PART III

PL/SQL Program Data

Just about every program you write will manipulate data—and much of that data is *local to* (i.e., defined in) your PL/SQL procedure or function. This part of the book concentrates on the various types of program data you can define in PL/SQL, such as numbers (including the datatypes introduced in Oracle Database 11*g*), strings, dates, timestamps, records, collections, XML datatypes, and user-defined datatypes. Chapter 7 through Chapter 13 also cover the various built-in functions provided by Oracle that allow you to manipulate and modify data.

CHAPTER 7

Working with Program Data

Almost every PL/SQL block you write will define and manipulate *program data*. Program data consists of data structures that exist only within your PL/SQL session (physically, within the Program Global Area, or PGA, for your session); they are not stored in the database. Program data can be:

Variable or constant
: The values of variables can change during a program's execution. The values of constants are static once they are set at the time of declaration.

Scalar or composite
: Scalars are made up of a single value, such as a number or a string. Composite data consists of multiple values, such as a record, a collection, or an object type instance.

Containerized
: Containers may contain information obtained from the database, or data that was never in the database and might not ever end up there.

Before you can work with program data inside your PL/SQL code, you must declare data structures, giving them names and datatypes.

This chapter describes how you declare program data. It covers the rules governing the format of the names you give your data structures. It offers a quick reference to all the different types of data supported in PL/SQL and explores the concept of datatype conversion. The chapter finishes with some recommendations for working with program data. The remaining chapters in this part of the book describe specific types of program data.

Naming Your Program Data

To work with a variable or a constant, you must first declare it, and when you declare it, you give it a name. Here are the rules that PL/SQL insists you follow when naming your data structures (these are the same rules applied to names of database objects, such as tables and columns):

- Names can be up to 30 characters in length.
- Names must start with a letter.
- After the first letter, names can be composed of any of the following: letters, numerals, $, #, and _.
- All names are case insensitive (unless those names are placed within double quotes).

Given these rules, the following names are valid:

```
l_total_count
first_12_years
total_#_of_trees
salary_in_$
```

These next two names are not only valid but considered identical by PL/SQL because it is *not* a case-sensitive language:

```
ExpertsExchange
ExpertSexChange
```

The next three names are invalid, for the reasons indicated:

```
1st_account  -- Starts with a number instead of a letter
favorite_ice_cream_flavors_that_dont_contain_nuts  -- Too long
email_address@business_loc  -- Contains an invalid character (@)
```

There are some exceptions to these rules (why am I not surprised?). If you embed a name within double quotes when you declare it, you can bypass all the aforementioned rules *except* the maximum length of 30 characters. For example, all of the following declarations are valid:

```
DECLARE
   "truly_lower_case" INTEGER;
   "     " DATE; -- Yes, a name consisting of five spaces!
   "123_go!" VARCHAR2(10);
BEGIN
   "123_go!"  := 'Steven';
END;
```

Note that when you reference these strange names in your execution section, you will need to do so within double quotes, as shown. Otherwise, your code will not compile.

Why would you use double quotes? There is little reason to do so in PL/SQL programs. It is a technique programmers sometimes employ when creating database objects because it preserves case sensitivity (in other words, if I CREATE TABLE "docs", then the name of the table is docs and not DOCS), but in general, you should avoid using double quotes in PL/SQL.

Another exception to these naming conventions has to do with the names of Java objects, which can be up to 4,000 characters in length. See the Java chapter included on the book's website for more details about this variation and what it means for PL/SQL developers.

Here are two key recommendations for naming your variables, constants, and types:

Ensure that each name accurately reflects its usage and is understandable at a glance
: You might even take a moment to write down—in noncomputer terms—what a variable represents. You can then easily extract an appropriate name from that statement. For example, if a variable represents the "total number of calls made about lukewarm coffee," a good name for that variable might be total_calls_on_cold_coffee, or tot_cold_calls, if you are allergic to five-word variable names. Bad names for that variable would be totcoffee or t_#_calls_lwcoff, both of which are too cryptic to get the point across.

Establish consistent, sensible naming conventions
: Such conventions usually involve the use of prefixes and/or suffixes to indicate type and usage. For example, all local variables should be prefixed with "l_", while global variables defined in packages have a "g_" prefix. All record types should have a suffix of "_rt", and so on. A comprehensive set of naming conventions is available for download with the example code for my book Oracle PL/SQL Best Practices.

Overview of PL/SQL Datatypes

Whenever you declare a variable or a constant, you must assign it a datatype. PL/SQL is, with very few exceptions, a "statically typed programming language" (see the following sidebar for a definition). PL/SQL offers a comprehensive set of predefined scalar and composite datatypes, and you can create your own user-defined types (also known as *abstract datatypes*). Database columns do not support many of the PL/SQL datatypes, such as Boolean and NATURAL, but within PL/SQL code these datatypes are quite useful.

Virtually all of these predefined datatypes are defined in the PL/SQL STANDARD package. Here, for example, are the statements that define the Boolean datatype and two of the numeric datatypes:

```
create or replace package STANDARD is

  type BOOLEAN is (FALSE, TRUE);

  type NUMBER is NUMBER_BASE;
  subtype INTEGER is NUMBER(38,);
```

When it comes to datatypes, PL/SQL supports the usual suspects and a whole lot more. This section provides a quick overview of the various predefined datatypes. They are covered in detail in Chapter 8 through Chapter 13, Chapter 15, and Chapter 26; you will find detailed references to specific chapters in the following sections.

What Does "Static Typing" Mean?

A programming language uses *static* typing, also called *strong* typing, if type checking is performed at compile time as opposed to at runtime. Some programming languages that use static typing include PL/SQL, Ada, C, and Pascal. A *dynamically* typed programming language—like JavaScript, Perl, or Ruby—performs most type checking at runtime. Static typing can find type errors at compile time, which can increase the reliability of the program and make it execute faster. An optimizing compiler that knows the exact datatypes in use can find assembler shortcuts more easily and produce more highly optimized machine code. Dynamic typing also has some advantages, though: for example, metaclasses and introspection are easier to implement with dynamic typing.

Character Data

PL/SQL supports both fixed- and variable-length strings as both traditional character and Unicode character data. CHAR and NCHAR are fixed-length datatypes; VARCHAR2 and NVARCHAR2 are variable-length datatypes. Here is a declaration of a variable-length string that can hold up to 2,000 characters:

```
DECLARE
   l_accident_description VARCHAR2(2000);
```

Chapter 8 explores the rules for character data, provides many examples, and explains the built-in functions provided to manipulate strings in PL/SQL.

For very large character strings, PL/SQL has the CLOB (character large object) and NCLOB (National Language Support CLOB) datatypes. For backward compatibility, PL/SQL also supports the LONG datatype. These datatypes allow you to store and manipulate very large amounts of data; in Oracle Database 11*g*, a LOB can hold up to 128 terabytes of information.

There are many rules restricting the use of LONGs. I recommend that you avoid using LONGs (assuming that you are running Oracle8 Database or later).

Chapter 13 explores the rules for large objects, provides many examples, and explains the built-in functions and the DBMS_LOB package provided to manipulate large objects in PL/SQL.

Numbers

PL/SQL supports an increasing variety of numeric datatypes. NUMBER has long been the workhorse of the numeric datatypes, and you can use it for decimal fixed- and floating-point values, and for integers. Following is an example of some typical NUMBER declarations:

```
/* File on web: numbers.sql */
DECLARE
   salary NUMBER(9,2); -- fixed-point, seven to the left, two to the right
   raise_factor NUMBER; -- decimal floating-point
   weeks_to_pay NUMBER(2); -- integer
BEGIN
   salary := 1234567.89;
   raise_factor := 0.05;
   weeks_to_pay := 52;
END;
```

Because of its internal decimal nature, NUMBER is particularly useful when you're working with monetary amounts, as you won't incur any rounding error as a result of binary representation. For example, when you store 0.95, you won't come back later to find only 0.949999968.

Prior to Oracle Database 10*g*, NUMBER was the only one of PL/SQL's numeric datatypes to correspond directly to a database datatype. You can see this subtyping by examining the package STANDARD. This exclusiveness is one reason you'll find NUMBER so widely used in PL/SQL programs.

Oracle Database 10*g* introduced two binary floating-point types: BINARY_FLOAT and BINARY_DOUBLE. Like NUMBER, these binary datatypes are supported in both PL/SQL and the database. Unlike NUMBER, these binary datatypes are not decimal in nature—they have binary precision—so you can expect rounding. The BINARY_FLOAT and BINARY_DOUBLE types support the special values NaN (not a number) and positive and negative infinity. Given the right type of application, their use can lead to tremendous performance gains, as arithmetic involving these binary types is performed in hardware whenever the underlying platform allows.

Oracle Database 11*g* added two more variations on these floating-point types. SIMPLE_FLOAT and SIMPLE_DOUBLE are like BINARY_FLOAT and BINARY_DOUBLE, but they do not allow NULL values, nor do they raise an exception when an overflow occurs.

PL/SQL also supports several numeric types and subtypes that do not correspond to database datatypes, but are nevertheless quite useful. Notable here are PLS_INTEGER and SIMPLE_INTEGER. PLS_INTEGER is an integer type with its arithmetic implemented in hardware. FOR loop counters are implemented as PLS_INTEGERs. SIMPLE_INTEGER, introduced in Oracle Database 11*g*, has the same range of values as PLS_INTEGER, but it does not allow NULL values, nor does it raise an exception when an overflow occurs. SIMPLE_INTEGER, like SIMPLE_FLOAT and SIMPLE_DOUBLE, is extremely speedy—especially with natively compiled code. I've measured stunning performance improvements using SIMPLE_INTEGER compared to other numeric datatypes.

Chapter 9 explores the rules for numeric data, provides many examples, and explains the built-in functions provided to manipulate numbers in PL/SQL.

Dates, Timestamps, and Intervals

Prior to Oracle9*i* Database, the Oracle world of dates was limited to the DATE datatype, which stores both a date and a time (down to the nearest second). Oracle9*i* Database introduced two sets of new, related datatypes: INTERVALs and TIMESTAMPs. These datatypes greatly expand the capability of PL/SQL developers to write programs that manipulate and store dates and times with very high granularity, and also compute and store intervals of time.

Here is an example of a function that computes the age of a person as an interval with month granularity:

```
/* File on web: age.fnc */
FUNCTION age (dob_in IN DATE)
   RETURN INTERVAL YEAR TO MONTH
IS
BEGIN
   RETURN (SYSDATE - dob_in) YEAR TO MONTH;
END;
```

Chapter 10 explores the rules for date-related data, provides many examples, and explains the built-in functions provided to manipulate dates, timestamps, and intervals in PL/SQL.

Booleans

PL/SQL supports a three-value Boolean datatype. A variable of this type can have one of only three values: TRUE, FALSE, and NULL.

Booleans help us write very readable code, especially involving complex logical expressions. Here's an example of a Boolean declaration, along with an assignment of a default value to that variable:

```
DECLARE
   l_eligible_for_discount BOOLEAN :=
      customer_in.balance > min_balance AND
      customer_in.pref_type = 'MOST FAVORED' AND
      customer_in.disc_eligibility;
```

Chapter 13 explores the rules for Boolean data and provides examples of usage.

Binary Data

Oracle supports several forms of *binary data* (unstructured data that is not interpreted or processed by Oracle), including RAW, BLOB, and BFILE. The BFILE datatype stores unstructured binary data in operating system files outside the database. RAW is a variable-length datatype like the VARCHAR2 character datatype, except that Oracle utilities do not perform character set conversion when transmitting RAW data.

The datatype LONG RAW is still supported for backward compatibility, but PL/SQL offers only limited support for LONG RAW data. In an Oracle database, a LONG RAW column can be up to 2 GB long, but PL/SQL will only be able to access the first 32,760 bytes of a LONG RAW. If, for example, you try to fetch a LONG RAW from the database into your PL/SQL variable that exceeds the 32,760 byte limit, you will encounter an *ORA-06502: PL/SQL: numeric or value error* exception. To work with LONG RAWs longer than PL/SQL's limit, you need an OCI program; this is a good reason to migrate your legacy code from LONG RAWs to BLOBs, which have no such limit.

Chapter 13 explores the rules for binary data, provides many examples, and explains the built-in functions and the DBMS_LOB package provided to manipulate BFILEs and other binary data in PL/SQL.

ROWIDs

Oracle provides two proprietary datatypes, ROWID and UROWID, used to represent the address of a row in a table. ROWID represents the unique physical address of a row in its table; UROWID represents the logical position of a row in an index-organized table (IOT). ROWID is also a SQL pseudocolumn that can be included in SQL statements.

Chapter 13 explores the rules for working with the ROWID and UROWID datatypes.

REF CURSORs

The REF CURSOR datatype allows developers to declare cursor variables. A cursor variable can then be used with static or dynamic SQL statements to implement more flexible programs. There are two forms of REF CURSORs: the strong REF CURSOR and the weak REF CURSOR. PL/SQL is a statically typed language, and the weak REF CURSOR is one of the few dynamically typed constructs supported.

Here is an example of a strong REF CURSOR declaration. I associate the cursor variable with a specific record structure (using a %ROWTYPE attribute):

```
DECLARE
   TYPE book_data_t IS REF CURSOR RETURN book%ROWTYPE;
   book_curs_var book_data_t;
```

And here are two weak REF CURSOR declarations in which I do not associate any particular structure with the resulting variable. The second declaration (the last line) showcases SYS_REFCURSOR, a predefined weak REF CURSOR type:

```
DECLARE
   TYPE book_data_t IS REF CURSOR;
   book_curs_var book_data_t;
   book_curs_var_b SYS_REFCURSOR;
```

Chapter 15 explores REF CURSORs and cursor variables in much more detail.

Internet Datatypes

Beginning with Oracle 9*i* Database, there is native support for several Internet-related technologies and types of data, specifically XML (Extensible Markup Language) and URIs (universal resource identifiers). The Oracle database provides datatypes for handling XML and URI data, as well as a class of URIs called *DBUri-REFs* that access data stored within the database itself. The database also includes a set of datatypes used to store and access both external and internal URIs from within the database.

The XMLType allows you to query and store XML data in the database using functions like SYS_XMLGEN and the DBMS_XMLGEN package. It also allows you to use native operators in the SQL language to search XML documents using the XPath language.

The URI-related types, including URIType and HttpURIType, are all part of an object type inheritance hierarchy and can be used to store URLs to external web pages and files, as well as to refer to data within the database.

Chapter 13 explores the rules for working with XMLType and URI types, provides some examples, and explains the built-in functions and packages provided to manipulate these datatypes.

“Any” Datatypes

Most of the time, our programming tasks are fairly straightforward and very specific to the requirement at hand. At other times, however, we write more generic code. For those situations, the Any datatypes might come in very handy.

The Any types were introduced in Oracle9*i* Database and are very different from any other kind of datatype available in an Oracle database. They let you dynamically encapsulate and access type descriptions, data instances, and sets of data instances of any other SQL type. You can use these types (and the methods defined for them, as they are object types) to do things like determine the type of data stored in a particular nested table without having access to the actual declaration of that table type!

The Any datatypes include AnyType, AnyData, and AnyDataSet.

Chapter 13 explores the rules for working with the Any datatypes and provides some working examples of these dynamic datatypes.

User-Defined Datatypes

You can use Oracle built-in datatypes and other user-defined datatypes to create arbitrarily complex types of your own that model closely the structure and behavior of data in your systems.

Chapter 26 explores this powerful feature in more detail and describes how to take advantage of the support for object type inheritance in Oracle9*i* Database through Oracle Database 11*g*.

Declaring Program Data

With few exceptions, you must declare your variables and constants before you use them. These declarations are in the declaration section of your PL/SQL program. (See Chapter 3 for more details on the structure of the PL/SQL block and its declaration section.)

Your declarations can include variables, constants, TYPEs (such as collection types or record types), and exceptions. This chapter focuses on the declarations of variables and constants. (See Chapter 11 for an explanation of TYPE statements for records and Chapter 12 for collection types. See Chapter 6 to learn how to declare exceptions.)

Declaring a Variable

When you declare a variable, PL/SQL allocates memory for the variable's value and names the storage location so that the value can be retrieved and changed. The declaration also specifies the datatype of the variable; this datatype is then used to validate values assigned to the variable.

The basic syntax for a declaration is:

```
name datatype [NOT NULL] [ := | DEFAULT default_assignment];
```

where *name* is the name of the variable or constant to be declared, and *datatype* is the datatype or subtype that determines the type of data that can be assigned to the variable. You can include a NOT NULL clause, which tells the database to raise an exception if no value is assigned to this variable. The *default_assignment* clause tells the database to initialize the variable with a value; this is optional for all declarations except those of constants. If you declare a variable NOT NULL, you must assign a value to it in the declaration line.

The following examples illustrate declarations of variables of different datatypes:

```
DECLARE
   -- Simple declaration of numeric variable
   l_total_count NUMBER;

   -- Declaration of number that rounds to nearest hundredth (cent):
   l_dollar_amount NUMBER (10,2);

   -- A single datetime value, assigned a default value of the database
   -- server's system clock. Also, it can never be NULL.
   l_right_now DATE NOT NULL  DEFAULT SYSDATE;

   -- Using the assignment operator for the default value specification
   l_favorite_flavor VARCHAR2(100) := 'Anything with chocolate, actually';

   -- Two-step declaration process for associative array.
   -- First, the type of table:
   TYPE list_of_books_t IS TABLE OF book%ROWTYPE INDEX BY BINARY_INTEGER;

   -- And now the specific list to be manipulated in this block:
   oreilly_oracle_books list_of_books_t;
```

The DEFAULT syntax (see l_right_now in the previous example) and the assignment operator syntax (see l_favorite_flavor in the previous example) are equivalent and can be used interchangeably. So which should you use? I like to use the assignment operator (:=) to set default values for constants, and the DEFAULT syntax for variables. In the case of a constant, the assigned value is not really a default but an initial (and unchanging) value, so the DEFAULT syntax feels misleading to me.

Declaring Constants

There are just two differences between declaring a variable and declaring a constant: for a constant, you include the CONSTANT keyword, and you must supply a default value (which isn't really a *default* at all, but rather is the *only* value). So, the syntax for the declaration of a constant is:

```
name CONSTANT datatype [NOT NULL] := | DEFAULT default_value;
```

The value of a constant is set upon declaration and may not change thereafter.

Here are some examples of declarations of constants:

```
DECLARE
   -- The current year number; it's not going to change during my session.
   l_curr_year CONSTANT PLS_INTEGER :=
      TO_NUMBER (TO_CHAR (SYSDATE, 'YYYY'));

   -- Using the DEFAULT keyword
   l_author CONSTANT VARCHAR2(100) DEFAULT 'Bill Pribyl';

   -- Declare a complex datatype as a constant-
   -- this isn't just for scalars!
   l_steven CONSTANT  person_ot :=
     person_ot ('HUMAN', 'Steven Feuerstein', 175,
                TO_DATE ('09-23-1958', 'MM-DD-YYYY') );
```

Unless otherwise stated, the information provided in the rest of this chapter for variables also applies to constants.

> An unnamed constant is a literal value, such as 2 or 'Bobby McGee'. A literal does not have a name, although it does have an implied (undeclared) datatype.

The NOT NULL Clause

If you do assign a default value, you can also specify that the variable must be NOT NULL. For example, the following declaration initializes the company_name variable to 'PCS R US' and makes sure that the name can never be set to NULL:

```
company_name VARCHAR2(60) NOT NULL DEFAULT 'PCS R US';
```

If your code executes a line like this:

```
company_name := NULL;
```

then PL/SQL will raise the VALUE_ERROR exception. In addition, you will receive a compilation error with this next declaration, because the declaration does not include an initial or default value:

```
company_name VARCHAR2(60) NOT NULL; -- must assign a value if declared NOT NULL!
```

Anchored Declarations

You can and often will declare variables using "hardcoded" or explicit datatypes, as follows:

```
l_company_name VARCHAR2(100);
```

A better practice for data destined for or obtained from a database table or other PL/SQL program structure is to *anchor* your variable declaration to that object. When you anchor a datatype, you tell PL/SQL to set the datatype of your variable to the datatype of an already defined data structure: another PL/SQL variable, a predefined TYPE or SUBTYPE, a database table, or a specific column in a table.

PL/SQL offers two kinds of anchoring:

Scalar anchoring
: Use the %TYPE attribute to define your variable based on a table's column or some other PL/SQL scalar variable.

Record anchoring
: Use the %ROWTYPE attribute to define your record structure based on a table or a predefined PL/SQL explicit cursor.

The syntax for an anchored datatype is:

```
variable_name type_attribute%TYPE [optional_default_value_assignment];
variable_name table_name | cursor_name%ROWTYPE [optional_default_value_assignment];
```

where *variable_name* is the name of the variable you are declaring, and *type_attribute* is either a previously declared PL/SQL variable name or a table column specification in the format *table.column*.

This anchoring reference is resolved at the time the code is compiled; there is no runtime overhead to anchoring. The anchor also establishes a dependency between the code and the anchored element (the table, cursor, or package containing the variable referenced). This means that if those elements are changed, the code in which the anchoring takes place is marked INVALID. When it is recompiled, the anchor will again be resolved, thereby keeping the code current with the anchored element.

Figure 7-1 shows how the datatype is drawn from both a database table and a PL/ SQL variable.

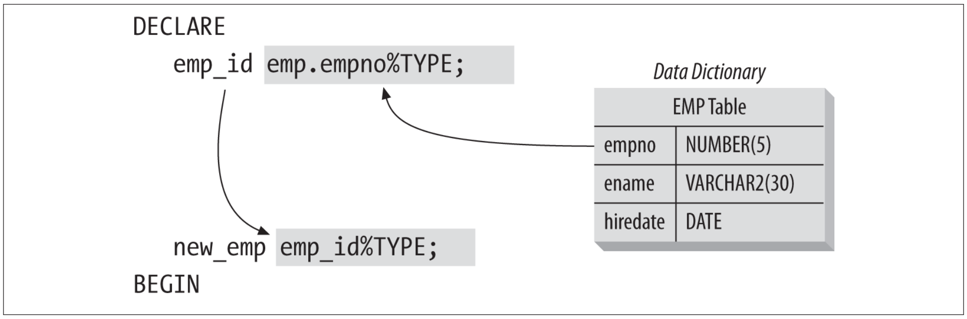


Figure 7-1. Anchored declarations with %TYPE

Here is an example of anchoring a variable to a database column:

```
l_company_id company.company_id%TYPE;
```

You can also anchor against PL/SQL variables; this is usually done to avoid redundant declarations of the same hardcoded datatype. In this case, the best practice is to create a "reference" variable in a package and then reference that package variable in %TYPE statements. (You could also create SUBTYPEs in your package; this topic is covered later in the chapter.) The following example shows just a portion of a package intended to make it easier to work with Oracle Advanced Queuing (AQ):

```
/* File on web: aq.pkg */
PACKAGE aq
IS

/* Standard datatypes for use with Oracle AQ. */
   v_msgid                RAW (16);
   SUBTYPE msgid_type IS v_msgid%TYPE;
   v_name                 VARCHAR2 (49);
   SUBTYPE name_type IS v_name%TYPE;
   ...
END aq;
```

AQ message IDs are of type RAW(16). Rather than have to remember that (and hardcode it into my application again and again), I can simply declare an AQ message ID as follows:

```
DECLARE
  my_msg_id aq.msgid_type;
BEGIN
```

Then, if the database ever changes its datatype for a message ID, I can change the SUBTYPE definition in the AQ package, and all declarations will be updated with the next recompilation.

Anchored declarations provide an excellent illustration of the fact that PL/SQL is not just a procedural-style programming language, but was designed specifically as an extension to the Oracle SQL language. Oracle Corporation made a very thorough effort to tightly integrate the programming constructs of PL/SQL to the underlying SQL database.

Anchored declarations offer some important benefits when it comes to writing applications that adapt easily to change over time.

Anchoring to Cursors and Tables

You've seen an example of anchoring to a database column and to another PL/SQL variable. Now let's take a look at the use of the %ROWTYPE anchoring attribute.

Suppose that I want to query a single row of information from the book table. Rather than declare individual variables for each column in the table (which, of course, I should do with %TYPE), I can simply rely on %ROWTYPE:

```
DECLARE
   l_book book%ROWTYPE;
BEGIN
   SELECT * INTO l_book
      FROM book
    WHERE isbn = '1-56592-335-9';
   process_book (l_book);
END;
```

Suppose now that I only want to retrieve the author and title from the book table. In this case, I build an explicit cursor and then %ROWTYPE against that cursor:

```
DECLARE
   CURSOR book_cur IS
      SELECT author, title FROM book
       WHERE  isbn = '1-56592-335-9';
   l_book book_cur%ROWTYPE;
BEGIN
   OPEN book_cur;
   FETCH book_cur INTO l_book;  END;
```

Finally, here is an example of an *implicit* use of the %ROWTYPE declaration: the cursor FOR loop.

```
BEGIN
   FOR book_rec IN (SELECT * FROM book)
   LOOP
      process_book (book_rec);
   END LOOP;
END;
```

Now let's explore some of the benefits of anchored declarations.

Benefits of Anchored Declarations

Most of the declarations you have seen so far—character, numeric, date, Boolean—specify explicitly the type of data for the variable being declared. In each of these cases, the declaration contains a direct reference to a datatype and, in most cases, a constraint on that datatype. You can think of this as a kind of hardcoding in your program. While this approach to declarations is certainly valid, it can cause problems in the following situations:

Synchronization with database columns
: The PL/SQL variable "represents" database information in the program. If I declare explicitly and then change the structure of the underlying table, my program may not work properly.

Normalization of local variables
: The PL/SQL variable stores calculated values used throughout the application. What are the consequences of repeating (hardcoding) the same datatype and constraint for each declaration in all of our programs?

Let's take a look at each of these scenarios in detail.

Synchronization with database columns

Databases hold information that needs to be stored and manipulated. Both SQL and PL/SQL perform these manipulations. Your PL/SQL programs often read data from a database into local program variables, and then write information from those variables back into the database.

Suppose that I have a company table with a column called NAME and a datatype of VARCHAR2(60). I can create a local variable to hold this data as follows:

```
DECLARE
   cname VARCHAR2(60);
```

and then use this variable to represent this database information in my program. Now consider an application that uses the company entity. There may be a dozen different screens, procedures, and reports that contain this same PL/SQL declaration, VARCHAR2(60), over and over again. And everything works just fine... until the business requirements change, or the DBA has a change of heart. With a very small effort, the definition of the name column in the company table changes to VARCHAR2(100) in order to accommodate longer company names. Suddenly the database can store names that will raise VALUE_ERROR exceptions when FETCHed into the cname variable.

My program has now become incompatible with the underlying data structures. All declarations of cname (and all the variations programmers have employed for this data throughout the system) must be modified and retested—otherwise, my application is simply a ticking time bomb, just waiting to fail. My variable, which is a local representation of database information, is no longer synchronized with that database column.

Normalization of local variables

Another drawback to explicit declarations arises when you're working with PL/SQL variables that store and manipulate calculated values not found in the database. Suppose that I hire some programmers to build an application to manage my company's finances. I am very bottom-line-oriented, so many different programs make use of a total_revenue variable, declared as follows:

```
total_revenue NUMBER (10,2);
```

Yes, I like to track my total revenue down to the last penny. In 2002, when specifications for the application were first written, the maximum total revenue I ever thought I could possibly obtain was $99 million, so I used the NUMBER(10,2) declaration. Then, in 2005, business grew beyond my expectations and $99 million was not enough, so we increased the maximum to NUMBER(14,2). But then we had a big job of finding and changing all the places where the variables were too small. I searched out any and all instances of the revenue variables so that I could change the declarations. This was a time-consuming and error-prone job—I initially missed a couple of the declarations, and the full regression test had to find them for me. I had spread equivalent declarations throughout the entire application. I had, in effect, denormalized my local data structures, with the usual consequences on maintenance. If only I had a way to define each of the local total_revenue variables in relation to a single datatype...

If only I had used %TYPE!

Anchoring to NOT NULL Datatypes

When you declare a variable, you can also specify the need for the variable to be NOT NULL. This NOT NULL declaration constraint is transferred to variables declared with the %TYPE attribute. If I include a NOT NULL in my declaration of a source variable (one that is referenced afterward in a %TYPE declaration), I must also make sure to specify a default value for the variables that use that source variable. Suppose that I declare max_available_date NOT NULL in the following example:

```
DECLARE
   max_available_date DATE NOT NULL :=
   ADD_MONTHS (SYSDATE, 3);
   last_ship_date max_available_date%TYPE;
```

The declaration of last_ship_date then fails to compile, with the following message:

```
PLS_00218: a variable declared NOT NULL must have an initialization assignment.
```

If you use a NOT NULL variable in a %TYPE declaration, the new variable must have a default value provided. The same is not true, however, for variables declared with %TYPE where the source is a database column defined as NOT NULL. A NOT NULL constraint from a database table is *not* automatically transferred to a variable.

Programmer-Defined Subtypes

With the SUBTYPE statement, PL/SQL allows you to define your own subtypes or aliases of predefined datatypes, sometimes referred to as *abstract datatypes*. In PL/SQL, a subtype of a datatype is a variation that specifies the same set of rules as the original datatype, but that might allow only a subset of the datatype's values.

There are two kinds of subtypes, constrained and unconstrained:

Constrained subtype

A subtype that restricts or constrains the values normally allowed by the datatype itself. POSITIVE is an example of a constrained subtype of BINARY_ INTEGER. The package STANDARD, which predefines the datatypes and the functions that are part of the standard PL/SQL language, declares the subtype POSITIVE as follows:

```
SUBTYPE POSITIVE IS BINARY_INTEGER RANGE 1 .. 2147483647;
```

A variable that is declared POSITIVE can store only integer values greater than zero.

Unconstrained subtype
: A subtype that does not restrict the values of the original datatype in variables declared with the subtype. FLOAT is an example of an unconstrained subtype of NUMBER. Its definition in the STANDARD package is:

```
SUBTYPE FLOAT IS NUMBER;
```

To make a subtype available, you first have to declare it in the declaration section of an anonymous PL/SQL block, procedure, function, or package. You've already seen the syntax for declaring a subtype used by PL/SQL in the STANDARD package. The general format of a subtype declaration is:

```
SUBTYPE subtype_name IS base_type;
```

where *subtype_name* is the name of the new subtype, and *base_type* is the datatype on which the subtype is based.

In other words, an unconstrained subtype provides an alias or alternate name for the original datatype. Here are a few examples:

```
PACKAGE utility
AS
   SUBTYPE big_string IS VARCHAR2(32767);
   SUBTYPE big_db_string IS VARCHAR2(4000);
END utility;
```

Be aware that an anchored subtype does not carry over the NOT NULL constraint to the variables it defines. Nor does it transfer a default value that was included in the original declaration of a variable or column specification.

Conversion Between Datatypes

There are many different situations in which you need to convert data from one datatype to another. You can perform this conversion in two ways:

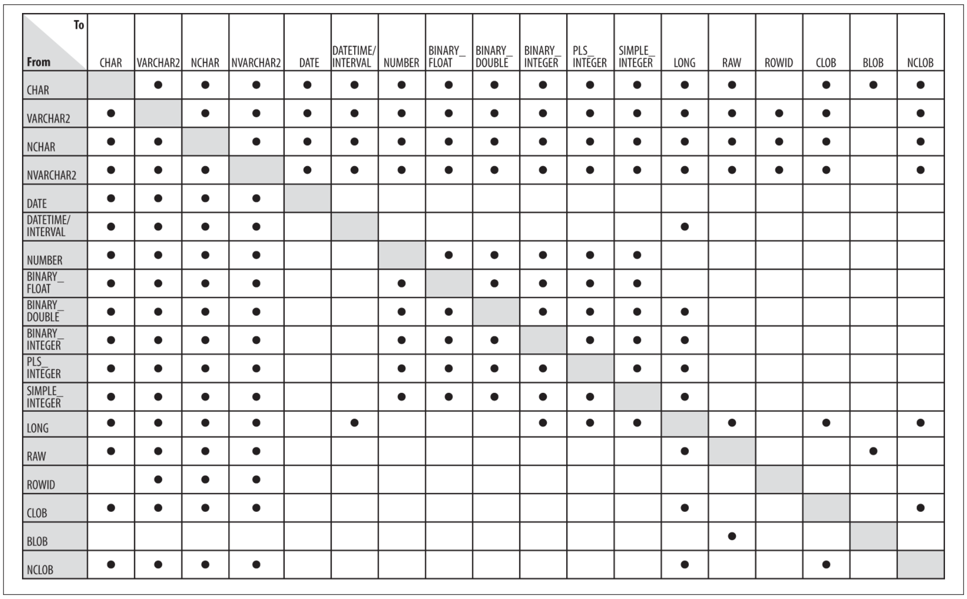
Implicitly
: By allowing the PL/SQL runtime engine to take its "best guess" at performing the conversion

Explicitly
: By calling a PL/SQL function or operator to do the conversion

In this section I will first review how and when PL/SQL performs implicit conversions, and then focus on the functions and operators available for explicit conversions.

Implicit Data Conversion

Whenever PL/SQL detects that a conversion is necessary, it attempts to change the values as required to perform the operation. You would probably be surprised to learn how often PL/SQL performs conversions on your behalf. Figure 7-2 shows what kinds of implicit conversions PL/SQL can perform.



From \ To	CHAR	VARCHAR2	NCHAR	NVARCHAR2	DATE	DATETIME/INTERVAL	NUMBER	BINARY_FLOAT	BINARY_DOUBLE	BINARY_INTEGER	PLS_INTEGER	SIMPLE_INTEGER	LONG	RAW	ROWID	CLOB	BLOB	NCLOB
CHAR		●	●	●	●	●	●	●	●	●	●	●	●	●		●	●	●
VARCHAR2	●		●	●	●	●	●	●	●	●	●	●	●	●	●	●		●
NCHAR	●	●		●	●	●	●	●	●	●	●	●	●	●	●	●		●
NVARCHAR2	●	●	●		●	●	●	●	●	●	●	●	●	●	●	●		●
DATE	●	●	●	●														
DATETIME/INTERVAL	●	●	●	●									●					
NUMBER	●	●	●	●				●	●	●	●	●						
BINARY_FLOAT	●	●	●	●			●		●	●	●	●						
BINARY_DOUBLE	●	●	●	●			●	●		●	●	●	●					
BINARY_INTEGER	●	●	●	●			●	●	●		●	●	●					
PLS_INTEGER	●	●	●	●			●	●	●	●		●	●					
SIMPLE_INTEGER	●	●	●	●			●	●	●	●	●		●					
LONG	●	●	●	●		●				●	●	●		●		●		●
RAW	●	●	●	●									●				●	
ROWID		●	●	●														
CLOB	●	●	●	●									●					●
BLOB														●				
NCLOB	●	●	●	●									●			●		

Figure 7-2. Implicit conversions performed by PL/SQL

With implicit conversions you can specify a literal value in place of data with the correct internal format, and PL/SQL will convert that literal as necessary. In the following example, PL/SQL converts the literal string '125' to the numeric value 125 in the process of assigning a value to the numeric variable:

```
DECLARE
   a_number NUMBER;
BEGIN
   a_number := '125';
END;
```

You can also pass parameters of one datatype into a module and then have PL/SQL convert that data into another format for use inside the program. In the following procedure, the second parameter is a date. When I call that procedure, I pass a string value in the form DD-MON-YY, and PL/SQL converts that string automatically to a date:

```
PROCEDURE change_hiredate
   (emp_id_in IN INTEGER, hiredate_in IN DATE)

change_hiredate (1004, '12-DEC-94');
```

The implicit conversion from string to date datatype follows the NLS_DATE_FORMAT specification. The danger here is that if the NLS_DATE_FORMAT changes, your program breaks.

Limitations of implicit conversion

As shown in Figure 7-2, conversions are limited; PL/SQL cannot convert any arbitrary datatype to any other datatype. Furthermore, some implicit conversions raise exceptions. Consider the following assignment:

```
DECLARE
   a_number NUMBER;
BEGIN
   a_number := 'abc';
END;
```

PL/SQL cannot convert 'abc' to a number and so will raise the VALUE_ERROR exception when it executes this code. It is up to you to make sure that if PL/SQL is going to perform implicit conversions, it is given values it can convert without error.

Drawbacks of implicit conversion

There are several drawbacks to implicit conversion:

- PL/SQL is generally a static typing language. When your program performs an implicit conversion, you lose some of the benefits of the static typing, such as clarity and safety of your code.
- Each implicit conversion PL/SQL performs represents a loss, however small, in the control you have over your program. You do not expressly perform or direct the performance of the conversion; you simply make an assumption that it will take place and that it will have the intended effect. There is always a danger in making this assumption. If Oracle changes the way and circumstances under which it performs conversions or if the data itself no longer conforms to your (or the database's) expectations, your code is then affected.
- The implicit conversion that PL/SQL performs depends on the context in which the code executes. The conversion that PL/SQL performs is not necessarily the one you might expect.
- Your code is easier to read and understand if you explicitly convert between datatypes where needed. Such conversions provide documentation of variances in datatypes between tables or between code and tables. By removing an assumption and a hidden action from your code, you remove a potential misunderstanding as well.

I strongly recommend that you avoid allowing either the SQL or PL/SQL languages to perform implicit conversions on your behalf, especially with datetime conversions. Instead, use conversion functions to guarantee that the right kinds of conversions take place.

Explicit Datatype Conversion

Oracle provides a comprehensive set of conversion functions and operators to be used in SQL and PL/SQL; a complete list is shown in Table 7-1. Most of these functions are described in other chapters (for those, the table indicates the chapter number). For functions not described elsewhere, brief descriptions are provided later in this chapter.

Table 7-1. The built-in conversion functions

Name	Description	Chapter
ASCIISTR	Converts a string in any character set to an ASCII string in the database character set.	8, 25
CAST	Converts one built-in datatype or collection-typed value to another built-in datatype or collection-typed value; this very powerful conversion mechanism can be used as a substitute for traditional functions like TO_DATE.	7, 9, 10
CHARTOROWID	Converts a string to a ROWID.	7
CONVERT	Converts a string from one character set to another.	7
FROM_TZ	Adds time zone information to a TIMESTAMP value, thus converting it to a TIMESTAMP WITH TIME ZONE value.	10
HEXTORAW	Converts from hexadecimal to raw format.	7
MULTISET	Maps a database table to a collection.	12
NUMTODSINTERVAL	Converts a number (or numeric expression) to an INTERVAL DAY TO SECOND literal.	10
NUMTOYMINTERVAL	Converts a number (or numeric expression) to an INTERVAL YEAR TO MONTH literal.	10
RAWTOHEX, RAWTONHEX	Converts from a raw value to hexadecimal.	7
REFTOHEX	Converts a REF value into a character string containing the hexadecimal representation of the REF value.	26
ROWIDTOCHAR, ROWIDTONCHAR	Converts a binary ROWID value to a character string.	7
TABLE	Maps a collection to a database table; this is the inverse of MULTISET.	12
THE	Maps a single column value in a single row into a virtual database table.	12
TO_BINARY_DOUBLE	Converts a number or a string to a BINARY_DOUBLE.	9
TO_BINARY_FLOAT	Converts a number or a string to a BINARY_FLOAT.	9
TO_BLOB	Converts from a RAW value to a BLOB.	13
TO_CHAR, TO_NCHAR (character version)	Converts character data between the database character set and the national character set.	8
TO_CHAR, TO_NCHAR (date version)	Converts a date to a string.	10
TO_CHAR, TO_NCHAR (number version)	Converts a number to a string (VARCHAR2 or NVARCHAR2, respectively).	9
TO_CLOB, TO_NCLOB	Converts from a VARCHAR2, NVARCHAR2, or NCLOB value to a CLOB (or NCLOB).	13

Name	Description	Chapter
TO_DATE	Converts a string to a date.	10
TO_DSINTERVAL	Converts a character string of a CHAR, VARCHAR2, NCHAR, or NVARCHAR2 datatype to an INTERVAL DAY TO SECOND type.	10
TO_LOB	Converts from a LONG to a LOB.	13
TO_MULTI_BYTE	Where possible, converts single-byte characters in the input string to their multibyte equivalents.	8
TO_NUMBER	Converts a string or a number (such as a BINARY_FLOAT) to a NUMBER.	9
TO_RAW	Converts from a BLOB to a RAW.	13
TO_SINGLE_BYTE	Converts multibyte characters in the input string to their corresponding single-byte characters.	8
TO_TIMESTAMP	Converts a character string to a value of type TIMESTAMP.	10
TO_TIMESTAMP_TZ	Converts a character string to a value of type TO_TIMESTAMP_TZ.	10
TO_YMINTERVAL	Converts a character string of a CHAR, VARCHAR2, NCHAR, or NVARCHAR2 datatype to an INTERVAL YEAR TO MONTH type.	10
TRANSLATE ... USING	Converts supplied text to the character set specified for conversions between the database character set and the national character set.	8
UNISTR	Takes as its argument a string in any character set and returns it in Unicode in the database Unicode character set.	8, 25

The CHARTOROWID function

The CHARTOROWID function converts a string of either type CHAR or type VARCHAR2 to a value of type ROWID. The specification of the CHARTOROWID function is:

```
FUNCTION CHARTOROWID (string_in IN CHAR) RETURN ROWID
FUNCTION CHARTOROWID (string_in IN VARCHAR2) RETURN ROWID
```

In order for CHARTOROWID to successfully convert the string, it must be an 18-character string of the format:

```
OOOOOFFFBBBBBBRRR
```

where *OOOOO* is the data object number, *FFF* is the relative file number of the database file, *BBBBBB* is the block number in the file, and *RRR* is the row number within the block. All four numbers must be in base-64 format. If the input string does not conform to the preceding format, PL/SQL raises the VALUE_ERROR exception.

The CAST function

The CAST function is a very handy and flexible conversion mechanism. It converts from one (and almost any) built-in datatype or collection-typed value to another built-in datatype or collection-typed value. CAST will be a familiar operator to anyone working with object-oriented languages, in which it is often necessary to “cast” an object of one class into that of another.

With Oracle’s CAST function, you can convert an unnamed expression (a number, a date, NULL, or even the result set of a subquery) or a named collection (a nested table, for instance) to a datatype or named collection of a compatible type.

Figure 7-3 shows the supported conversions between built-in datatypes. Note the following:

- You cannot cast LONG, LONG RAW, any of the LOB datatypes, or the Oracle-supplied types.
- “DATE” in the figure includes DATE, TIMESTAMP, TIMESTAMP WITH TIME-ZONE, INTERVAL DAY TO SECOND, and INTERVAL YEAR TO MONTH.
- You can cast a named collection type into another named collection type only if the elements of both collections are of the same type.
- You cannot cast a UROWID to a ROWID if the UROWID contains the value of a ROWID of an index-organized table.

To / From	BINARY_FLOAT, BINARY_DOUBLE	CHAR, VARCHAR2	NUMBER	DATE, TIMESTAMP, INTERVAL	RAW	ROWID, UROWID	NCHAR, NVARCHAR2
BINARY_FLOAT, BINARY_DOUBLE	●	●	●				●
CHAR, VARCHAR2	●	●	●	●	●	●	
NUMBER	●	●	●				●
DATE, TIMESTAMP, INTERVAL		●		●			●
RAW		●			●		●
ROWID, UROWID		●					
NCHAR, NVARCHAR2	●		●				●

Figure 7-3. Casting built-in datatypes

First let’s take a look at using CAST as a replacement for scalar datatype conversion. You can use it in a SQL statement:

```
SELECT employee_id, cast (hire_date AS  VARCHAR2 (30))
  FROM employee;
```

and you can use it in native PL/SQL syntax:

```
DECLARE
   hd_display VARCHAR2 (30);
BEGIN
   hd_display := CAST (SYSDATE AS  VARCHAR2);
END;
```

A much more interesting application of CAST comes into play when you are working with PL/SQL collections (nested tables and VARRAYs). For these datatypes, you use CAST to convert from one type of collection to another. You can also use CAST to manipulate (from within a SQL statement) a collection that has been defined as a PL/SQL variable.

Chapter 12 covers these topics in more detail, but the following example should give you a sense of the syntax and possibilities. First I create two nested table types and a relational table:

```
CREATE TYPE names_t AS TABLE OF VARCHAR2 (100);

CREATE TYPE authors_t AS TABLE OF VARCHAR2 (100);

CREATE TABLE favorite_authors (name VARCHAR2(200))
```

I would then like to write a program that blends together data from the favorite_ authors table with the contents of a nested table declared and populated in my program. Consider the following block:

```
     /* File on web: cast.sql */
 1   DECLARE
 2      scifi_favorites   authors_t
 3         := authors_t ('Sheri S. Tepper', 'Orson Scott Card', 'Gene Wolfe');
 4   BEGIN
 5      DBMS_OUTPUT.put_line ('I recommend that you read books by:');
 6
 7      FOR rec IN  (SELECT column_value favs
 8                     FROM TABLE (CAST (scifi_favorites AS  names_t))
 9                  UNION
10                    SELECT NAME
11                      FROM favorite_authors)
12       LOOP
13          DBMS_OUTPUT.put_line (rec.favs);
14       END LOOP;
15    END;
```

On lines 2 and 3, I declare a local nested table and populate it with a few of my favorite science fiction/fantasy authors. In lines 7 through 11, I use the UNION operator to merge together the rows from favorite_authors with those of scifi_favorites. To do this, I *cast* the PL/SQL nested table (local and not visible to the SQL engine) to a type of nested table known in the database. Notice that I am able to cast a collection of type authors_t to a collection of type names_t; this is possible because they are of compatible types. Once the cast step is completed, I call the TABLE operator to ask the SQL engine to treat the nested table as a relational table. Here is the output I see on my screen:

```
I recommend that you read books by:
Gene Wolfe
Orson Scott Card
Robert Harris
Sheri S. Tepper
Tom Segev
Toni Morrison
```

The CONVERT function

The CONVERT function converts strings from one character set to another character set. The specification of the CONVERT function is:

```
FUNCTION CONVERT
   (string_in IN VARCHAR2,
    new_char_set VARCHAR2
    [, old_char_set VARCHAR2])
RETURN VARCHAR2
```

The third argument, *old_char_set*, is optional. If this argument is not specified, then the default character set for the database instance is used.

The CONVERT function does *not* translate words or phrases from one language to another. CONVERT simply substitutes the letter or symbol in one character set with the corresponding letter or symbol in another character set. (Note that a character set is not the same thing as a human language.)

Two commonly used character sets are WE8MSWIN1252 (Microsoft Windows 8-bit Code Page 1252 character set) and AL16UTF16 (16-bit Unicode character set).

The HEXTORAW function

The HEXTORAW function converts a hexadecimal string from type CHAR or VARCHAR2 to type RAW. The specification of the HEXTORAW function is:

```
FUNCTION HEXTORAW (string_in IN CHAR) RETURN RAW
FUNCTION HEXTORAW (string_in IN VARCHAR2) RETURN RAW
```

The RAWTOHEX function

The RAWTOHEX function converts a value from type RAW to a hexadecimal string of type VARCHAR2. The specification of the RAWTOHEX function is:

```
FUNCTION RAWTOHEX (binary_value_in IN RAW) RETURN VARCHAR2
```

RAWTOHEX always returns a variable-length string value, even if its mirror conversion function is overloaded to support both types of input.

The ROWIDTOCHAR function

The ROWIDTOCHAR function converts a binary value of type ROWID to a string of type VARCHAR2. The specification of the ROWIDTOCHAR function is:

```
FUNCTION ROWIDTOCHAR (row_in IN ROWID ) RETURN VARCHAR2
```

The string returned by this function has the format:

```
OOOOOFFFBBBBBBRRR
```

where *OOOOO* is the data object number, *FFF* is the relative file number of the database file, *BBBBBB* is the block number in the file, and *RRR* is the row number within the block. All four numbers are in base-64 format. For example:

```
AAARYiAAEAAAEG8AAB
```