are needed for this case study. There are only four schema objects and all are PL/SQL compilation units: the package specification and body of *Pkg* — which models the package in which the need for the cube root helper arises; the procedure *Print* that wraps *DMBS_Output.Put_Line()*; and the procedure *Test_It* that invokes both *Pkg.P()* — which models the package's "ordinary" API — and *Pkg.Run_The_Tests()*. I decided to implement the unit tests within the package body itself. This first script also creates the specification of *Pkg*.
- The second script (*Create_Package_Body.sql* on page 75) is responsible for all the pedagogy in this study; it creates the body of *Pkg*.
- The third script (*Create_Test_Harness.sql* on page 84) creates the procedure *Test_It*.
- The fourth script (*Exercise_Test_Harness.sql* on page 85) repeatedly recompiles the package body *Pkg*, with different values for the controlling *cflag*s, and after each recompilation invokes *Test_It()*.

The rest of the discussion can concentrate on the design of the package body *Pkg*. The best starting point is the function *Fast_Cbrt()* itself (see page 82). As mentioned, my implementation provides both approaches in the single script file that creates the body of *Pkg*; the code where they differ is guarded by the *inquiry directive $$Alt*. The body of *Pkg* also has assertions guarded by the *inquiry directive $$Asserting*; and it has tracing code guarded by the *inquiry directive $$Tracing*.

---

62. The *DBMS_Utility.Get_CPU_Time()* function was introduced in Oracle Database 10*g* Release 1 as a partner to *DBMS_Utility.Get_Time()* which measures elapsed wall-clock time.

Code_46 shows the code for *Fast_Cbrt()* that is compiled when *Alt=1* and when *Asserting* and *Tracing* are *false*. (I removed blank lines by hand.)

```
-- Code_46
function Fast_Cbrt(n in Math_Real) return Math_Real is
  Root Math_Real := null;
begin
  -- Starting Approximation.
  Root := n/One_Hundred;

  -- Newton-Raphson improvement.
  -- The classic convergence test.
  declare
    Tolerance constant Math_Real :=  0.01;
    Last_Root Math_Real := n;
  begin
    while Abs(Root - Last_Root)/Root > Tolerance loop
      Last_Root := Root;
      Root := (Two*Root*Root*Root + n)/(Three*Root*Root);
    end loop;
  end;
  return Root;
end Fast_Cbrt;
```

Two, *Three*, and *One_Hundred* are constants[63] with the obvious meaning declared at package level. The subtype *Math_Real* is also declared at package level. You can guess that it is either *number* or *binary_float* according to the value of the *ccflag Use_Number*.

Code_47 shows the code for *Fast_Cbrt()* that is compiled when *Alt=2* and when *Asserting* and *Tracing* are *false*.

```
-- Code_47
function Fast_Cbrt(n in Math_Real) return Math_Real is
  Root Math_Real := null;
begin
  -- Starting Approximation.
  declare
    Idx pls_integer := Number_Of_Thresholds;
  begin
    while Idx > 0 loop
      if n >= Thresholds(Idx) then
        Root := Starting_Cbrts(Idx);
        exit;
      end if;
      Idx := Idx - 1;
    end loop;
  end;

  -- Newton-Raphson improvement.
  -- It is now sufficient to iterate just ONCE because
  -- the starting approximation is so good.
  Root := (Two*Root*Root*Root + n)/(Three*Root*Root);
  return Root;
end Fast_Cbrt;
```

Code_47[64] also depends on constants with the obvious meanings and on the subtype *Math_Real*. It depends further on the collection *Thresholds* — which stores the boundaries of the intervals into which the input range is logarithmically divided — and *Starting_Cbrts* — which stores the pre-calculated

63. You might think that this is a bit excessive. I prefer this style, not only because it reduces the risk of typos, but also because such constants have an explicitly declared datatype and therefore their use as actuals for overloaded subprograms is easier (for the human) to understand. Of course, the compiler knows its rules and is never confused.

cube roots[65]. Both of these collections are declared as *constants*[66] at package level and are of datatype *varray of Math_Real*.

The use of PL/SQL conditional compilation

The code abounds with *selection directives*. In this case, every one tests an *inquiry directive* because all the conditionalization choices are to be made to support experimentation, debugging, quality assurance, and testing during the development phase. The following aspects of the implementation are conditionalized:

- I implemented the procedure *Run_The_Tests()* (see page 79) inside the body of package *Pkg* following the second of the two approaches described in the section *Unit testing of subprograms declared only in a package body* on page 27. The performance test is implemented as the inner procedure *Time_Slow_And_Fast()* (see page 80); the correctness test is implemented as the inner procedure *Find_Max_Fractional_Error()* (see page 80).

  Run_The_Tests() and various helpers it needed were declared conditionally using the *ccflag Testing*. I preferred this to the first approach described in that section — exposing *Fast_Cbrt()* conditionally for an external testing program to exercise it — for these reasons.

  - The performance test should be as direct as possible. *Fast_Cbrt()* in normal use will be called only from other sites inside the same compilation unit. I wanted to avoid needing to reason about the cost of an extra level of invocation — using, for example, an *Expose_Fast_Cbrt()* function — and about the possible extra cost of invoking a subprogram from a different compilation unit[67].
  - The function *Slow_Cbrt()* (see page 78), which has no role except in correctness and performance testing, is immediately accessible to the testing subprogram without the added burden of exposing yet another element conditionally.

---

64. The last two statements would more naturally be collapsed, thus:

    ```
    return (Two*Root*Root*Root + n)/(Three*Root*Root);
    ```

    However, the final value of Root needs to be available for tracing and for the assertion when these are conditionally selected.

65. *Starting_Cbrts(n)* stores the cube root of the geometric mean of *Thresholds(n)* and *Thresholds(n+1)*.

66. I went to some trouble to declare *Starting_Cbrts* and *Thresholds* as *constants*. As has been mentioned, this is always a sound correctness idiom and can often help the optimizing compiler. I have learned that the technique — forward declare a function to compute the values, initialize the constant using the function, and then define the function — is not widely known. The full details are shown in *Appendix G: SQL*Plus scripts for the case study* on page 73 and in the downloadable code.

67. This consideration might become less significant in next major release of Oracle Database after *10.2* when, as it is hoped, intra-unit and inter-unit inlining are supported. The former will be easier to control. Nevertheless, writing the test programs outside of the package would — in a performance test — incur a cost of thinking things through and writing them up in the test report. The putative benefit of writing the tests in a separate file so that two developers could work concurrently, each on his own file, would not, in my view, justify the cost.

- I wanted to produce trace output to show the Newton-Raphson convergence. Yet, because, I would invoke *Fast_Cbrt()* about a million times during the test sequence (650,000 times for the CPU time measurement and the same number for the correctness test) I needed to be able to generate the trace output only on every *Nth* call (I found that every 65,00th call was convenient) and then only during the correctness test. This was easy to achieve by incrementing and testing a counter declared — conditionally on *Tracing*, of course — at package level.

- As mentioned, I wanted to trace the Newton-Raphson convergence. I also wanted to print the *Thresholds* and *Starting_Cbrts* collections on which the *Alt=2* approach critically depend. The *ccflag Tracing* controls these print statements and also guards the code — implemented cooperatively in the procedures *Find_Max_Fractional_Error()* (see page 80) inside *Run_The_Tests()* and *Fast_Cbrt()* (see page 82) — that ensures that the Newton-Raphson convergence is traced only on every *Nth* call. In fact, this frequency was controlled by the *ccflag Trace_Step*. It was very much more convenient to control all the testing parameters using a single *alter... compile* statement than it would have been to invent some other way to vary the tracing frequency.

- The datatype of the subtype *Math_Real* — *number* or *binary_float* — is determined by the *ccflag Use_Number*. This makes no other appearance in the code except to provide the appropriate information — *"Number version"* or *"IEEE version"* — in the trace output. I included this more as an indulgence — to demonstrate the phenomenal performance difference between *number* and *binary_float* — than as an example of an expected realistic choice[68].

- Assertions are enabled or disabled by the *ccflag Asserting*. The procedure *Assert_Valid_Input()* (see page 77) is invoked on entry to *Fast_Cbrt()* and to *Slow_Cbrt()*. The procedure *Assert_Cbrt_Cubed_Gives_Input()* (see page 77) is invoked immediately before each function return. The declarations of these functions are at package level and are also guarded by *Asserting*. These procedures, in turn, use similarly guarded package level helpers to format error messages in the event of assertion failure.

  Other uses for assertions arose naturally in connection with the *Alt=2* approach:

  - The function *Calculated_Thresholds()* (see page 81), which initializes the constant *Thresholds* collection, should end up having computed — by incrementing and testing whether a limit is exceeded — the intended number of values. Failure to do so is a classic example of the "logical impossibility" and would indicate a bug (for example, some forgotten tolerance in a comparison of mathematical real numbers)[69]. A similar case

---

68. Just conceivably, the code in question might have to be viable in Oracle9*i* Database Release 2 where the *binary_float* datatype is not available. In that case, as will be made clear later (see *The availability of PL/SQL conditional compilation in Oracle Database 10g Release 1 and in Oracle9i Database Release 2* on page 59), it would not be practical to use *inquiry directives* — and therefore every *selection directive* would have to test a static package constant.

69. Of course, I was smitten by just such a bug while I developed this code.

arises in the function *Calculated_Cbrts()* (see page 81) which initializes the constant *Starting_Cbrts* collection.

- And the loop in the body of *Fast_Cbrt()*, which aims to place the input value in the interval to which it belongs, must succeed. Otherwise, again, there is a "logical impossibility".

- Finally, the choice between the two approaches, *Alt=1* and *Alt=2*, was governed by the *ccflag Alt*. Unlike the other flags, which were boolean in nature and for which the notion of a list of values therefore does not apply, there might in principle be more than two alternatives for consideration during prototyping. I therefore designed *Alt* to act as a *pls_integer* and implemented a *error directive* to ensure that it had only the values 1 or 2. Notice that the implementation of *Fast_Cbrt()* has a significant amount of common code. (In this example, all the common code except the declare section and the return statement implements assertions and tracing.)

The final production version of the code would be unlikely to retain the *Alt=1* code (the next section shows that *Alt=2* is the better choice) and would probably declare the subtype *Math_Real* unconditionally as *binary_float*. However, all the other conditional compilation directives — using *Asserting*, *Tracing*, *Trace_Step*, and *Testing*, could be expected to remain in place to support possible maintenance cycles.

The test results

The first test compiles the body of *Pkg* for normal production use. It sets *Tracing*, *Trace_Step*, *Asserting*, and *Testing* to *null* to disable these development-time quality assurance features; it sets *Use_Number* to *null* to select the *binary_float* implementation; and it sets *Alt* to select the faster *Alt=2* cube root method. By design, the invocation of *Pkg.Run_The_Tests()* raises an exception. The procedure *Test_It()* catches this and reports *"Run_The_Tests() is disabled for production deployment."*. *Test_It()* also invokes *Pkg.Some_Proc()* — which models *Pkg*'s intended public API; this runs without impact from the development-time quality assurance features, passes its self-check, and reports *"Some_Proc finished OK."*. This output is shown on page 88.

The second test compiles the body of *Pkg* for maximal development-time quality assurance. It uses the *number* implementation and the slower *Alt=1* cube root method. Notice that, by setting *Trace_Step=65536*, the Newton-Raphson iteration for just nine representative cube root calculations is traced. This output is listed on page 88 and page 89.

The next four tests exercise the *number* implementation with tracing turned off. The first two of these tests, using *Alt=1* and then *Alt=2*, are run with the assertions turned on to ensure correctness. The second two of these four tests, again using *Alt=1* and then *Alt=2*, are run with the assertions turned off to allow accurate timing. The output for these four tests is listed on page 90.

The final four tests repeat the previous four tests using the *binary_float* implementation. The output for these four tests is listed on page 91.

Conclusion to the *Fast_Cbrt()* case study

Although the exercise — to implement a fast function to calculate the cube root of a number lying within a relatively narrow range and to a stated low accuracy

— was invented in order to show the advantageous use of PL/SQL conditional compilation, it is nevertheless interesting to comment on the success of the implementation.

- The specially designed *Alt=2* approach — which takes specific account of the specified narrow range of input values to select a good starting estimate from manageably small a pre-computed list — succeeded in producing a sufficiently accurate result over a very finely sampled set of input values that covered the specified range. The naïve *Alt=1* approach also passed the correctness test. But the performance test showed the *Alt=2* approach to be very much faster.
- If one were forced to implement the fast cube root function in an earlier release than Oracle Database 10*g* Release 1 — before the availability of the IEEE datatypes for real numbers — then the *Alt=2* approach is overwhelmingly faster, by a *factor of twenty*, than the formula approach (exponentiating one third of the natural logarithm).
- If one is able to use a modern version of Oracle Database — and for an application such as this it would seem crazy not to — then the *Alt=2* approach less dramatically faster than the formula approach. I measured a performance improvement factor of 2.5x when I tested the code on a Linux x86 machine and a rather smaller improvement factor of 1.3x when I tested the code on Windows XP Pentium machine.

Of course, the main conclusion is that the use of PL/SQL conditional compilation helped me enormously this implementation project:

- it significantly increased my productivity when prototyping the two alternative approaches.
- and it allowed my final deliverable to include "executable" documentation in the form of latent assertions, tracing, and unit testing procedures.

I suggest that you try to imagine how you would have implemented all the quality requirements for this project — enabling tracing on a sampling basis, implementing assertions without penalizing performance in production deployment, testing for correctness and performance of the body-only procedure, and experimentally comparing the desirability of two (or more) competing algorithm designs — without conditional compilation. I suspect that you would be driven to implement what amounts to this functionality by hand by formalizing a regime of uncommenting and commenting out what in the implementation shown here was handled by using *selection directives*. Of course, this would be substantially more time-consuming and — even worse — substantially more error-prone.

HOW DOES PL/SQL CONDITIONAL COMPILATION COMPARE WITH SIMILAR FEATURES IN OTHER PROGRAMMING ENVIRONMENTS?

Preprocessors have been around for as long as compilers. Wiki[70] gives a good general account which begins as follows.

> "In computer science, a preprocessor is a program that takes text and performs lexical conversions on it. The conversions may include macro substitution, conditional inclusion, and inclusion of other files."

As this implies, the preprocessor is usually a distinct program that runs before the compiler proper. We know of no implementation, except that of PL/SQL conditional compilation, where the input to the preprocessor and its output are anything but ordinary operating system files. And in these other implementations, inclusion of one file in another — usually in many other files — is essential to the definition and the visibility of the "flags" that control conditionalization.

The PL/SQL implementation is distinguished from the usual approach by these unique features:

- The syntax and semantics of PL/SQL's conditional compilation directives are part of the definition of the PL/SQL language.
- PL/SQL's conditional compilation is implemented as a step — and not the first step — *within* the PL/SQL compiler. It is not a separate program that runs before the compiler.
- The source text — as the PL/SQL compiler sees it[71] — resides in the Oracle Database and not on the file system.
- The primitive determinants of conditionalization — static package constants and *ccflag*s — are stored inside and are managed by Oracle Database. Therefore — in order that PL/SQL conditional compilation can be fully functional for the very wide range of applications that this paper has discussed — there is no need for a mechanism to include one source fragment within another.
- The static package constants and *ccflag*s used to express the test for a *selection directive* are combined in a boolean expression which is evaluated at compile-time using exactly the same mechanism in the Oracle executable that would evaluate such boolean expressions at run time. This is the implementation that guarantees (as was stated in the section *The selection directive* on page 6) that the rules for evaluating the static *boolean* expression that governs the selection of text for compilation are the same rules that the PL/SQL programmer has learned for ordinary PL/SQL expressions. This helps us define the syntax and semantics of conditional compilation as part of the definition of the PL/SQL itself.

---

70. See *en.wikipedia.org/wiki/Preprocessor*

71. Think of the *create or replace* statement as doing two distinct operations: first it loads the text into the catalog table that is exposed by the *All_Source* view family; and then it reads the source from this and compiles it. Recall that the *alter* statement operates directly on the stored source from the catalog.

- The static package constants within *selection directives* set up dependencies in exactly the same way that *any* reference from a compilation unit to an external element sets up dependencies for that unit. The consequences of this — possible invalidation and subsequent implicit recompilation — are well understood by PL/SQL programmers. This mechanism completely removes the need to construct and to maintain *makefiles*.

As Wiki says, not only do preprocessors usually support the inclusion of files, but also they support macro substitution. PL/SQL conditional compilation in Oracle Database 10*g* Release 2 supports neither of these features. However, there is nothing in the design and present implementation that would prevent the addition of these features in a later release. Notice, though, that opinion varies on the desirability of relying on a highly functional preprocessor. Here are two extracts from the Wiki article to conclude this section:

> "...overuse of the preprocessor might yield quite chaotic code..."

> "...use of preprocessors... getting less common as... languages provide more abstract features rather than lexical-oriented ones."

THE AVAILABILITY OF PL/SQL CONDITIONAL COMPILATION IN ORACLE DATABASE 10*g* RELEASE 1 AND IN ORACLE9*i* DATABASE RELEASE 2

Unusually, but for very compelling reasons, PL/SQL conditional compilation has been made available in patchsets of releases of Oracle Database earlier than the one that introduced the feature. It is available in the first release of Oracle Database 10*g* from *10.1.0.4* onwards and in Oracle9*i* Database from *9.2.0.6* onwards.

- In *10.1.0.4*, the feature is available by default but can be totally disabled by setting the PL/SQL conditional compilation underscore parameter[72] to *"disable condtional compilation"*.
- In *9.2.0.6*, the feature is totally disabled by default but can be made available by setting the PL/SQL conditional compilation underscore parameter to *"enable condtional compilation"*.
- From *10.2*[73], the feature cannot be disabled and the PL/SQL conditional compilation underscore parameter is obsolete.

This section explains the rationale for making the feature available in *10.1.0.4* and in *9.2.0.6* and describes the feature's functionality restrictions — with respect to the full *10.2* functionality — in these releases.

Of course, you can skip this section entirely if you will use PL/SQL conditional compilation only in Oracle Database 10*g* Release 2 and later releases.

The Catch 22

ISVs generally support every currently supported release of Oracle Database. The earliest of these is often two releases behind the latest release. For example, at the time of writing, *10.2* is the latest release and *9.2* is still fully supported by Oracle Corporation. Therefore, if PL/SQL conditional compilation had been made available only from *10.2* onwards, then the benefit that motivated the introduction of the feature (see *Spanning different releases of Oracle Database with a single source code corpus* on page 46) would not have been realized until the *second* major release after the one that introduced the feature; until then, ISVs will still be supporting *10.1* — and this would not have had PL/SQL conditional compilation. Oracle's PL/SQL Team and Oracle's Applications Division were unanimous that such an outcome would have been unacceptable.

---

72. An underscore parameter is a special kind of initialization parameter. It is never documented. When it has not been set, it is not listed in the *v$parameter* view. It cannot be set while the instance is running (the attempt causes *ORA-02095*) and so it must be set either via the *pfile* or by using *"alter system ... scope = spfile"*. Customers are permitted to set an underscore parameter only under the direct guidance of Oracle Support.

73. This section will refer repeatedly to various major releases of Oracle Database. For brevity, these nicknames will be used:

    Oracle9*i* Database Release 1 . . . . . *9.0*
    Oracle9*i* Database Release 2 . . . . . *9.2*
    Oracle Database 10*g* Release 1 . . . . . *10.1*
    Oracle Database 10*g* Release 2 . . . . . *10.2*
    next major release of Oracle Database after *10.2* . . . . . *R.Next*

This, then, is the Catch 22: in order that PL/SQL conditional compilation can deliver its motivating benefit in the new Oracle Database release for which the project was done, it must be made available — in patchsets — in each earlier release that is still supported at the time that the new release becomes generally available. This was one of the non-negotiable requirements; the design of the feature took account of this from the outset.

The decision to make PL/SQL conditional compilation available in *10.1* and in *9.2*

Normally a new feature must be introduced only in a *major* release of Oracle Database. However, the explanation of the Catch 22 shows that PL/SQL conditional compilation warranted exceptional treatment. A very careful risk analysis was conducted. It reached these conclusions:

- It was feasible to make the PL/SQL conditional compilation feature available — with some functionality restrictions — in patchsets for *10.1* and for *9.2*[74].
- The syntax for the new constructs (the *selection directive*, the *inquiry directive*, and the *error directive*) was very carefully designed to guarantee the following:
  - For an input source text that compiled without error in *10.1* (prior to the *10.1.0.4* patchset) and in *9.2* (prior to the *9.2.0.6* patchset), PL/SQL conditional compilation will have no effect.[75]
  - Conversely, PL/SQL source text that uses any of the new constructs will fail to compile in an environment where PL/SQL conditional compilation does not exist.

  The use of the *$* sign as the so-called *trigger character* guarantees these two conclusions.
- The source text output by the conditional compilation stage goes through the same subsequent compiler stages as it would in a release of Oracle Database where PL/SQL conditional compilation does not exist.
- PL/SQL conditional compilation adds no measurable time to the compilation of a source text that has no conditional compilation directives.
- The design of the feature allows the new compilation regime that supports PL/SQL conditional compilation to coexist with the compilation regime from before the advent of the feature. A simple switch can enable the new regime or disable it in favor of the old regime. When the new regime is disabled, compilation is guaranteed to be completely unaffected by the conditional compilation stage. The choice of regime can be made using an underscore parameter.

These conclusions were presented to senior management in both the Development organization and the DDR organization[76] for Oracle Database

---

74. Following a cost-benefit analysis it was decided to compromise; PL/SQL conditional compilation is not available for *9.0*.

75. This is achieved because, for such texts, the output from the conditional compilation stage will be identical to the input. See *How does PL/SQL conditional compilation work?* on page 17.

and it was agreed to treat PL/SQL conditional compilation exceptionally as follows:

- The feature would be made available in the *10.1.0.4* and *9.2.0.6* patchsets.
- The user-defined *inquiry directive* feature (which depends on the new-in-*10.2 PLSQL_CCFlags* parameter) would not be made available. This — and some other minor restrictions — are described in the section *Functionality restrictions in 10.1 and 9.2* on page 61.
- The PL/SQL conditional compilation underscore parameter would be implemented to enable or disable conditional compilation. Its default would be *"enable condtional compilation"* in *10.1.0.4* and *"disable condtional compilation"* in *9.2.0.6*.
- The availability of PL/SQL conditional compilation in *10.1.0.4* would be described in the Oracle Database Documentation Library but the availability in *9.2.0.6* would not. In line with normal policy, the PL/SQL conditional compilation underscore parameter would not be documented.

The plan was carried out as described and, of course, PL/SQL conditional compilation was subjected to extensive testing — both for correctness of the new behavior it supports and for no impact on source text that makes no use of the feature. The tests for *10.1.0.4* and *9.2.0.6* were conducted with both values — *"enable condtional compilation"* and *"disable condtional compilation"* — for the PL/SQL conditional compilation underscore parameter. There are no known PL/SQL bugs caused by conditional compilation in any of *10.2.0.1*, *10.1.0.4*, or *9.2.0.6*.

The PL/SQL conditional compilation underscore parameter

This paper intentionally does not give the name of the PL/SQL conditional compilation underscore parameter that is used to reverse the default by enabling PL/SQL conditional compilation in *9.2.0.6* or by disabling it in *10.1.0.4*.

Customers who want to reverse the default must contact Oracle Support. A Support Engineer will explain the procedure for setting the PL/SQL conditional compilation underscore parameter.

Functionality restrictions in *10.1* and *9.2*

No restrictions for the *DBMS_DB_Version* package

The *DBMS_DB_Version* package is installed, with appropriate source, in each of *10.1.0.4* and *9.2.0.6* (see *Appendix C: The source code of the DBMS_DB_Version package in 10.2, 10.1, and 9.2* on page 69). Its exposure does *not* depend on the PL/SQL conditional compilation underscore parameter. Customers may use it for any purpose. *Code_48* shows a regular run-time *if* construct that determines

76. DDR — the Defects Diagnosis and Resolution organization — is responsible for determining the causes of product bugs and fixing them. It administers the production of patchset releases.

the correct action by testing one of the constants exposed by the *DBMS_DB_Version* package.

```
-- Code_48
if DBMS_DB_Version.Ver_LE_9_2 then
  Print('do something right in 9.2');
else
  Print('do something right in >= 10.1');
end if;
```

Compare this with *Code_49*.

```
-- Code_49
$if DBMS_DB_Version.Ver_LE_9_2 $then
  Print('do something right in 9.2');
$else
  Print('do something right in >= 10.1');
$end
```

Code_49 was derived "mechanically" from *Code_48* simply by replacing the run-time *if* construct keywords with their compile-time *$if* construct counterparts. In this use case, PL/SQL conditional compilation is used to choose between alternatives all of which are viable in each release of Oracle Database of interest. This is a perfectly respectable application of the feature (we expect to see this use in especially connection with self-tracing or self-debugging code), but the feature is not *required* for this use. It is easy to see that *Code_48* and *Code_49* will give identical behavior. *Code_49* has a theoretical advantage over *Code_48*: it has less executable code and it runs faster[77]. But it has the significant practical disadvantage that it requires that the PL/SQL conditional compilation underscore parameter is actively set to *"enable condtional compilation"* in *9.2.0.6*.

Developers are discouraged from using PL/SQL conditional compilation in *9.2.0.6* if its *only* purpose is to support the compile-time *$if* construct where the run-time *if* construct could be used instead.

No restrictions for the *error directive*

The *error directive per se* is supported in each of *10.2*, *10.1*, and *9.2*. However, some of its value depends on being able to report an erroneous value of a static package constant of datatype *pls_integer* or of an *inquiry directive* that produces this datatype. This depends on using the *To_Char()* built-in (see *Restrictions for the To_Char() built-in* on page 63).

Restrictions for the *inquiry directive*

The *PLSQL_CCFlags* PL/SQL compilation parameter is not implemented in *10.1* or *9.2* and is therefore not reflected in the corresponding dictionary views (the *All_PLSQL_Object_Settings* view family in *10.1* and the *All_Stored_Settings* view family in *9.2*). The PL/SQL compilation parameters (in the release in question), *PLSQL_Unit*, and *PLSQL_Line* are supported.

When an *inquiry directive* refers to an unknown *ccflag*, the compilation error *PLS-00175: unknown inquiry directive...* is raised. (This is the case both when the source is not wrapped and when it is wrapped.)

---

77. The differences and similarities between the compile-time *$if* construct and the run-time *if* construct are discussed in the section *Choosing between the compile-time $if construct and the run-time if construct* on page 20.

Because of these restrictions, we expect that the *inquiry directive* will be used in *10.1* and *9.2* only to test expressions based on static package constants.

Restrictions for the *To_Char()* built-in

In *10.2*, an invocation of the *To_Char()* built-in with just one actual argument of datatype *pls_integer* is taken by PL/SQL conditional compilation to be a *static* function[78]. This means that it can be used in the *error directive* as shown in *Code_50*.

```
-- Code_50
procedure P is
begin
  $error 'wrong optimize level:'||
    To_Char($$PLSQL_Optimize_Level) $end
end P;
```

An attempt to compile *P()* causes this compilation error:

```
PLS-00179: $ERROR: wrong optimize level:2
```

In *10.1* and *9.2*, an invocation of the *To_Char()* built-in with just one actual is *not* taken to be static. Therefore, in *10.1*, *Code_50* fails to compile with *this* error:

```
PLS-00178: a static character expression must be used
```

In *9.2*, the PL/SQL compilation parameter *PLSQL_Optimize_Level* is unknown, but a corresponding attempt to use the *error directive* with a static package constant of datatype *pls_integer* fails in the same way.

This restriction will be met only in connection with the *error directive*.

Notice that in *10.2* — but only here — the conversion from *pls_integer* may sometimes be implicit. *Code_50* may be rewritten as *Code_51*.

```
-- Code_51
procedure P is
begin
  $error 'wrong optimize level:'||$$PLSQL_Optimize_Level $end
end P;
```

Restrictions for the *DBMS_Preprocessor* package

The full functionality, as implemented in *10.2* has been explained earlier (see *Using the DBMS_Preprocessor package to see the conditional compilation output* on page 17).

The *DBMS_Preprocessor* package is present in *10.1* and has identical behavior to *10.2*. (Of course, the PL/SQL conditional compilation constructs themselves are limited with respect to *10.2* as already discussed and so some source texts that would compile without error in *10.2* will cause compilation errors in *10.1*.) Its presence is unaffected by the current value of the PL/SQL conditional compilation underscore parameter but, of course, it is meaningless to use it unless this has the value *"enable condtional compilation"*.

The *DBMS_Preprocessor* package is not present in *9.2*.

---

78. *To_Char(x, f, n)* where *x* is a *pls_integer* static expression and *f* and *n* are *varchar2* static expressions is also taken to be a static function.

The purpose of the *Ver_LE_* constants in the *DBMS_DB_Version* package

Look again at *Code_44* on page 48. Why does it begin with this?

```
$if DBMS_DB_Version.Ver_LE_9_2 $then
```

Why does it not just test on *Version* and *Release* explicitly? The two approaches express precisely the same condition and, of course, produce the same result, as *Code_52* and *Code_53* show.

```
-- Code_52
procedure P is
begin
  $if DBMS_DB_Version.Ver_LE_9_2 $then
    Print('Ver <= 9.2');
  $else
    Print('Ver > 9.2');
  $end
end P;

-- Code_53
procedure P is
begin
  $if (DBMS_DB_Version.Version = 9
       and DBMS_DB_Version.Release <= 2)
      or DBMS_DB_Version.Version < 9
  $then
    Print('Ver <= 9.2');
  $else
    Print('Ver > 9.2');
  $end
end P;
```

The rationale for preferring the approach used in *Code_52*[79] can be understood only in the light of plans for the next major release of Oracle Database after *10.2* — hereinafter *R.Next* — to introduce the so-called *fine-grained-dependency* feature[80]. This feature will replace the dependency model seen through *10.2* with

79. *Code_53* is also more verbose than *Code_52* and there is some risk of programmer error — but that is not the important point here.

80. The following account of the planned fine-grained-dependency feature represents Oracle's PL/SQL Team's *intention* at the time of writing. It is *not a commitment* to deliver the functionality in next major release of Oracle Database after 10.2 nor in any subsequent release.

a refined model designed to reduce consequential invalidation. Run the SQL*Plus script shown in *Code_54* in *10.2*.

```
-- Code_54
create package Pkg is
  procedure P1;
end Pkg;
/
create procedure P is
begin
  Pkg.P1();
end P;
/
-- Shows 'VALID'
select Status from User_Objects
  where Object_Type = 'PROCEDURE' and Object_Name = 'P'
/
create or replace package Pkg is
  procedure P1;
  procedure P2;
end Pkg;
/
-- Shows 'INVALID'
select Status from User_Objects
  where Object_Type = 'PROCEDURE' and Object_Name = 'P'
/
alter procedure P compile reuse settings
/
-- Shows 'VALID'
select Status from User_Objects
  where Object_Type = 'PROCEDURE' and Object_Name = 'P'
/
```

There is no logical requirement that the addition of the new subprogram *P2()* to *Pkg* should invalidate the compilation unit *P* — and in *R.Next* it will not. In *10.2*, the metadata expresses the fact that compilation unit *P* depends on compilation unit *Pkg* as a whole. In *R.Next*, the metadata will express the fact that compilation unit *P* depends on the element *Pkg.P1*.

Therefore, when *Code_52* is compiled in *R.Next*, the metadata will express the fact that compilation unit *P* depends on the element *DBMS_DB_Version.Ver_LE_9_2*; and when *Code_53* is compiled in *R.Next*, the metadata will express the fact that compilation unit *P* depends on the two elements *DBMS_DB_Version.Version* and *DBMS_DB_Version.Release*.

A constant such as *DBMS_DB_Version.Ver_LE_9_2* will never change its value in any later release following the one that introduces it. But at least one of the constants *DBMS_DB_Version.Version* and *DBMS_DB_Version.Release* will change in value from one major release to the next.

Therefore, using the *Ver_LE_* constants will "future-proof" code against unnecessary invalidations in the following circumstances: application code that tests one or more *Ver_LE_* constant is installed and deployed; then the Oracle Database is upgraded "under the feet" of that application.

CONCLUDING REMARKS

This paper has shown that PL/SQL conditional compilation is elegantly designed, easy to understand, and easy to use — not least because the syntax and semantics of the feature are part of the PL/SQL language itself.

It has also shown, by examining seven distinct use cases, that its simplicity belies its power.

- It provides the individual developer with new techniques for prototyping and for tracing that will boost quality of code and personal productivity.
- It lets development managers define new best practices to formalize quality assurance.
- It supports a new paradigm for component based design.
- It allows architects to create designs that use the latest SQL and PL/SQL features brought by the most recent Oracle Database in such a way that applications which must also run in the environment of earlier releases can be viable when these new features are not available.

Enjoy.

Bryn Llewellyn,
PL/SQL Product Manager, Oracle Headquarters
bryn.llewellyn@oracle.com
10-November-2005

APPENDIX A: CHANGE HISTORY

20-September-2005

- First published version.

26-September-2005

- Correcting minor typos.

14-October-2005

- New wording in the section *The PL/SQL conditional compilation underscore parameter* on page 61.
- Changing footnote numbering to continue from the previous footnote throughout the whole paper. (Before, the numbering restarted at 1 on each page.)
- Correcting minor typos.

18-October-2005

- Correcting minor typos.

10-November-2005

- Adding the description of how to use a DDL trigger to re-write the conditional compilation control package in the use case *Component based installation* on page 40.
- Correcting minor typos.

APPENDIX B:
ORACLE DATABASE DOCUMENTATION LIBRARY REFERENCES

PL/SQL conditional compilation is explained in the *PL/SQL User's Guide and Reference* book. The section *Understanding Conditional compilation*
download.oracle.com/docs/cd/B19306_01/appdev.102/b14261/overview.htm#sthref188
gives a brief overview. The section *Conditional compilation*
download.oracle.com/docs/cd/B19306_01/appdev.102/b14261/fundamentals.htm#sthref545
gives the full explanation.

The *DBMS_DB_Version* and *DBMS_Preprocessor* packages are described in the *PL/SQL Packages and Types Reference* book. *DBMS_DB_Version* is described here
download.oracle.com/docs/cd/B19306_01/appdev.102/b14258/d_dbver.htm#sthref2260
and *DBMS_Preprocessor* is described here
download.oracle.com/docs/cd/B19306_01/appdev.102/b14258/d_preproc.htm#sthref5443

The SQL syntax to set the *PLSQL_CCFlags* PL/SQL compilation parameter at system level, at session level, and — as part of the *alter <plsql unit> compile* command — for an individual PL/SQL compilation unit is given in the *SQL Reference* book
download.oracle.com/docs/cd/B19306_01/server.102/b14200/toc.htm

The *All_PLSQL_Object_Settings* view family, which shows the value of *PLSQL_CCFlags* for a PL/SQL compilation unit is described in the *Reference* book
download.oracle.com/docs/cd/B19306_01/server.102/b14237/toc.htm

APPENDIX C: THE SOURCE CODE OF THE *DBMS_DB_VERSION* PACKAGE IN *10.2*, *10.1*, AND *9.2*

The easiest way to be certain what the values are of the static constants that the *DBMS_DB_Version* package exposes in a particular Oracle Database of interest is to connect as any user and to run this query:

```
select text from all_source
  where owner = 'SYS'
  and name = 'DBMS_DB_VERSION'
  and type = 'PACKAGE'
  order by line
```

The constants convey information only about the version number and the release number for the Oracle Database and so the values will not change when a patchset is applied.

The results of the *all_source* query for each of *10.2*, *10.1*, and *9.2* are shown below. To make comparison easier, the comments were removed by hand.

For *10.2*:

```
package DBMS_DB_Version is
  Version       constant pls_integer := 10;
  Release       constant pls_integer := 2;

  Ver_LE_9_1    constant boolean := false;
  Ver_LE_9_2    constant boolean := false;
  Ver_LE_9      constant boolean := false;
  Ver_LE_10_1   constant boolean := false;
  Ver_LE_10_2   constant boolean := true;
  Ver_LE_10     constant boolean := true;
end DBMS_DB_Version;
```

For *10.1*:

```
package DBMS_DB_Version is
  Version       constant pls_integer := 10;
  Release       constant pls_integer := 1;

  Ver_LE_9_1    constant boolean := false;
  Ver_LE_9_2    constant boolean := false;
  Ver_LE_9      constant boolean := false;
  Ver_LE_10_1   constant boolean := true;
  Ver_LE_10     constant boolean := true;
end DBMS_DB_Version;
```

For *9.2*:

```
package DBMS_DB_Version is
  version       constant pls_integer := 9;
  release       constant pls_integer := 2;

  Ver_LE_9_1    constant boolean := false;
  Ver_LE_9_2    constant boolean := true;
  Ver_LE_9      constant boolean := true;
end DBMS_DB_Version;
```

APPENDIX D: SELF-CONTAINED SQL*PLUS SCRIPT FROM WHICH *CODE_44* IS AN EXTRACT

Code_44 on page 48 shows an extract of the following script from the *$if* through to the *$end*.

```
-- Connect as an ordinary "connect, resource" user
-- in turn to a 9.2.0.6 database, to a 10.1.0.4 database,
-- and to a 10.2 database.

CONNECT Usr/p@rel_10_2
create procedure Print(v in varchar2) is
begin
  DBMS_Output.Put_Line(v);
end Print;
/
create table Tbl (m integer, n integer)
/
create procedure P is
  subtype Index_t is pls_integer;
  type t is table of Tbl%rowtype index by Index_t;
  Sparse t; Dense t;
begin
  -- make a sparse table w/ 2500 elements
  for k in 1..50
  loop
    for j in 1..50
    loop
      Sparse(k*100 + j).n := k*j;
      Sparse(k*100 + j).m := k*j;
    end loop;
  end loop;

  $if DBMS_DB_Version.Ver_LE_9_2 $then

    Print('Fallback. Selected in 9.2 and earlier.');
    declare k Index_t := 1; j Index_t := Sparse.First();
    begin
      while j is not null loop
        Dense(k) := Sparse(j);
        j := Sparse.Next(j);
        k := k + 1;
      end loop;
    end;

    forall j in Dense.First..Dense.Last()
      insert into Tbl values Dense(j);

  $else

    Print('Ideal. Selected in 10.1 and later.');
    forall j in indices of Sparse
      insert into tbl values Sparse(j);

  $end
  declare c integer;
  begin
    select Count(*) into c from Tbl;
    Print('"select Count(*) from Tbl" gives '||c);
  end;
  rollback;
end;
/
begin P(); end;
/
```

APPENDIX E:
TRACKING INFORMATION FROM THE BUG DATABASE

Bug #3644582 "PL/SQL conditional compilation is not supported" against *10.2* was filed to support the requests to make the feature available in a *10.1* patchset and in a *9.2* patchset. The report contains this:

auto patchset request #21513 created in bug #3660882 for fix in 10.1.0

and this:

patchset request #21512 created in bug #3660881 for fix in 9.2.0

Bug #3660882 is fixed in *10.1.0.4* and *bug #3660881* is fixed in *9.2.0.6*.

None of these bugs is mentioned in the "bugs fixed" list posted on Metalink for either of these two patchsets:

Metalink Note:283897.1
Bugs fixed in the 9.2.0.6 Patch Set
metalink.oracle.com/metalink/plsql/showdoc?db=NOT&id=283897.1

Metalink Note:295763.1
Bugs fixed in the 10.1.0.4 Patch Set
metalink.oracle.com/metalink/plsql/showdoc?db=NOT&id=295763.1

APPENDIX F: THE NEWTON-RAPHSON FORMULA FOR IMPROVING AN APPROXIMATION FOR THE CUBE ROOT OF A NUMBER

The Newton-Raphson method (also called Newton's method) is a well known root-finding algorithm[81] that uses the first few terms of the Taylor series of a function in the vicinity of a suspected root of a function.

Suppose that the function is $f(x)$. A root is a value of x for which $f(x)$ is zero. Suppose that x_{old} is the current estimate of the root. The following formula gives an improved estimate — x_{new} — of the root:

$$x_{new} = x_{old} - \frac{f(x_{old})}{f'(x_{old})}$$

The following function has its root when x is the real[82] cube root of n.

$$f(x) = x^3 - n$$

Here is its first derivative:

$$f'(x) = 3x^2$$

Substituting $f(x)$ and $f'(x)$ in the formula for the improved estimate of the root gives this:

$$x_{new} = x_{old} - \frac{x_{old}^3 - n}{3x_{old}^2}$$

Finally, this is rearranged for computation thus:

$$x_{new} = \frac{2x_{old}^3 + n}{3x_{old}^2}$$

In the function *Fast_Cbrt()*, *n* is the input formal parameter and *Root* is the local variable which, after sufficient improvement, will be the function's return value. The final formula therefore becomes this PL/SQL statement[83] which is repeated to give an acceptable final approximation:

```
Root := (Two*Root*Root*Root + n)/(Three*Root*Root);
```

---

81. For example, *www.google.com/search?q=%22Newton-Raphson+method%22* leading to *mathworld.wolfram.com/NewtonsMethod.html*

82. The requirements specification for the *Fast_Cbrt()* function demands that it must return only the *real* cube root of its input; the *imaginary* roots are of no interest here.

83. Such computer program versions abound on the internet. The code *fXnext = (2*fX*fX*fX + fPassed)/(3*fX*fX);* is given here: *www.isr.umd.edu/~austin/ence756.d/homework1.html#sec5*

APPENDIX G:
SQL*PLUS SCRIPTS FOR THE CASE STUDY

This appendix lists the complete, self-contained SQL*Plus scripts that provide the basis for the section *Case study: implementing unit testing, assertions, and tracing for a fast cube root body-private helper function* on page 49. Read that section before reading the code. It specifies the requirements for the *Fast_Cbrt()* function and explains the algorithm that is used to implement it.

The code has been formatted for this appendix by first eliding the details (sometimes known as *folding*) and by using hyperlinks progressively to disclose these details. Here is an example:

➪ *page 82*

The scripts are available for download here:

www.oracle.com/technology/tech/pl_sql/files/Fast_Cube_Root_Case_Study.zip

Master_Script.sql invokes further scripts to create an ordinary user, to create all the code objects, and to exercise these under various schemes for conditionalization.

```
-- Master_Script.sql
--
SPOOL Master_Script.txt
PROMPT Master_Script.txt
PROMPT ~~~~~~~~~~~~~~~~~

-- Create a new ordinary user,
-- the Print procedure,
-- and the package specification
@@Setup.sql

-- Now, as the filenames imply...
@@Create_Package_Body.sql
@@Create_Test_Harness.sql
@@Exercise_Test_Harness.sql
SPOOL OFF
```

The units of progressive disclosure have been arranged so that each successive view takes up no more than one page.

```
-- Setup.sql
--
-- You'll need to change the password(s) and the connect string.
-- You might want to change the name "Usr" to avoid conflicts.
--
-- (Quietly) dropping then creating the user documents clearly
-- that it's an "ordinary" user with no required pre-existing
-- objects.
CONNECT System/p@Rel_10_2
begin
  declare
    -- user 'USR' does not exist
    ORA_01918 exception; pragma Exception_Init(ORA_01918, -01918);
  begin
    execute immediate '
      drop user Usr cascade';
  exception when ORA_01918 then null; end;

  execute immediate '
    grant Create Session, Resource to Usr identified by p';
end;
/

-- You might need to start here using, for example, "Scott" instead of "Usr".
-- If so, use "create or replace" rather than "create" in the following.
CONNECT Usr/p@Rel_10_2

-- Print is effectively a "synonym" for DBMS_Output.Put_Line
-- to save line length in the important code.
create procedure Print(V in varchar2) is
begin
  DBMS_Output.Put_Line(V);
end Print;
/

create package Pkg is
  -- This models a to-be-exposed ordinarily subprogram that
  -- uses the helper according to the invariants it assumes.
  procedure Some_Proc;

  -- Not intended for use in production. There, it will raise an exception.
  procedure Run_The_Tests;
end Pkg;
/
```

```
-- Create_Package_Body.sql
--
-- General safety practice. Ensure that no earlier regime upsets the intention.
-- But first save the current PLSQL_CCFlags value.
-- See "Restore the saved PLSQL_CCFlags" at the end.
VARIABLE SAVED_PLSQL_CCFLAGS VARCHAR2(512)
begin :SAVED_PLSQL_CCFLAGS := $$PLSQL_CCFlags; end;
/
alter session set PLSQL_CCFlags = 'Alt:1'
/
create package body Pkg is
  ------------------------------------------------------------------------------
  -- SUBTYPES, TYPES, GLOBAL CONSTANTS, AND SUBPROGRAM FORWARD DECLARATIONS
  ➪ page 76

  ------------------------------------------------------------------------------
  -- CONDITIONALIZED CODE FOR TESTING, ASSERTING, AND TRACING.
  -- Helpers to implement the assertions and to format the trace o/p.
  $if $$Asserting or $$Tracing or $$Testing $then
    ➪ page 77
  $end

  -- Used ONLY when $$Testing is true.
  $if $$Testing $then
    -- The reference standard for accuracy for the datatype.
    -- Used to compare speed and to measure the maximum fractional error
    -- that Fast_Cbrt has (in its advertised range).
    function Slow_Cbrt(n in Math_Real) return Math_Real is
      ➪ page 78
    end Slow_Cbrt;

    procedure Run_The_Tests is
      ➪ page 79
    end Run_The_Tests;

  $else
    procedure Run_The_Tests is
    begin
      Raise_Application_Error(-20000, '~#^*+{obscure}|[]<>?');
    end Run_The_Tests;
  $end

  ------------------------------------------------------------------------------
  -- ORDINARY CODE FOR USE IN THE PRODUCTION DEPLOYMENT
  $if $$Alt = 2 $then
    function Calculated_Thresholds return Math_Reals_t is
      ➪ page 81
    end Calculated_Thresholds;

    function Calculated_Cbrts return Math_Reals_t is
      ➪ page 81
    end Calculated_Cbrts;
  $end

  function Fast_Cbrt(n in Math_Real) return Math_Real is
    ➪ page 82
  end Fast_Cbrt;

  procedure Some_Proc is
    ➪ page 83
  end Some_Proc;
end Pkg;
/
-- Restore the saved PLSQL_CCFlags. This script might fit into a bigger install picture.
declare Quote constant char(1) := '''';
begin
  execute immediate '
    alter session set PLSQL_CCFlags = '||Quote||:SAVED_PLSQL_CCFLAGS||Quote;
end;
/
```

```
-- SUBTYPES, TYPES, GLOBAL CONSTANTS, AND SUBPROGRAM FORWARD DECLARATIONS
subtype Math_Real is $if $$Use_Number $then number;
                     $else                binary_float;
                     $end

function Fast_Cbrt(n in Math_Real) return Math_Real;

-- Ensure no confusion when using literals in overload situations.
Zero                          constant Math_Real :=  0.0;
One                           constant Math_Real :=  1.0;
Two                           constant Math_Real :=  2.0;
Three                         constant Math_Real :=  3.0;
Ten                           constant Math_Real := 10.0;
Twenty                        constant Math_Real := 20.0;
One_Hundred                   constant Math_Real := 100.0;
One_Half                      constant Math_Real := One/Two;
One_Third                     constant Math_Real := One/Three;
Sqrt_Two                      constant Math_Real := Sqrt(Two);

-- The bounds within which the input to Fast_Cbrt() must lie.
Lower_Limit                   constant Math_Real := One;
Upper_Limit                   constant Math_Real := 32767.0;

$if $$Alt = 2 $then
  -- The choice determines the accuracy of the cube root
  -- using just one N-R iteration.
  Number_Of_Thresholds constant pls_integer := 13;
  -- Ideally you'd write type Math_Reals_t is varray(Number_Of_Thresholds)...
  type Math_Reals_t is varray(13) of Math_Real;

  -- These provide sufficiently good starting approximations
  -- that just one N-R iteration is needed.
  function Calculated_Thresholds return Math_Reals_t;
  Thresholds constant Math_Reals_t := Calculated_Thresholds();

  function Calculated_Cbrts return Math_Reals_t;
  Starting_Cbrts constant Math_Reals_t := Calculated_Cbrts();
$end
```

```
-- Helpers to implement the assertions and to format the trace o/p.
$if $$Asserting or $$Tracing or $$Testing $then
  Cubed_Fractional_Error_Limit constant Math_Real :=  0.07;

  -- Support to trace only in every Trace_Step'th call to Fast_Cbrt.
  Tracing_Counter     pls_integer := 0;
  Trace_In_Fast_Cbrt  boolean := false;
  Trace_In_NR         boolean;

  -- To format the trace o/p.
  function Tr(r in Math_Real) return varchar2 is
  begin
    return Lpad(To_Char(r, '99999.999999'), 20);
  end Tr;

  procedure Show_N_And_Root_N(n Math_Real, r in Math_Real) is
  begin
    Print(Tr(n)||Tr(r));
  end Show_N_And_Root_N;

  -- When asserting, check the input to Fast_Cbrt and Slow_Cbrt
  procedure Assert_Valid_Input(Method in varchar2, n in Math_Real) is
  begin
    if n is null then
      Raise_Application_Error(-20000, Method||': Input value is null');
    elsif n > Upper_Limit then
      Raise_Application_Error(-20000, Method||': Input value is too big');
    elsif n < Lower_Limit then
      Raise_Application_Error(-20000, Method||': Input value is too small');
    end if;
  end Assert_Valid_Input;

  -- When asserting, check - in Fast_Cbrt and Slow_Cbrt - that
  -- the result cubed is close enough to the input value.
  procedure Assert_Cbrt_Cubed_Gives_Input(
    Method in varchar2, n in Math_Real, r in Math_Real)
  is
  begin
    if r is null then
      Raise_Application_Error(-20000, Method||': root is null');
    else
      declare
        Cubed_Fractional_Error Math_Real := Abs(r*r*r - n)/n;
      begin
        if Cubed_Fractional_Error > Cubed_Fractional_Error_Limit then
          Show_N_And_Root_N(n, r);
          Raise_Application_Error(-20000,
            Method||': cubed error too big: '||Cubed_Fractional_Error);
        end if;
      end;
    end if;
  end Assert_Cbrt_Cubed_Gives_Input;
$end
```

```
function Slow_Cbrt(n in Math_Real) return Math_Real is
  Root Math_Real;
begin
  $if $$Asserting $then Assert_Valid_Input('Slow_Cbrt', n); $end

  Root := Exp(Ln(n)/Three);

  $if $$Asserting $then Assert_Cbrt_Cubed_Gives_Input('Slow_Cbrt', n, Root); $end
  return Root;
end Slow_Cbrt;
```

```
procedure Run_The_Tests is
  Procedure Rule_Off is
  begin
    Print(Rpad('-', 60, '-'));
  end Rule_Off;

  procedure Time_Slow_And_Fast is
    ⇨ page 80
  end Time_Slow_And_Fast;

  procedure Find_Max_Fractional_Error is
    ⇨ page 80
  end Find_Max_Fractional_Error;
begin
  Rule_Off();
  $if $$Use_Number $then
    Print('Number version.');
  $else
    Print('IEEE version.');
  $end

  -- Fast_Cbrt will cause a compile error if Alt not in (1, 2)
  $if $$Alt = 1 $then
    Print('Crude estimate. Iterate to tolerance.');
  $elsif $$Alt = 2 $then
    Print('Good estimate. Just one iteration.');
  $end

  $if $$Asserting $then
    Print('Assertions turned on.');
  $else
    Print('Assertions turned off.');
  $end

  $if $$Tracing and $$Alt = 2 $then
    for j in 1..Number_Of_Thresholds-1 loop
      Print(Tr(Thresholds(j))||Tr(Starting_Cbrts(j)));
    end loop;
    Print(Tr(Thresholds(Number_Of_Thresholds)));
  $end

  Time_Slow_And_Fast();
  Find_Max_Fractional_Error();

  Print(Chr(10)||'All helper tests for Pkg succeeded.');
  Rule_Off();
end Run_The_Tests;
```

```
procedure Time_Slow_And_Fast is
  Increment constant Math_Real := One/Twenty;
  v Math_Real;
  r Math_Real;
  t0 integer; t1 integer;
  procedure Show_Time(What in varchar2, t in integer) is
  begin
    Print(Rpad(What, 30, '.')||Lpad(t ,4)||' centiseconds');
  end Show_Time;
begin
  v := Lower_Limit;
  t0 := DBMS_Utility.Get_CPU_Time();
  while v <= Upper_Limit loop
    r := Slow_Cbrt(v);
    v := v + Increment;
  end loop;
  t1 := DBMS_Utility.Get_CPU_Time();
  Show_Time('Slow_Cbrt', (t1-t0));

  v := Lower_Limit;
  t0 := DBMS_Utility.Get_CPU_Time();
  while v <= Upper_Limit loop
    r := Fast_Cbrt(v);
    v := v + Increment;
  end loop;
  t1 := DBMS_Utility.Get_CPU_Time();
  Show_Time('Fast_Cbrt', (t1-t0));
end Time_Slow_And_Fast;
```

```
procedure Find_Max_Fractional_Error is
  Increment constant Math_Real := One/Twenty;
  v Math_Real;
  Max_Fractional_Error Math_Real := Zero;
  function Fractional_Error(
    n in Math_Real) return Math_Real
  is
    Fast_R constant Math_Real := Fast_Cbrt(n);
    Slow_R constant Math_Real := Slow_Cbrt(n);
  begin
    return Abs(Fast_R - Slow_R)/Slow_R;
  end Fractional_Error;
begin
  $if $$Tracing $then
    Trace_In_Fast_Cbrt := true;
  $end

  v := Lower_Limit;
  while v <= Upper_Limit loop
    Max_Fractional_Error := Greatest(Max_Fractional_Error, Fractional_Error(v));
    v := v + Increment;
  end loop;

  declare t varchar2(80) := To_Char(Max_Fractional_Error, '0.99999');
  begin
    Print('Max_Fractional_Error:'||Lpad(t, 19));
  end;

  $if $$Tracing $then
    Trace_In_Fast_Cbrt := false;
  $end
end Find_Max_Fractional_Error;
```

```
function Calculated_Thresholds return Math_Reals_t is
  Results  Math_Reals_t := Math_Reals_t();
  No_Of_Cbrts  constant Math_Real := Number_Of_Thresholds-1;

  -- Want the biggest threshold to be just greater than Upper_Limit
  Lim_Plus     constant Math_Real := Upper_Limit + One_Half;
  -- Divide the input range up logarithmically.
  f            constant Math_Real := Exp(Ln(Lim_Plus)/No_Of_Cbrts);
  -- For a safe comparison
  Lim_Minus    constant Math_Real := Upper_Limit - One_Half;
  n            pls_integer := 1;
begin
  Results.Extend(Number_Of_Thresholds);
  Results(n) := Lower_Limit;
  while Results(n) < Lim_Minus loop
    n := n + 1;
    Results(n) := Results(n-1)*f;
  end loop;

  $if $$Asserting $then
    if Results.Count() <> Number_Of_Thresholds then
      Raise_Application_Error(-20000,
        'function Calculated_Thresholds: Logic Error.');
    end if;
  $end
  return Results;
end Calculated_Thresholds;
```

```
function Calculated_Cbrts return Math_Reals_t is
  Results  Math_Reals_t := Math_Reals_t();
begin
  -- There's one fewer starting cube roots than thresholds because
  -- the starting values are in the gaps, derived from
  -- the geom. mean of adjacent thresholds.
  Results.Extend(Number_Of_Thresholds-1);
  for j in 2..Thresholds.Last() loop
    declare
      Geom_Mean constant Math_Real := Sqrt(Thresholds(j-1)*Thresholds(j));
    begin
      -- This is exactly the implementation of Slow_Cbrt.
      -- But Slow_Cbrt, as written with the assertions, is needed
      -- only for unit Testing
      Results(j-1) := Exp(Ln(Geom_Mean)/Three);
    end;
  end loop;

  $if $$Asserting $then
    if Results.Count() <> Number_Of_Thresholds-1 then
      Raise_Application_Error(-20000,
        'function Calculated_Cbrts: Logic Error.');
    end if;
  $end
  return Results;
end Calculated_Cbrts;
```

```
function Fast_Cbrt(n in Math_Real) return Math_Real is
  Root Math_Real := null;
begin
  $if $$Asserting $then Assert_Valid_Input('Fast_Cbrt', n); $end

  -- Starting Approximation.
  $if $$Alt = 1 $then
    Root := n/One_Hundred;

  $elsif $$Alt = 2 $then
    declare
      Idx pls_integer := Number_Of_Thresholds;
    begin
      while Idx > 0 loop
        if n >= Thresholds(Idx) then
          Root := Starting_Cbrts(Idx);
          exit;
        end if;
        Idx := Idx - 1;
      end loop;
    end;
    $if $$Asserting $then
     if Root is null then Raise_Application_Error(-20000,
       'Fast_Cbrt: Starting Approximation is null');
     end if;
    $end

  $else
    $error 'Alt must be 1 or 2' $end
  $end

  $if $$Tracing $then
    if Trace_In_Fast_Cbrt then
      Tracing_Counter := Tracing_Counter + 1;
      if Remainder(Tracing_Counter, $$Trace_Step) = 0 then
        Print(null);
        Show_N_And_Root_N(n, Root);
        Trace_In_NR := true;
      else
        Trace_In_NR := false;
      end if;
    end if;
  $end

  -- Newton-Raphson improvement.
  $if $$Alt = 1 $then
    -- The classic convergence test.
    declare
      Tolerance constant Math_Real :=  0.01;
      Last_Root Math_Real := n;
    begin
      while Abs(Root - Last_Root)/Root > Tolerance loop
        Last_Root := Root;
        Root := (Two*Root*Root*Root + n)/(Three*Root*Root);
        $if $$Tracing $then
          if Trace_In_Fast_Cbrt and Trace_In_NR then
            Print(Lpad(' ', 20)||Tr(Root));
          end if;
        $end
      end loop;
    end;

  $elsif $$Alt = 2 $then
    -- It is now sufficient to iterate just ONCE because the starting approxn is so good.
    Root := (Two*Root*Root*Root + n)/(Three*Root*Root);
    $if $$Tracing $then
      if Trace_In_Fast_Cbrt and Trace_In_NR then
        Print(Lpad(' ', 20)||Tr(Root));
      end if;
    $end
  $end

  $if $$Asserting $then Assert_Cbrt_Cubed_Gives_Input('Fast_Cbrt', n, Root); $end
  return Root;
end Fast_Cbrt;
```

```
procedure Some_Proc is
  n1           constant Math_Real := 200;
  n2           constant Math_Real := 300;
  s            Math_Real;
  Expected_s   constant Math_Real := 12.5;
  Epsilon      constant Math_Real :=  0.03;
begin
  -- Notice that n1 and n2 are inside the supported range for Fast_Cbrt.
  if n1 < Lower_Limit or
     n2 < Lower_Limit or
     n1 > Upper_Limit or
     n2 > Upper_Limit
  then Raise_Application_Error(-20000, 'Some_Proc: Invariants broken.'); end if;

  s := Fast_Cbrt(n1) + Fast_Cbrt(n2);
  if s is null or Abs(s - Expected_s)/Expected_s > Epsilon then
    Raise_Application_Error(-20000,
      'wrong s: '||To_Char(s, '999.999999999'));
  end if;
  Print('Some_Proc finished OK.');
end Some_Proc;
```

```
-- Create_Test_Harness.sql
--
-- Simple test harness to show the use of the ordinary api and the testing api
create procedure Test_It is
begin
  begin
    Pkg.Run_The_Tests();
  exception when others then
    if DBMS_Utility.Format_Error_Stack() like
     '%~#^*+{obscure}|[]<>?%'
    then
      Print('Run_The_Tests() is disabled for production deployment.');
    else
      raise;
    end if;
  end;
  Pkg.Some_Proc();
end Test_It;
/
```

```
-- Exercise_Test_Harness.sql
--
-- Test sequence. Exercise with various values for the ccflags.
-------------------------------------------------------------------------------
-- IEEE version: Production configuration.
-- Ultimately, the code for Alt=1 and all reference to $$Alt will be removed.
alter package Pkg compile body PLSQL_CCFlags = 'Alt:2' reuse settings
/
begin Test_It(); end;
/



-------------------------------------------------------------------------------
-- Number version, Alt 1
-- Full testing, with assertions and tracing
alter package Pkg compile body PLSQL_CCFlags = '
    Use_Number:true,    Testing:true,
    Alt:1,              Asserting:true,
                        Tracing:true,
                        Trace_Step:65536'
  reuse settings
/
begin Test_It(); end;
/
```

```
-- Number version, Alt 1
-- Assertions but no tracing
alter package Pkg compile body PLSQL_CCFlags = '
    Use_Number:true,    Testing:true,
    Alt:1,              Asserting:true,
                        Tracing:false,
                        Trace_Step:null'
  reuse settings
/
begin Test_It(); end;
/


-------------------------------------------------------------------------------
-- Number version, Alt 2
-- Assertions but no tracing
alter package Pkg compile body PLSQL_CCFlags = '
    Use_Number:true,    Testing:true,
    Alt:2,              Asserting:true,
                        Tracing:false,
                        Trace_Step:null'
  reuse settings
/
begin Test_It(); end;
/


-------------------------------------------------------------------------------
-- Number version, Alt 1
-- For timing: no assertions or tracing
alter package Pkg compile body PLSQL_CCFlags = '
    Use_Number:true,    Testing:true,
    Alt:1,              Asserting:false,
                        Tracing:false,
                        Trace_Step:null'
  reuse settings
/
begin Test_It(); end;
/


-------------------------------------------------------------------------------
-- Number version, Alt 2
-- For timing: no assertions or tracing
alter package Pkg compile body PLSQL_CCFlags = '
    Use_Number:true,    Testing:true,
    Alt:2,              Asserting:false,
                        Tracing:false,
                        Trace_Step:null'
  reuse settings
/
begin Test_It(); end;
/
```

```
-- IEEE version, Alt 1
-- Assertions but no tracing
alter package Pkg compile body PLSQL_CCFlags = '
    Use_Number:false,   Testing:true,
    Alt:1,              Asserting:true,
                        Tracing:false,
                        Trace_Step:null'
  reuse settings
/
begin Test_It(); end;
/


-------------------------------------------------------------------------------
-- IEEE version, Alt 2
-- Assertions but no tracing
alter package Pkg compile body PLSQL_CCFlags = '
    Use_Number:false,   Testing:true,
    Alt:2,              Asserting:true,
                        Tracing:false,
                        Trace_Step:null'
  reuse settings
/
begin Test_It(); end;
/


-------------------------------------------------------------------------------
-- IEEE version, Alt 1
-- For timing: no assertions or tracing
alter package Pkg compile body PLSQL_CCFlags = '
    Use_Number:false,   Testing:true,
    Alt:1,              Asserting:false,
                        Tracing:false,
                        Trace_Step:null'
  reuse settings
/
begin Test_It(); end;
/


-------------------------------------------------------------------------------
-- IEEE version, Alt 2
-- For timing: no assertions or tracing
alter package Pkg compile body PLSQL_CCFlags = '
    Use_Number:false,   Testing:true,
    Alt:2,              Asserting:false,
                        Tracing:false,
                        Trace_Step:null'
  reuse settings
/
begin Test_It(); end;
/
```

```
Master_Script.txt
~~~~~~~~~~~~~~~~~

Run_The_Tests() is disabled for production deployment.
Some_Proc finished OK.



------------------------------------------------------------
Number version.
Crude estimate. Iterate to tolerance.
Assertions turned on.
Slow_Cbrt.....................6825 centiseconds
Fast_Cbrt.....................3203 centiseconds

         3277.750000           32.777500
                               22.868624
                               17.334923
                               15.192505
                               14.861983
                               14.854523

         6554.550000           65.545500
                               44.205553
                               30.588436
                               22.727399
                               19.381422
                               18.737296
                               18.714645

         9831.350000           98.313500
                               65.881385
                               44.675957
                               31.425862
                               24.268889
                               21.743323
                               21.427254
                               21.422545

        13108.150000          131.081500
                               87.641961
                               58.996822
                               40.586559
                               29.710208
                               24.756846
                               23.633579
                               23.578500

        16384.950000          163.849500
                              109.436439
                               73.413663
                               49.955817
                               35.492404
                               27.997241
                               25.632591
                               25.401032

        19661.750000          196.617500
                              131.247867
                               87.879045
                               59.434683
                               41.478451
                               31.461695
                               27.595663
                               27.003464
                               26.990286

        22938.550000          229.385500
                              153.068982
                              102.372328
                               68.977810
                               47.592244
                               35.103929
                               29.607497
                               28.460838
                               28.413400
```

```
        26215.350000          262.153500
                              174.896152
                              116.883111
                               78.561707
                               53.790306
                               38.880339
                               31.700845
                               29.829360
                               29.707032

        29492.150000          294.921500
                              196.727358
                              131.405585
                               88.173045
                               60.046514
                               42.757535
                               33.882269
                               31.151461
                               30.898081
Max_Fractional_Error:            0.00010

All helper tests for Pkg succeeded.
------------------------------------------------------------
Some_Proc finished OK.
```

```
------------------------------------------------------------
Number version.
Crude estimate. Iterate to tolerance.
Assertions turned on.
Slow_Cbrt.....................7007 centiseconds
Fast_Cbrt.....................3187 centiseconds
Max_Fractional_Error:            0.00010

All helper tests for Pkg succeeded.
------------------------------------------------------------
Some_Proc finished OK.




------------------------------------------------------------
Number version.
Good estimate. Just one iteration.
Assertions turned on.
Slow_Cbrt.....................6822 centiseconds
Fast_Cbrt..................... 533 centiseconds
Max_Fractional_Error:            0.02197

All helper tests for Pkg succeeded.
------------------------------------------------------------
Some_Proc finished OK.




------------------------------------------------------------
Number version.
Crude estimate. Iterate to tolerance.
Assertions turned off.
Slow_Cbrt.....................6493 centiseconds
Fast_Cbrt.....................2914 centiseconds
Max_Fractional_Error:            0.00010

All helper tests for Pkg succeeded.
------------------------------------------------------------
Some_Proc finished OK.




------------------------------------------------------------
Number version.
Good estimate. Just one iteration.
Assertions turned off.
Slow_Cbrt.....................6518 centiseconds
Fast_Cbrt..................... 320 centiseconds
Max_Fractional_Error:            0.02197

All helper tests for Pkg succeeded.
------------------------------------------------------------
Some_Proc finished OK.
```

```
------------------------------------------------------------
IEEE version.
Crude estimate. Iterate to tolerance.
Assertions turned on.
Slow_Cbrt..................... 190 centiseconds
Fast_Cbrt..................... 166 centiseconds
Max_Fractional_Error:            0.00010

All helper tests for Pkg succeeded.
------------------------------------------------------------
Some_Proc finished OK.




------------------------------------------------------------
IEEE version.
Good estimate. Just one iteration.
Assertions turned on.
Slow_Cbrt..................... 189 centiseconds
Fast_Cbrt.....................  64 centiseconds
Max_Fractional_Error:            0.02197

All helper tests for Pkg succeeded.
------------------------------------------------------------
Some_Proc finished OK.




------------------------------------------------------------
IEEE version.
Crude estimate. Iterate to tolerance.
Assertions turned off.
Slow_Cbrt..................... 108 centiseconds
Fast_Cbrt..................... 128 centiseconds
Max_Fractional_Error:            0.00010

All helper tests for Pkg succeeded.
------------------------------------------------------------
Some_Proc finished OK.




------------------------------------------------------------
IEEE version.
Good estimate. Just one iteration.
Assertions turned off.
Slow_Cbrt..................... 107 centiseconds
Fast_Cbrt.....................  42 centiseconds
Max_Fractional_Error:            0.02197

All helper tests for Pkg succeeded.
------------------------------------------------------------
Some_Proc finished OK.
```

PL/SQL conditional compilation
October 2005
Bryn Llewellyn, PL/SQL Product Manager, Oracle Headquarters

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
oracle.com