CHAPTER 8
Strings

Variables with character datatypes store text and are manipulated by character functions. Working with character data can range in difficulty from easy to quite challenging. In this chapter, I discuss PL/SQL's core string functionality largely in the context of single-byte character sets—for example, those that are commonly used in Western Europe and the United States. If you are working with Unicode or with multibyte character sets, or are dealing with multiple languages, be sure to read about globalization and localization issues in Chapter 25.

CLOB (character large object) and LONG, while arguably character types, cannot be used in the same manner as the character types discussed in this chapter, and are more usefully thought of as *large object types*. I discuss large object types in Chapter 13.

String Datatypes

Oracle supports four string datatypes, summarized in the following table. Which type you should use depends on your answers to the following two questions:

- Are you working with variable-length or fixed-length strings?
- Do you want to use the database character set or the national character set?

	Fixed-length	Variable-length
Database character set	CHAR	VARCHAR2
National character set	NCHAR	NVARCHAR2

You will rarely need or want to use the fixed-length CHAR and NCHAR datatypes in Oracle-based applications; in fact, I recommend that you never use these types unless there is a specific requirement for fixed-length strings. See "Mixing CHAR and VARCHAR2 Values" on page 229 for a description of problems you may encounter when mixing fixed- and variable-length string variables. (The NCHAR and NVARCHAR2 datatypes are discussed in Chapter 25.)

The VARCHAR2 Datatype

VARCHAR2 variables store variable-length character strings. When you declare a variable-length string, you must also specify a maximum length for the string, which can range from 1 to 32,767 bytes. You may specify the maximum length in terms of characters or bytes, but either way the length is ultimately defined in bytes. The general format for a VARCHAR2 declaration is:

```
variable_name VARCHAR2 (max_length [CHAR | BYTE]);
```

where:

variable_name
: Is the name of the variable you want to declare

max_length
: Is the maximum length of the variable

CHAR
: Indicates that *max_length* is expressed in terms of characters

BYTE
: Indicates that *max_length* represents a number of bytes

When you specify the maximum length of a VARCHAR2 string in terms of characters (using the CHAR qualifier), the actual length in bytes is determined using the largest number of bytes that the database character set uses to represent a character. For example, the Unicode UTF-8 character set uses up to 4 bytes for some characters; thus, if UTF-8 is your underlying character set, declaring a VARCHAR2 variable with a maximum length of 100 characters is equivalent to declaring the same variable with a maximum length of 300 bytes.

> You'll find the CHAR length qualifier most useful when working with multibyte character sets such as Unicode UTF-8. Read more about character semantics and character sets in Chapter 25.

If you omit the CHAR or BYTE qualifier when declaring a VARCHAR2 variable, then whether the size is in characters or bytes depends on the NLS_LENGTH_SEMANTICS initialization parameter. You can determine your current setting by querying NLS_SESSION_PARAMETERS.

Following are some examples of VARCHAR2 declarations:

```
DECLARE
   small_string VARCHAR2(4);
   line_of_text VARCHAR2(2000);
   feature_name VARCHAR2(100 BYTE); -- 100-byte string
   emp_name VARCHAR2(30 CHAR); -- 30-character string
```

The maximum length allowed for PL/SQL VARCHAR2 variables is 32,767 bytes. This size limit applies regardless of whether you declare a variable's size in terms of characters or bytes.

Prior to 12*c*, the maximum length of the VARCHAR2 datatype *in SQL* was 4,000; in 12*c*, this is now increased to match the PL/SQL maximum: 32,767 bytes. Note, however, that SQL supports this maximum size only if the MAX_SQL_STRING_SIZE initialization parameter is set to EXTENDED; the default value is STANDARD.

If you need to work with strings in SQL that are greater than 4,000 bytes in length and you have not yet upgraded to 12*c*, consider storing those strings in CLOB columns. See Chapter 13 for information on CLOBs.

The CHAR Datatype

The CHAR datatype specifies a fixed-length character string. When you declare a fixed-length string, you also specify a maximum length for the string, which can range from 1 to 32,767 bytes. You can specify the length in terms of bytes or in terms of characters. For example, the following two declarations create strings of 100 bytes and 100 characters, respectively:

```
feature_name CHAR(100 BYTE);
feature_name CHAR(100 CHAR);
```

The actual number of bytes in a 100-character string depends on the underlying database character set. If you are using a variable-width character set, PL/SQL will allocate enough bytes to the string to accommodate the specified number of worst-case characters. For example, UTF-8 uses between 1 and 4 bytes per character, so PL/SQL will assume the worst and allocate 3 bytes × 100 characters, for a total of 300 bytes.

If you leave off the BYTE or CHAR qualifier, the results will depend on the setting of the NLS_LENGTH_SEMANTICS initialization parameter. When you compile your program, this setting is saved along with it and may be reused or overwritten during later recompilation. (Compilation settings are discussed in Chapter 20.) Assuming the default setting, the following declaration results in a 100-byte string:

```
feature_name CHAR(100);
```

If you do not specify a length for the string, PL/SQL declares a string of one byte. Suppose you declare a variable as follows:

```
feature_name CHAR;
```

As soon as you assign a string of more than one character to the variable feature_name, PL/ SQL will raise the generic VALUE_ERROR exception:

```
ORA-06502: PL/SQL: numeric or value error: character string buffer too small
```

Notice that the message does not indicate which variable was involved in the error. If you get this error after declaring some new variables or constants, check your declarations for a lazy use of CHAR. To avoid mistakes and to prevent future programmers from wondering about your intent, you should *always* specify a length when you use the CHAR datatype. Several examples follow:

```
yes_or_no CHAR (1) DEFAULT 'Y';
line_of_text    CHAR (80 CHAR); -- Always a full 80 characters!
whole_paragraph CHAR (10000 BYTE); -- Think of all those spaces...
```

Because CHAR is fixed-length, PL/SQL will right-pad any value assigned to a CHAR variable with spaces to the maximum length specified in the declaration.

Prior to 12*c*, the maximum length of the CHAR datatype *in SQL* was 2,000; in 12*c*, this is now increased to match the PL/SQL maximum: 32,767 bytes. Note, however, that SQL supports these maximum sizes only if the MAX_SQL_STRING_SIZE initialization parameter is set to EXTENDED.

String Subtypes

PL/SQL supports several string subtypes, listed in Table 8-1, that you can use when declaring character string variables. Many of these subtypes exist for the ostensible purpose of providing compatibility with the ANSI SQL standard. It's unlikely that you'll ever need to use these—I never do—but you should be aware that they exist.

Table 8-1. PL/SQL subtypes and their equivalents

Subtype	Equivalent PL/SQL type
CHAR VARYING	VARCHAR2
CHARACTER	CHAR
CHARACTER VARYING	VARCHAR2
NATIONAL CHAR	NCHAR
NATIONAL CHAR VARYING	NVARCHAR2
NATIONAL CHARACTER	NCHAR
NATIONAL CHARACTER VARYING	NVARCHAR2
NCHAR VARYING	NVARCHAR2
STRING	VARCHAR2
VARCHAR	VARCHAR2

Each subtype listed in the table is equivalent to the base PL/SQL type shown in the right column. For example, the following declarations all have the same effect:

```
feature_name VARCHAR2(100);
feature_name CHARACTER VARYING(100);
feature_name CHAR VARYING(100);
feature_name STRING(100);
```

The VARCHAR subtype deserves special mention. For years now, Oracle Corporation has been threatening to change the meaning of VARCHAR (to something not equivalent to VARCHAR2) and warning against its use. I agree with Oracle's recommendation: if there is a possibility of VARCHAR's behavior being changed by Oracle (or the ANSI committee), it's senseless to depend on its current behavior. Don't use VARCHAR; use VARCHAR2.

Working with Strings

Working with strings is largely a matter of manipulating them using Oracle's rich library of built-in string functions. To that end, I recommend that you become broadly familiar with the functions Oracle has to offer. In the subsections that follow, I'll begin by showing you how to write string constants, and then introduce you to the string manipulation functions that I have come to find most important in my own work.

Specifying String Constants

One way to get strings into your PL/SQL programs is to issue a SELECT statement that returns character string values. Another way is to place string constants directly into your code. You write such constants by enclosing them within single quotes:

```
'Brighten the corner where you are.'
```

If you want to embed a single quote within a string constant, you can do so by typing the single quote twice:

```
'Aren''t you glad you''re learning PL/SQL with O''Reilly?'
```

If your program will be dealing with strings that contain embedded single-quote characters, a more elegant approach is to specify your own string delimiters. Do this using the *q* prefix (uppercase *Q* may also be specified). For example:

```
q'!Aren't you glad you're learning PL/SQL with O'Reilly?!'
```

or:

```
q'{Aren't you glad you're learning PL/SQL with O'Reilly?}'
```

When you use the *q* prefix, you still must enclose the entire string within single quotes. The character immediately following the first quotation mark—an exclamation point (!) in the first of my two examples—then becomes the *delimiter* for the string. Thus, the first of my *q*-prefixed strings consists of all characters between the two exclamation points.

Special rule: if your start delimiter character is one of [, {, <, or (, then your end delimiter character must be], }, >, or), respectively.

Normally, string constants are represented using the database character set. If such a string constant is assigned to an NCHAR or NVARCHAR2 variable, the constant will be implicitly converted to the national character set (see Chapter 25). The database performs such conversions when necessary, and you rarely need to worry about them. Occasionally, however, you may need to explicitly specify a string constant to be represented in the national character set. You can do so using the *n* prefix:

```
n'Pils vom faß: 1€'
```

If you need a string in the national character set, and you also want to specify some characters by their Unicode code point, you can use the *u* prefix:

```
u'Pils vom fa\00DF: 1\20AC'
```

00DF is the code point for the German letter ß, while 20AC is the code point for the Euro symbol. The resulting string constant is the same as for the preceding *n*-prefixed example.

Using the assignment operator, you can store the value of a string constant within a variable:

```
DECLARE
   jonathans_motto VARCHAR2(50);
BEGIN
   jonathans_motto := 'Brighten the corner where you are.';
END;
```

You can also pass string constants to built-in functions. For example, to find out the number of characters in Jonathan's motto, you can use the LENGTH function:

```
BEGIN
   DBMS_OUTPUT.PUT_LINE(
      LENGTH('Brighten the corner where you are.')
   );
END;
```

Run this code, and you'll find that the number of characters is 34.

While this is not strictly a PL/SQL issue, you'll often find that ampersand (&) characters cause problems if you're executing PL/SQL code via SQL*Plus or SQL Developer. Both tools use ampersands to prefix substitution variables. When they encounter an ampersand, these tools "see" the next word as a variable and prompt you to supply a value:

```
SQL> BEGIN
  2     DBMS_OUTPUT.PUT_LINE ('Generating & saving test data.');
  3  END;
  4  /
Enter value for saving:
```

There are several solutions to this problem. One that works well with SQL*Plus and SQL Developer is to issue the command SET DEFINE OFF to disable the variable substitution feature. Other solutions can be found in Jonathan Gennick's book *Oracle SQL*Plus: The Definitive Guide*.

Using Nonprintable Characters

The built-in CHR function is especially valuable when you need to make reference to a nonprintable character in your code. Suppose you have to build a report that displays the address of a company. A company can have up to four address strings (in addition to city, state, and zip code). Your boss wants each address string to start on a new line. You can do that by concatenating all the address lines together into one long text value and using CHR to insert linefeeds where desired. The location in the standard ASCII collating sequence for the linefeed character is 10, so you can code:

```
SELECT name || CHR(10)
       || address1 || CHR(10)
       || address2 || CHR(10)
       || city || ', ' || state || ' ' || zipcode
       AS company_address
FROM company
```

Suppose I have inserted the following row into the table:

```
BEGIN
   INSERT INTO company
        VALUES ('Harold Henderson',
                '22 BUNKER COURT',
                NULL,
                'WYANDANCH',
                'MN',
                '66557');

   COMMIT;
END;
/
```

The results will end up looking like:

```
COMPANY_ADDRESS
--------------------
Harold Henderson
22 BUNKER COURT

WYANDANCH, MN 66557
```

Linefeed is the newline character for Linux and Unix systems. Windows uses the carriage return character together with the newline CHR(13)||CHR(10). In other environments, you may need to use some other character.

What? You say your boss doesn't want to see any blank lines? No problem. You can eliminate those with a bit of cleverness involving the NVL2 function:

```
SELECT name
       || NVL2(address1, CHR(10) || address1, '')
       || NVL2(address2, CHR(10) || address2, '')
       || CHR(10) || city || ', ' || state || ' ' || zipcode
       AS company_address
FROM company
```

Now the query returns a single formatted column per company. The NVL2 function returns the third argument when the first is NULL, and otherwise returns the second argument. In this example, when address1 is NULL, the empty string (‘’) is returned, and likewise for the other address columns. In this way, blank address lines are not returned, so the address will be scrunched down to:

```
COMPANY_ADDRESS
--------------------
Harold Henderson
22 BUNKER COURT
WYANDANCH, MN 66557
```

The ASCII function, in essence, does the reverse of CHR: it returns the decimal representation of a given character in the database character set. For example, execute the following code to display the decimal code for the letter *J*:

```
BEGIN
   DBMS_OUTPUT.PUT_LINE(ASCII('J'));
END;
```

and you'll find that, in UTF-8 at least, the underlying representation of *J* is the value 74.

Watch for an interesting use of CHR in the section "Traditional Searching, Extracting, and Replacing" on page 210.

Concatenating Strings

There are two mechanisms for concatenating strings: the CONCAT function and the concatenation operator, represented by two vertical bar characters (||). By far the more commonly used approach is the concatenation operator. Why, you may be asking yourself, are there two mechanisms? Well... there may be issues in translating the vertical bars in code between ASCII and EBCDIC servers, and some keyboards make typing the vertical bars a feat of finger agility. If you find it difficult to work with the vertical bars, use the CONCAT function, which takes two arguments as follows:

```
CONCAT (string1, string2)
```

CONCAT always appends *string2* to the end of *string1* and returns the result. If either string is NULL, CONCAT returns the non-NULL argument all by its lonesome. If both strings are NULL, CONCAT returns NULL. If the input strings are non-CLOB, the resulting string will be a VARCHAR2. If one or both input strings are CLOBs, then the resulting datatype will be a CLOB as well. If one string is an NCLOB, the resulting datatype will be an NCLOB. In general, the return datatype will be the one that preserves the most information. Here are some examples of uses of CONCAT (where --> means that the function returns the value shown):

```
CONCAT ('abc', 'defg') --> 'abcdefg'
CONCAT (NULL, 'def') --> 'def'
CONCAT ('ab', NULL) --> 'ab'
CONCAT (NULL, NULL) --> NULL
```

Notice that you can concatenate only two strings with the database function. With the concatenation operator, you can combine several strings. For example:

```
DECLARE
   x VARCHAR2(100);
BEGIN
   x := 'abc' || 'def' || 'ghi';
   DBMS_OUTPUT.PUT_LINE(x);
END;
```

The output is:

```
abcdefghi
```

To perform the identical concatenation using CONCAT, you would need to nest one call to CONCAT inside another:

```
x := CONCAT(CONCAT('abc','def'),'ghi');
```

You can see that the || operator not only is much easier to use than CONCAT, but also results in much more readable code.

Dealing with Case

Letter case is often an issue when working with strings. For example, you might want to compare two strings regardless of case. There are different approaches you can take to dealing with this problem, depending partly on the database release you are running and partly on the scope that you want your actions to have.

Forcing a string to all upper- or lowercase

One way to deal with case issues is to use the built-in UPPER and LOWER functions. These functions let you force case conversion on a string for a single operation. For example:

```
DECLARE
   name1 VARCHAR2(30) := 'Andrew Sears';
   name2 VARCHAR2(30) := 'ANDREW SEARS';
BEGIN
   IF LOWER(name1) = LOWER(name2) THEN
       DBMS_OUTPUT.PUT_LINE('The names are the same.');
   END IF;
END;
```

In this example, both strings are passed through LOWER so the comparison ends up being between 'andrew sears' and 'andrew sears'.

Making comparisons case insensitive

Starting with Oracle Database 10*g* Release 2, you can use the initialization parameters NLS_COMP and NLS_SORT to render all string comparisons case insensitive. Set the NLS_COMP parameter to LINGUISTIC, which will tell the database to use NLS_SORT for string comparisons. Then set NLS_SORT to a case-insensitive setting, like BINARY_CI or XWEST_EUROPEAN_CI. The trailing _CI specifies *c*ase *i*nsensitivity. Here's a simple, SQL-based example that illustrates the kind of problem you can solve using NLS_COMP. The problem is to take a list of names and determine which should come first:

```
SELECT LEAST ('JONATHAN','Jonathan','jon') FROM dual
```

On my system the call to LEAST that you see here returns 'JONATHAN'. That's because the uppercase characters sort lower than the lowercase characters. By default, NLS_COMP is set to BINARY, meaning that string comparisons performed by functions such as LEAST are based on the underlying character code values.

You might like to see LEAST ignore case and return 'jon' instead of 'JONATHAN'. To that end, you can change NLS_COMP to specify that a linguistic sort (sensitive to the NLS_SORT settings) be performed:

```
ALTER SESSION SET NLS_COMP=LINGUISTIC
```

Next, you must change NLS_SORT to specify the sorting rules that you want. The default NLS_SORT value is often BINARY, but it may be otherwise depending on how your system is configured. For this example, use the sort BINARY_CI. The _CI suffix specifies a case-insensitive sort:

```
ALTER SESSION SET NLS_SORT=BINARY_CI
```

Now, try that call to LEAST one more time:

```
SELECT LEAST ('JONATHAN','Jonathan','jon') FROM dual
```

This time, the result is 'jon'. This may seem like a simple exercise, but this result is not so easy to achieve without the linguistic sorting I've just described.

And it's not just functions that are affected by linguistic sorting—simple string comparisons are affected as well. For example:

```
BEGIN
   IF 'Jonathan' = 'JONATHAN' THEN
      DBMS_OUTPUT.PUT_LINE('It is true!');
  END IF;
END;
```

With NLS_COMP and NLS_SORT set as I've described, the expression 'Jonathan' = 'JONATHAN' in this example evaluates to TRUE.

NLS_COMP and NLS_SORT settings affect all string manipulations that you do. The settings "stick" until you change them, or until you terminate your session.

Oracle also supports accent-insensitive sorting, which you can get by appending _AI (rather than _CI) to a sort name. To find a complete list of linguistic sort names, refer to Oracle's *Database Globalization Support Guide*. That guide also explains the operation of NLS_COMP and NLS_SORT in detail. Also refer to Chapter 25 of this book, which presents more information on the various NLS parameters at your disposal.

Case insensitivity and indexes

When dealing with strings, you often want to do case-insensitive searches and comparisons. But when you implement the nifty technique described here, you find that your application stops using indexes and starts performing poorly. Take care that you don't inadvertently negate the use of indexes in your SQL. Let's look at an example using the demonstration table hr.employees to illustrate. The employees table has the index emp_name_ix on columns last_name, first_name. My code includes the following SQL:

```
SELECT * FROM employees WHERE last_name = lname
```

Initially the code is using the emp_name_ix index, but when I set NLS_COMP=LINGUISTIC and NLS_SORT=BINARY_CI to enable case insensitivity I stop using the index and start doing full table scans instead—oops! One solution is to create a function-based, case-insensitive index, like this:

```
CREATE INDEX last_name_ci ON EMPLOYEES (NLSSORT(last_name, 'NLS_SORT=BINARY_CI'))
```

Now when I do my case-insensitive query, I use the case-insensitive index and keep my good performance.

Capitalizing each word in a string

A third case-related function, after UPPER and LOWER, is INITCAP. This function forces the initial letter of each word in a string to uppercase, and all remaining letters to lowercase. For example, if I write code like this:

```
DECLARE
   name VARCHAR2(30) := 'MATT williams';
BEGIN
   DBMS_OUTPUT.PUT_LINE(INITCAP(name));
END;
```

The output will be:

```
Matt Williams
```

It's wonderfully tempting to use INITCAP to properly format names, and all will be fine until you run into a case like:

```
DECLARE
   name VARCHAR2(30) := 'JOE mcwilliams';
BEGIN
   DBMS_OUTPUT.PUT_LINE(INITCAP(name));
END;
```

which generates this output:

```
Joe Mcwilliams
```

Joe McWilliams may not be so happy to see his last name written as "Mcwilliams," with a lowercase *w*. INITCAP is handy at times, but remember that it doesn't yield correct results for words or names having more than just an initial capital letter.

Traditional Searching, Extracting, and Replacing

Frequently, you'll find yourself wanting to search a string for a bit of text. Starting with Oracle Database 10*g*, you can use regular expressions for these textual manipulations; see "Regular Expression Searching, Extracting, and Replacing" on page 216 for the full details. If you're not yet using Oracle Database 10*g* or later, you can use an approach that is backward compatible with older database versions. The INSTR function returns the character position of a substring within a larger string. The following code finds the locations of all the commas in a list of names:

```
DECLARE
   names VARCHAR2(60) := 'Anna,Matt,Joe,Nathan,Andrew,Aaron,Jeff';
   comma_location NUMBER := 0;
BEGIN
   LOOP
      comma_location := INSTR(names,',',comma_location+1);
      EXIT WHEN comma_location = 0;
      DBMS_OUTPUT.PUT_LINE(comma_location);
   END LOOP;
END;
```

The output is:

```
5
10
14
21
28
34
```

The first argument to INSTR is the string to search. The second is the substring to look for, in this case a comma. The third argument specifies the character position at which to begin looking. After each comma is found, the loop begins looking again one character further down the string. When no match is found, INSTR returns zero, and the loop ends.

Now that we've found the location of some text in a string, a natural next step is to extract it. I don't care about those commas. Let's extract the names instead. For that, I'll use the SUBSTR function:

```
DECLARE
   names VARCHAR2(60) := 'Anna,Matt,Joe,Nathan,Andrew,Aaron,Jeff';
   names_adjusted VARCHAR2(61);
   comma_location NUMBER := 0;
   prev_location NUMBER := 0;
BEGIN
   -- Stick a comma after the final name
   names_adjusted := names || ',';
   LOOP
      comma_location := INSTR(names_adjusted,',',comma_location+1);
      EXIT WHEN comma_location = 0;
      DBMS_OUTPUT.PUT_LINE(
         SUBSTR(names_adjusted,
                prev_location+1,
                comma_location-prev_location-1));
      prev_location := comma_location;
   END LOOP;
END;
```

The list of names that I get is:

```
Anna
Matt
Joe
Nathan
Andrew
Aaron
Jeff
```

The keys to the preceding bit of code are twofold. First, a comma is appended to the end of the string to make the loop's logic easier to write. Every name in names_adjusted is followed by a comma. That simplifies life. Then, each time the loop iterates to DBMS_OUTPUT.PUT_LINE, the two variables named prev_location and comma_location point to the character positions on either side of the name to print. It's then just a matter of some simple math and the SUBSTR function. Three arguments are passed:

names_adjusted
: The string from which to extract a name.

prev_location+1
: The character position of the first letter in the name. Remember that prev_location will point to just before the name to display, usually to a comma preceding the name. That's why I add 1 to the value.

comma_location-prev_location-1
: The number of characters to extract. I subtract the extra 1 to avoid displaying the trailing comma.

All this searching and extracting is fairly tedious. Sometimes I can reduce the complexity of my code by cleverly using some of the built-in functions. Let's try the REPLACE function to swap those commas with newlines:

```
DECLARE
   names VARCHAR2(60) := 'Anna,Matt,Joe,Nathan,Andrew,Aaron,Jeff';
BEGIN
   DBMS_OUTPUT.PUT_LINE(
      REPLACE(names, ',', chr(10))
   );
END;
```

And the output is (!):

```
Anna
Matt
Joe
Nathan
Andrew
Aaron
Jeff
```

By using REPLACE I was able to avoid all that looping. I got the same results with code that is simpler and more elegant. Of course, you won't always be able to avoid loop processing by using REPLACE, but it's good to know about alternative algorithms. With programming, there are always several ways to get the results you want!

Negative String Positioning

Some of Oracle's built-in string functions, notably SUBSTR and INSTR, allow you to determine the position from which to begin extracting or searching by counting backward from the right end of a string. For example, to extract the final 10 characters of a string:

```
SUBSTR('Brighten the corner where you are',-10)
```

This function call returns “re you are”. The key is the use of –10 as the starting position. By making the starting position negative, you instruct SUBSTR to count backward from the end of the string.

INSTR adds an interesting twist to all of this. Specify a negative starting index, and INSTR will:

1. Count back from the end of the string to determine from whence to begin searching.
2. Search backward from that point toward the beginning of the string.

Step 1 is the same as for SUBSTR, but Step 2 proceeds in quite the opposite direction. For example, to find the occurrence of “re” that is second from the end:

```
INSTR('Brighten the corner where you are','re',-1,2)
```

To help illustrate these concepts, here are the letter positions in the string:

```
                111111111122222222223333
       123456789012345678901234567890123
INSTR('Brighten the corner where you are','re',-1,2)
```

The result is 24. The fourth parameter, a 2, requests the second occurrence of “re”. The third parameter is –1, so the search begins at the last character of the string (the first character prior to the closing quote). The search progresses backward toward the beginning, past the “re” at the end of “are” (the first occurrence), until reaching the occurrence of “re” at the end of “where”.

There is one subtle case in which INSTR with a negative position will search forward. Here’s an example:

```
INSTR('Brighten the corner where you are','re',-2,1)
```

The –2 starting position means that the search begins with the *r* in “are”. The result is 32. Beginning from the *r* in “are”, INSTR looks forward to see whether it is pointing at an occurrence of “re”. And it is, so INSTR returns the current position in the string, which happens to be the 32nd character. Thus, the “re” in “are” is found even though it extends past the point at which INSTR began searching.

Padding

Occasionally it’s helpful to force strings to be a certain size. You can use LPAD and RPAD to add spaces (or some other character) to either end of a string in order to make the string a specific length. The following example uses the two functions to display a list of names two-up in a column, with the leftmost name being flush left and the rightmost name appearing flush right:

```
DECLARE
   a VARCHAR2(30) := 'Jeff';
   b VARCHAR2(30) := 'Eric';
   c VARCHAR2(30) := 'Andrew';
   d VARCHAR2(30) := 'Aaron';
   e VARCHAR2(30) := 'Matt';
   f VARCHAR2(30) := 'Joe';
BEGIN
   DBMS_OUTPUT.PUT_LINE(   RPAD(a,10) || LPAD(b,10)   );
   DBMS_OUTPUT.PUT_LINE(   RPAD(c,10) || LPAD(d,10)   );
   DBMS_OUTPUT.PUT_LINE(   RPAD(e,10) || LPAD(f,10)   );
END;
```

The output is:

```
Jeff            Eric
Andrew         Aaron
Matt             Joe
```

The default padding character is the space. If you like, you can specify a fill character as the third argument. Change the lines of code to read:

```
   DBMS_OUTPUT.PUT_LINE(   RPAD(a,10,'.') || LPAD(b,10,'.')   );
   DBMS_OUTPUT.PUT_LINE(   RPAD(c,10,'.') || LPAD(d,10,'.')   );
   DBMS_OUTPUT.PUT_LINE(   RPAD(e,10,'.') || LPAD(f,10,'.')   );
```

And the output changes to:

```
Jeff............Eric
Andrew.........Aaron
Matt.............Joe
```

Your fill "character" can even be a string of characters:

```
   DBMS_OUTPUT.PUT_LINE(   RPAD(a,10,'-~-') || LPAD(b,10,'-~-')   );
   DBMS_OUTPUT.PUT_LINE(   RPAD(c,10,'-~-') || LPAD(d,10,'-~-')   );
   DBMS_OUTPUT.PUT_LINE(   RPAD(e,10,'-~-') || LPAD(f,10,'-~-')   );
```

Now the output looks like:

```
Jeff-~--~--~--~-Eric
Andrew-~---~--~Aaron
Matt-~--~--~--~--Joe
```

Fill characters or strings are laid down from left to right—always, even when RPAD is used. You can see that that's the case if you study carefully the 10-character "column" containing Joe's name.

One possible problem to think about when using LPAD and RPAD is the possibility that some of your input strings may already be longer than (or equal to) the width that you desire. For example, if I change the column width to four characters:

```
   DBMS_OUTPUT.PUT_LINE(   RPAD(a,4) || LPAD(b,4)   );
   DBMS_OUTPUT.PUT_LINE(   RPAD(c,4) || LPAD(d,4)   );
   DBMS_OUTPUT.PUT_LINE(   RPAD(e,4) || LPAD(f,4)   );
```

now the output looks like:

```
JeffEric
AndrAaro
Matt Joe
```

Notice particularly the second row: both "Andrew" and "Aaron" were truncated to just four characters.

Trimming

What LPAD and RPAD giveth, TRIM, LTRIM, and RTRIM taketh away. For example:

```
DECLARE
   a VARCHAR2(40) := 'This sentence has too many periods......';
   b VARCHAR2(40) := 'The number 1';
BEGIN
   DBMS_OUTPUT.PUT_LINE(   RTRIM(a,'.')   );
   DBMS_OUTPUT.PUT_LINE(
      LTRIM(b, 'ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz')
   );
END;
```

The output is:

```
This sentence has too many periods
1
```

As you can see, RTRIM removed all the periods. The second argument to that function (here, '.') specifies the character(s) to trim. My use of LTRIM is a bit absurd, but it demonstrates that you can specify an entire set of characters to trim. I asked that all letters and spaces be trimmed from the beginning of the string b, and I got what I asked for.

The default is to trim spaces from the beginning or end of the string. Specifying RTRIM(a) is the same as asking for RTRIM(a,' '). The same goes for LTRIM(a) and LTRIM(a,' ').

The other trimming function is just plain TRIM. Oracle added TRIM when Oracle8*i* Database was released in order to make the database more compliant with the ISO SQL standard. TRIM works a bit differently from LTRIM and RTRIM, as you can see:

```
DECLARE
   x VARCHAR2(30) := '.....Hi there!.....';
BEGIN
   DBMS_OUTPUT.PUT_LINE(   TRIM(LEADING '.' FROM x)   );
   DBMS_OUTPUT.PUT_LINE(   TRIM(TRAILING '.' FROM x)   );
   DBMS_OUTPUT.PUT_LINE(   TRIM(BOTH '.' FROM x)   );
   -- The default is to trim from both sides
   DBMS_OUTPUT.PUT_LINE(   TRIM('.' FROM x)   );

   -- The default trim character is the space:
   DBMS_OUTPUT.PUT_LINE(   TRIM(x)   );
END;
```

The output is:

```
Hi there!.....
.....Hi there!
Hi there!
Hi there!
.....Hi there!.....
```

It's one function, yet you can use it to trim from either side or from both sides. However, you can specify only a single character to remove. You cannot, for example, write:

```
TRIM(BOTH ',.;' FROM x)
```

Instead, to solve this particular problem, you can use a combination of RTRIM and LTRIM:

```
RTRIM(LTRIM(x,',.;'),',.;')
```

If you want to trim a set of characters, your options are RTRIM and LTRIM.

Regular Expression Searching, Extracting, and Replacing

Oracle Database 10*g* introduced a very powerful change to string manipulation: support for regular expressions. And I'm not talking the mundane regular expression support involving the LIKE predicate that you find in other database management systems. Oracle has given us a well-thought-out and powerful feature set—just what PL/SQL needed.

Regular expressions form a sort of pattern language for describing and manipulating text. Those of you familiar with Perl doubtless know a bit about the topic already, as Perl has done more to spread the use of regular expressions than perhaps any other language. Regular expression support in Oracle Database 10*g* followed closely the Portable Operating System Interface (POSIX) regular expression standard. Oracle Database 10*g* Release 2 added support for many nonstandard, but quite useful, operators from the world of Perl, and Oracle Database 11*g* augmented these features with yet more capabilities.

Detecting a pattern

Regular expressions give you a pattern language you can use to describe text that you want to find and manipulate. To illustrate, let's revisit the example used throughout the section "Traditional Searching, Extracting, and Replacing" on page 210:

```
DECLARE
   names VARCHAR2(60) := 'Anna,Matt,Joe,Nathan,Andrew,Aaron,Jeff';
```

I will assign myself the task of determining programmatically whether names represents a list of comma-delimited elements. I can do that using the REGEXP_LIKE function, which detects the presence of a pattern in a string:

```
DECLARE
   names VARCHAR2(60) := 'Anna,Matt,Joe,Nathan,Andrew,Jeff,Aaron';
   names_adjusted VARCHAR2(61);
   comma_delimited BOOLEAN;
BEGIN
   -- Look for the pattern
   comma_delimited := REGEXP_LIKE(names,'^([a-z A-Z]*,)+([a-z A-Z]*){1}$');

   -- Display the result
   DBMS_OUTPUT.PUT_LINE(
      CASE comma_delimited
         WHEN true THEN 'We have a delimited list!'
         ELSE 'The pattern does not match.'
      END);
END;
```

The result is:

```
We have a delimited list!
```

To understand what's going on here, you must begin with the expression defining the pattern you seek. The general syntax for the REGEXP_LIKE function is:

```
REGEXP_LIKE (source_string, pattern [,match_modifier])
```

Where *source_string* is the character string to be searched, *pattern* is the regular expression pattern to search for in *source_string*, and *match_modifier* is one or more modifiers that apply to the search. If REGEXP_LIKE finds *pattern* in *source_string*, then it returns the Boolean TRUE; otherwise, it returns FALSE.

The following recaps my thought process as I put the example together:

[a-z A-Z]
: Each entry in my list of names must consist of only letters and spaces. Square brackets define a set of characters on which to match. I use a-z to give all lowercase letters, and I use A-Z to give all uppercase letters. The space sits between those two parts of the expression. So, any lowercase character, any uppercase character, or a space would match this pattern.

*[a-z A-Z]**
: The asterisk is a *quantifier*, specifying that I want to see zero or more characters in each list item.

[a-z A-Z],*
: Each list item must terminate with a comma. An exception is the final item, but I can safely ignore that nuance for now.

([a-z A-Z],)*
: I use parentheses to define a subexpression that matches some number of characters terminated by a comma. I define this subexpression because I want to specify that the entire thing repeats.

([a-z A-Z],)+*
: The plus sign is another quantifier and applies to the preceding element, which happens to be the subexpression. In contrast to the *, the + requires "one or more." A comma-delimited list consists of one or more of my subexpressions.

([a-z A-Z],)+([a-z A-Z]*)*
: I add another subexpression: ([a-z A-Z]*). This is almost a duplicate of the first, but it doesn't include the comma. The final list item is not terminated by a comma.

([a-z A-Z],)+([a-z A-Z]*){1}*
: I add the quantifier {1} to allow for exactly one list element with no trailing comma.

^([a-z A-Z],)+([a-z A-Z]*){1}$*
: Finally, I use ^ and $ to anchor my expression to the beginning and end, respectively, of the target string. I do this to require that the entire string, rather than some subset of the string, match my pattern.

Using REGEXP_LIKE, I examine the names string to see whether it matches the pattern. And it does:

```
We have a delimited list!
```

REGEXP_LIKE is optimized to detect the mere presence of a pattern within a string. Other functions let you do even more. Keep reading!

Locating a pattern

You can use REGEXP_INSTR to locate occurrences of a pattern within a string. The general syntax for REGEXP_INSTR is:

```
REGEXP_INSTR (source_string, pattern [,beginning_position [,occurrence
   [,return_option [,match_modifier [,subexpression]]]]])
```

Where *source_string* is the character string to be searched, *pattern* is the regular expression pattern to search for in *source_string*, *beginning_position* is the character position at which to begin the search, *occurrence* is the ordinal occurrence desired (1 = first, 2 = second, etc.), *return_option* is either 0 for the beginning position or 1 for the ending position, and *match_modifier* is one or more modifiers that apply to the search, such as *i* for case insensitivity. Beginning with Oracle Database 11*g*, you can also specify a *subexpression* (1 = first subexpression, 2 = second subexpression, etc.), which causes

REGEXP_INST to return the starting position for the specified subexpression. A subexpression is a part of the pattern enclosed in parentheses.

For example, to find the first occurrence of a name beginning with the letter *A* and ending with a consonant, you might specify:

```
DECLARE
   names VARCHAR2(60) := 'Anna,Matt,Joe,Nathan,Andrew,Jeff,Aaron';
   names_adjusted VARCHAR2(61);
   comma_delimited BOOLEAN;
   j_location NUMBER;
BEGIN
   -- Look for the pattern
   comma_delimited := REGEXP_LIKE(names,'^([a-z ]*,)+([a-z ]*)$', 'i');

   -- Only do more if we do, in fact, have a comma-delimited list
   IF comma_delimited THEN
      j_location := REGEXP_INSTR(names, 'A[a-z]*[^aeiou],|A[a-z]*[^aeiou]$');
      DBMS_OUTPUT.PUT_LINE(j_location);
   END IF;
END;
```

Execute this code and you'll find that the first *A* name ending with a consonant, which happens to be Andrew, begins at position 22. Here's how I worked out the pattern:

A
: I begin with the letter *A*. No need to worry about commas, because I already know at this point that I am working with a delimited list.

*A[a-z]**
: I follow that *A* with some number of letters or spaces. The * allows for zero or more such characters following the *A*.

A[a-z][^aeiou]*
: I add [^aeiou] because I want my name to end with anything but a vowel. The caret ^ creates an exclusion set—any character *except* a vowel will match. Because I specify no quantifier, exactly one such nonvowel is required.

A[a-z][^aeiou],*
: I require a comma to end the pattern. Otherwise, I'd have a match on the "An" of "Anna." While adding the comma solves that problem, it introduces another, because my pattern now will never match Aaron at the end of the string. Uh-oh...

A[a-z][^aeiou],|A[a-z]*[^aeiou]$*
: Here I've introduced a vertical bar (|) into the mix. The | indicates alternation: I am now looking for a match with either pattern. The first pattern ends with a comma, whereas the second does not. The second pattern accommodates the possibility that the name I'm looking for is the final name in the list. The second pattern is thus anchored to the end of the string by the dollar sign ($).

Writing regular expressions is not easy! As a beginner, you'll discover subtleties to regular expression evaluation that will trip you up. I spent quite a bit of time working out just this one example, and went down several dead-end paths before getting it right. Don't despair, though. Writing regular expressions does become easier with practice.

While REGEXP_INSTR has its uses, I am often more interested in returning the text matching a pattern than I am in simply locating it.

Extracting text matching a pattern

Let's use a different example to illustrate regular expression extraction. Phone numbers are a good example because they follow a pattern, but often there are several variations on this pattern. The phone number pattern includes the area code (three digits) followed by the exchange (three digits) followed by the local number (four digits). So, a phone number is a string of 10 digits. But there are many optional and alternative ways to represent the number. The area code may be enclosed within parentheses and is usually, but not always, separated from the rest of the phone number with a space, dot, or dash character. The exchange is also usually, but not always, separated from the rest of the phone number with a space, dot, or dash character. Thus, a legal phone number may include any of the following:

```
7735555253
773-555-5253
(773)555-5253
(773) 555 5253
773.555.5253
```

This kind of loosey-goosey pattern is easy work using regular expressions, but very hard without them. I'll use REGEXP_SUBSTR to extract a phone number from a string containing contact information:

```
DECLARE
  contact_info VARCHAR2(200) := '
    address:
    1060 W. Addison St.
    Chicago, IL 60613
    home 773-555-5253
  ';
  phone_pattern  VARCHAR2(90) :=
    '\(?\d{3}\)?[[:space:]\.\-]?\d{3}[[:space:]\.\-]?\d{4}';
BEGIN
  DBMS_OUTPUT.PUT_LINE('The phone number is: '||
    REGEXP_SUBSTR(contact_info,phone_pattern,1,1));
END;
```

This code shows me the phone number:

```
  The phone number is: 773-555-5253
```

Whoa! That phone pattern is pretty intimidating, with all those punctuation characters strung together. Let me break it down into manageable pieces:

\(?
: My phone pattern starts with an optional open parenthesis character. Because the parentheses characters are *metacharacters* (have special meaning), I need to *escape* the open parenthesis by preceding it with a backslash. The question mark is a *quantifier*, specifying that the pattern allows zero or one of the preceding character. This portion of the pattern specifies an optional open parenthesis character.

\d{3}
: The \d is one of those Perl-influenced operators introduced with Oracle Database 10*g* Release 2; it specifies a digit. The curly brackets are a quantifier, specifying that the pattern allows an exact number of preceding characters—in this case, three. This portion of the pattern specifies three digits.

\)?
: This portion of the pattern specifies an optional close parenthesis character.

[[:space:]\.\-]?
: The square brackets define a set of characters on which to match—in this case, a whitespace character or a dot or a dash. The [:space:] notation is the POSIX character class for whitespace characters in our NLS character set—any whitespace character will match. A dot and a dash are metacharacters, so I need to escape them in my pattern by preceding each with a backslash. Finally, the question mark specifies that the pattern allows zero or one of the preceding characters. This portion of the pattern specifies an optional whitespace, dot, or dash character.

\d{3}
: As described previously, this portion of the pattern specifies three digits.

[[:space:]\.\-]?
: As described previously, this portion of the pattern specifies an optional whitespace, dot, or dash character.

\d{4}
: As described previously, this portion of the pattern specifies four digits.

When you code with regular expressions, commenting your code becomes more important to someone (including yourself six months from now) wanting to understand your cleverness.

The general syntax for REGEXP_SUBSTR is:

```
REGEXP_SUBSTR (source_string, pattern [,position [,occurrence
   [,match_modifier [,subexpression]]]])
```

REGEXP_SUBSTR returns a string containing the portion of the source string matching the pattern or subexpression. If no matching pattern is found, a NULL is returned.

source_string is the character string to be searched, *pattern* is the regular expression pattern to search for in *source_string*, *position* is the character position at which to begin the search, *occurrence* is the ordinal occurrence desired (1 = first, 2 = second, etc.), and *match_modifier* is one or more modifiers that apply to the search.

Beginning with Oracle Database 11*g*, you can also specify which subexpression to return (1 = first subexpression, 2 = second subexpression, etc.). A subexpression is a part of the pattern enclosed in parentheses. Subexpressions are useful when you need to match on the whole pattern but want only a portion of that patterned extracted. If I want to find the phone number but extract only the area code, I enclose the area code portion of the pattern in parentheses, making it a subexpression:

```
DECLARE
  contact_info VARCHAR2(200) := '
    address:
    1060 W. Addison   St.
    Chicago, IL 60613
    home 773-555-5253
    work (312) 555-1234
    cell 224.555.2233
    ';
  phone_pattern  VARCHAR2(90) :=
    '\(?(\d{3})\)?[[:space:]\.\-]?\d{3}[[:space:]\.\-]?\d{4}';
  contains_phone_nbr BOOLEAN;
  phone_number VARCHAR2(15);
  phone_counter NUMBER;
  area_code VARCHAR2(3);
BEGIN
  contains_phone_nbr := REGEXP_LIKE(contact_info,phone_pattern);
  IF contains_phone_nbr THEN
    phone_counter := 1;
    DBMS_OUTPUT.PUT_LINE('The phone numbers are:');
    LOOP
      phone_number := REGEXP_SUBSTR (contact_info,phone_pattern,1,phone_counter);
      EXIT WHEN phone_number IS NULL;  -- NULL means no more matches
      DBMS_OUTPUT.PUT_LINE(phone_number);
      phone_counter := phone_counter + 1;
    END LOOP;
    phone_counter := 1;
    DBMS_OUTPUT.PUT_LINE('The area codes are:');
    LOOP
      area_code := REGEXP_SUBSTR
        (contact_info,phone_pattern,1,phone_counter,'i',1);
      EXIT WHEN area_code IS NULL;
      DBMS_OUTPUT.PUT_LINE(area_code);
      phone_counter := phone_counter + 1;
    END LOOP;
  END IF;
END;
```

This snippet of code extracts the phone numbers and area codes:

```
  The phone numbers are:
  773-555-5253
  (312) 555-1234
  224.555.2233
  The area codes are:
  773
  312
  224
```

Counting regular expression matches

Sometimes, you just want a count of how many matches your regular expression has. Prior to Oracle Database 11*g*, you had to loop through and count each match. Now you can use the new function REGEXP_COUNT to tally up the number of matches. The general syntax for REGEXP_COUNT is:

```
REGEXP_COUNT (source_string, pattern [,position [,match_modifier]])
```

where *source_string* is the character string to be searched, *pattern* is the regular expression pattern to search for in *source_string*, *position* is the character position at which to begin the search, and *match modifier* is one or more modifiers that apply to the search. For example:

```
DECLARE
  contact_info VARCHAR2(200) := '
    address:
    1060 W. Addison   St.
    Chicago, IL 60613
    home 773-555-5253
    work (312) 123-4567';
  phone_pattern  VARCHAR2(90) :=
  '\(?(\d{3})\)?[[:space:]\.\-]?(\d{3})[[:space:]\.\-]?\d{4}';
BEGIN
  DBMS_OUTPUT.PUT_LINE('There are '
    ||REGEXP_COUNT(contact_info,phone_pattern)
    ||' phone numbers');
END;
```

The result is:

```
There are 2 phone numbers
```

Replacing text

Search and replace is one of the best regular expression features. Your replacement text can refer to portions of your source text (called *back references*), enabling you to manipulate text in very powerful ways. Imagine that you're faced with the problem of displaying a comma-delimited list of names two to a line. One way to do that is to replace every second comma with a newline character. Again, this is hard to do with standard REPLACE, but easy using REGEXP_REPLACE.

The general syntax for REGEXP_REPLACE is:

```
REGEXP_REPLACE (source_string, pattern [,replacement_string
   [,position [,occurrence [,match_modifier]])
```

where *source_string* is the character string to be searched, *pattern* is the regular expression pattern to search for in *source_string*, *replacement_string* is the replace text for *pattern*, *position* is the character position at which to begin the search, and *match_modifier* is one or more modifiers that apply to the search.

Let's look at an example:

```
DECLARE
   names VARCHAR2(60) := 'Anna,Matt,Joe,Nathan,Andrew,Jeff,Aaron';
   names_adjusted VARCHAR2(61);
   comma_delimited BOOLEAN;
   extracted_name VARCHAR2(60);
   name_counter NUMBER;
BEGIN
   -- Look for the pattern
   comma_delimited := REGEXP_LIKE(names,'^([a-z ]*,)+([a-z ]*){1}$', 'i');

   -- Only do more if we do, in fact, have a comma-delimited list
   IF comma_delimited THEN
      names := REGEXP_REPLACE(
                  names,
                  '([a-z A-Z]*),([a-z A-Z]*),',
                  '\1,\2' || chr(10)   );
   END IF;

   DBMS_OUTPUT.PUT_LINE(names);
END;
```

The output from this bit of code is:

```
Anna,Matt
Joe,Nathan
Andrew,Jeff
Aaron
```

I'll begin my explanation of this bit of wizardry by pointing out that I passed three arguments to REGEXP_REPLACE:

names
: The source string.

'([a-z A-Z]),([a-z A-Z]*),'*
: An expression specifying the text that I want to replace. More on this in just a bit.

'\1,\2 ' || chr(10)
: My replacement text. The \1 and \2 are back references and are what makes my solution work. I'll talk more about these in just a bit too.

The expression I'm searching for consists of two subexpressions enclosed within parentheses, plus two commas. Here's an explanation of how that expression works:

([a-z A-Z])*
: I want to begin by matching a name.

,
: I want that name to be terminated by a comma.

([a-z A-Z])*
: Then I want to match another name.

,
: And I again want to match the terminating comma.

Remember that my goal is to replace every second comma with a newline. That's why I wrote my expression to match two names and two commas. There's a reason, too, why I kept the commas out of the subexpressions.

Following is the first match that will be found for my expression upon invoking REGEXP_REPLACE:

```
Anna,Matt,
```

The two subexpressions will correspond to Anna and Matt, respectively. The key to my solution is that you can reference the text matching a given subexpression via a back reference. The two back references in my replacement text are \1 and \2, and they refer to the text matched by the first and second subexpressions. Here's how that plays out:

```
'\1,\2' || chr(10)       -- our replacement text
'Anna,\2' || chr(10)     -- fill in the value matched
                            by the first subexpression
'Anna,Matt' || chr(10)   -- fill in the value matched
                            by the second subexpression
```

I hope you can begin to see the power at your disposal here. I don't even use the commas from the original text. I use only the text matching the two subexpressions, the names Anna and Matt, and I insert those into a new string formatted with one comma and one newline.

I can do even more! I can easily change the replacement text to use a tab (an ASCII 9) rather than a comma:

```
names := REGEXP_REPLACE(
            names,
            '([a-z A-Z]*),([a-z A-Z]*),',
            '\1' || chr(9) || '\2' || chr(10)   );
```

And now I get my results in two nice, neat columns:

```
Anna    Matt
Joe     Nathan
```

```
Andrew  Jeff
Aaron
```

I think regular expression search and replace is a wonderful thing. It's fun. It's powerful. You can do a lot with it.

Groking greediness

Greediness is an important concept to understand when writing regular expressions. Consider the problem of extracting just the first name *and its trailing comma* from our comma-delimited list of names. Recall that the list looks like this:

```
names VARCHAR2(60) := 'Anna,Matt,Joe,Nathan,Andrew,Jeff,Aaron';
```

One solution that you might think of is to look for a series of characters ending in a comma:

```
.*,
```

Let's try this solution to see how it works:

```
DECLARE
   names VARCHAR2(60) := 'Anna,Matt,Joe,Nathan,Andrew,Jeff,Aaron';
BEGIN
   DBMS_OUTPUT.PUT_LINE(   REGEXP_SUBSTR(names, '.*,')   );
END;
```

My output is:

```
Anna,Matt,Joe,Nathan,Andrew,Jeff,
```

Well! This is certainly not what we were after. What happened? I was a victim of greediness. Not the sort of greediness your mother chastised you about, but rather a greediness of the regular expression sort: each element of a regular expression will match as many characters as it possibly can. When you and I see:

```
.*,
```

our natural tendency often is to think in terms of *stopping* at the first comma and returning "Anna,". However, the database looks for the longest run of characters it can find that terminates with a comma; the database stops not at the first comma, but at the *last*.

In Oracle Database 10*g* Release 1, when regular expression support was first introduced, you had limited options for dealing with greediness problems. You may be able to reformulate an expression to avoid the problem. For example, you can use '[^,]*,' to return the first name and its trailing comma from your delimited string. Sometimes, though, you are forced to change your whole approach to solving a problem, often to the point of using a completely different combination of functions than you first intended.

Starting with Oracle Database 10*g* Release 2 you get some relief from greed, in the form of nongreedy quantifiers inspired by those found in Perl. By adding a question mark

(?) to the quantifier for the period (.), changing that quantifier from an * to *?, I can request the shortest run of characters that precedes a comma, as follows:

```
DECLARE
   names VARCHAR2(60) := 'Anna,Matt,Joe,Nathan,Andrew,Jeff,Aaron';
BEGIN
   DBMS_OUTPUT.PUT_LINE(   REGEXP_SUBSTR(names, '(.*?,)')   );
END;
```

The output now is:

```
Anna,
```

The nongreedy quantifiers match as *soon* as they can, not as *much* as they can.

Learning more about regular expressions

Regular expressions can seem deceptively simple, but this is a surprisingly deep topic. They are simple enough that you'll be able to use them after just reading this chapter (I hope!), and yet there's so much more to learn. I'd like to recommend the following sources from Oracle and O'Reilly:

Oracle Database Application Developer's Guide—Fundamentals
: Chapter 4 of this Oracle manual is the definitive source of information on regular expression support in Oracle.

Oracle Regular Expressions Pocket Reference
: A fine introduction to regular expressions, written by Jonathan Gennick and Peter Linsley. Peter is one of the developers for Oracle's regular expression implementation.

Mastering Oracle SQL
: Contains an excellent chapter introducing regular expressions in the context of Oracle SQL. Aside from regular expressions, this book by Sanjay Mishra and Alan Beaulieu is an excellent read if you want to hone your SQL skills.

Mastering Regular Expressions
: Jeffrey Friedl's book is the definitive font of wisdom on using regular expressions. If you want to really delve deeply into the topic, this is the book to read.

Finally, in Appendix A you'll find a table describing each of the regular expression metacharacters supported in Oracle's implementation of regular expressions.

Working with Empty Strings

One issue that often causes great consternation, especially to people who come to Oracle after working with other databases, is that the Oracle database treats empty strings as NULLs. This is contrary to the ISO SQL standard, which recognizes the difference between an empty string and a string variable that is NULL.

The following code demonstrates the Oracle database's behavior:

```
/* File on web: empty_is_null.sql */
DECLARE
   empty_varchar2 VARCHAR2(10) := '';
   empty_char CHAR(10) := '';
BEGIN
   IF empty_varchar2 IS NULL THEN
      DBMS_OUTPUT.PUT_LINE('empty_varchar2 is NULL');
   END IF;

   IF '' IS NULL THEN
      DBMS_OUTPUT.PUT_LINE(''''' is NULL');
   END IF;

   IF empty_char IS NULL THEN
      DBMS_OUTPUT.PUT_LINE('empty_char is NULL');
   ELSIF empty_char IS NOT NULL THEN
      DBMS_OUTPUT.PUT_LINE('empty_char is NOT NULL');
   END IF;
END;
```

The output is:

```
empty_varchar2 is NULL
'' is NULL
empty_char is NOT NULL
```

You'll notice in this example that the CHAR variable is not considered NULL. That's because CHAR variables, as fixed-length character strings, are never truly empty. The CHAR variable in this example is padded with blanks until it is exactly 10 characters in length. The VARCHAR2 variable, however, is NULL, as is the zero-length string literal.

You have to really watch for this behavior in IF statements that compare two VARCHAR2 values. Recall that a NULL is never equal to a NULL. Consider a program that queries the user for a name, and then compares that name to a value read in from the database:

```
DECLARE
   user_entered_name VARCHAR2(30);
   name_from_database VARCHAR2(30);
   ...
BEGIN
...
IF user_entered_name <> name_from_database THEN
...
```

If the user had entered an empty string instead of a name, the IF condition shown in this example would never be TRUE. That's because a NULL is never not equal, or equal, to any other value. One alternative approach to this IF statement is the following:

```
IF (user_entered_name <> name_from_database)
   OR (user_entered_name IS NULL) THEN
```

This is just one way of dealing with the "empty string is NULL" issue; it's impossible to provide a solution that works in all cases. You must think through what you are trying to accomplish, recognize that any empty strings will be treated as NULLs, and code appropriately.

Mixing CHAR and VARCHAR2 Values

If you use both fixed-length (CHAR) and variable-length (VARCHAR2) strings in your PL/SQL code, you should be aware of how the database handles the interactions between these two datatypes, as described in the following subsections.

Database-to-variable conversion

When you SELECT or FETCH data from a CHAR database column into a VARCHAR2 variable, the trailing spaces are retained. If you SELECT or FETCH from a VARCHAR2 database column into a CHAR variable, PL/SQL automatically pads the value with spaces out to the maximum length. In other words, the type of the variable, not the column, determines the variable's resulting value.

Variable-to-database conversion

When you INSERT or UPDATE a CHAR variable into a VARCHAR2 database column, the SQL kernel does not trim the trailing blanks before performing the change. When the following PL/SQL is executed, the company_name in the new database record is set to "ACME SHOWERS........" (where . indicates a space). It is, in other words, padded out to 20 characters, even though the default value was a string of only 12 characters.

```
DECLARE
   comp_id# NUMBER;
   comp_name CHAR(20) := 'ACME SHOWERS';
BEGIN
   SELECT company_id_seq.NEXTVAL
      INTO comp_id#
      FROM dual;
   INSERT INTO company (company_id, company_name)
      VALUES (comp_id#, comp_name);
END;
```

On the other hand, when you INSERT or UPDATE a VARCHAR2 variable into a CHAR database column, the SQL kernel automatically pads the variable-length string with spaces out to the maximum (fixed) length specified when the table was created, and places that expanded value into the database.

String comparisons

Suppose your code contains a string comparison such as the following:

```
IF company_name = parent_company_name ...
```

PL/SQL must compare company_name to parent_company_name. It performs the comparison in one of two ways, depending on the types of the two variables:

- If a comparison is made between two CHAR variables, then PL/SQL uses *blank-padding* comparison.
- If at least one of the strings involved in the comparison is variable-length, then PL/SQL performs *non-blank-padding* comparison.

The following code snippet illustrates the difference between these two comparison methods:

```
DECLARE
   company_name CHAR(30)
      := 'Feuerstein and Friends';
   char_parent_company_name CHAR(35)
      := 'Feuerstein and Friends';
   varchar2_parent_company_name VARCHAR2(35)
      := 'Feuerstein and Friends';
BEGIN
   -- Compare two CHARs, so blank-padding is used
   IF company_name = char_parent_company_name THEN
      DBMS_OUTPUT.PUT_LINE ('first comparison is TRUE');
   ELSE
      DBMS_OUTPUT.PUT_LINE ('first comparison is FALSE');
   END IF;

   -- Compare a CHAR and a VARCHAR2, so nonblank-padding is used
   IF company_name = varchar2_parent_company_name THEN
      DBMS_OUTPUT.PUT_LINE ('second comparison is TRUE');
   ELSE
      DBMS_OUTPUT.PUT_LINE ('second comparison is FALSE');
   END IF;
END;
```

The output is:

```
first comparison is TRUE
second comparison is FALSE
```

The first comparison is between two CHAR values, so blank-padding is used: PL/SQL blank-pads the shorter of the two values out to the length of the longer value. It then performs the comparison. In this example, PL/SQL adds five spaces to the end of the value in company_name and then performs the comparison between company_name and char_parent_company_name. The result is that both strings are considered equal. Note that PL/SQL does not actually change the company_name variable's value. It copies the value to another memory structure and then modifies this temporary data for the comparison.

The second comparison involves a VARCHAR2 value, so PL/SQL performs a non-blank-padding comparison. It makes no changes to any of the values, uses the existing lengths, and performs the comparison. In this case, the first 22 characters of both strings are the same, "Feuerstein and Friends", but the fixed-length company_name is padded with eight space characters, whereas the variable-length VARCHAR2 company_name is not. Because one string has trailing blanks and the other does not, the two strings are not considered equal.

The fact that one VARCHAR2 value causes non-blank-padding comparisons is also true of expressions involving more than two variables, as well as of expressions involving the IN operator. For example:

```
IF menu_selection NOT IN
      (save_and_close, cancel_and_exit, 'OPEN_SCREEN')
   THEN ...
```

If any of the four strings in this example (menu_selection, the two named constants, and the single literal) is declared VARCHAR2, then exact comparisons without modification are performed to determine if the user has made a valid selection. Note that a literal like OPEN_SCREEN is always considered a fixed-length CHAR datatype.

Character functions and CHAR arguments

A character function is a function that takes one or more character values as parameters and returns either a character value or a number value. When a character function returns a character value, that value is always of type VARCHAR2 (variable length), with the exceptions of UPPER and LOWER. These functions convert to uppercase and lowercase, respectively, and return CHAR values (fixed length) if the strings they are called on to convert are fixed-length CHAR arguments.

String Function Quick Reference

As I have already pointed out, PL/SQL provides a rich set of string functions that allow you to get information about strings and modify the contents of those strings in very high-level, powerful ways. The following list gives you an idea of the power at your disposal and will be enough to remind you of syntax. For complete details on a given function, see Oracle's *SQL Reference* manual.

ASCII(`single_character`)
: Returns the NUMBER code that represents the specified character in the database character set.

ASCIISTR(`string1`)
: Takes a string in any character set and converts it into a string of ASCII characters. Any non-ASCII characters are represented using the form \XXXX, where *XXXX* represents the Unicode value for the character.

For information on Unicode, including the underlying bytecodes used to represent characters in the Unicode character set, visit the Unicode Consortium website.

CHR(`code_location`)
: Returns a VARCHAR2 character (length 1) that corresponds to the location in the collating sequence provided as a parameter. This is the reverse of ASCII. One variation is useful when working with national character set data:

  CHR(`code_location` USING NCHAR_CS)
  : Returns an NVARCHAR2 character from the national character set.

COMPOSE(`string1`)
: Takes a Unicode string as input and returns that string in its fully normalized form. For example, you can use the unnormalized representation 'a\0303' to specify the character *a* with a ~ on top (i.e., ã). COMPOSE('a\0303') will then return '\00E3', which is the Unicode code point (in hexadecimal) for the character ã.

In Oracle9*i* Database Release 1, COMPOSE must be called from a SQL statement; it cannot be used in a PL/SQL expression. From Oracle9*i* Database Release 2 onward, you can invoke COMPOSE from a PL/SQL expression.

CONCAT(`string1`, `string2`)
: Appends *string2* to the end of *string1*. You'll get the same results as from the expression *string1* || *string2*. I find the || operator so much more convenient that I almost never invoke the CONCAT function.

CONVERT(`string1`, `target_char_set`)
: Converts a string from the database character set to the specified target character set. You may optionally specify a source character set:

```
CONVERT(string1, target_char_set, source_character_set)
```

DECOMPOSE(string1)
: Takes a Unicode string as input and returns that string with any precomposed characters decomposed into their separate elements. This is the opposite of COMPOSE. For example, DECOMPOSE('ã') yields 'a˜'. (See COMPOSE.)

  Two variations are available:

  DECOMPOSE(`string1` CANONICAL)
  : Results in canonical decomposition, which gives a result that may be reversed using COMPOSE. This is the default.

DECOMPOSE(`string1`)
: Results in decomposition in what is referred to as *compatibility mode*. Recomposition using COMPOSE may not be possible.

> Like COMPOSE, DECOMPOSE cannot be invoked directly from a PL/SQL expression in Oracle9*i* Database Release 1; you must invoke it from a SQL statement. From Oracle9*i* Database Release 2 onward, this restriction is removed.

GREATEST(`string1`, `string2`, ...)
: Takes one or more strings as input, and returns the string that would come last (i.e., that is the greatest) if the inputs were sorted in ascending order. Also see the LEAST function, which is the opposite of GREATEST.

INITCAP(`string1`)
: Reformats the case of the string argument, setting the first letter of each word to uppercase and the remainder of the letters to lowercase. This is sometimes called *title case*. A word is a set of characters separated by a space or nonalphanumeric character (such as # or _). For example, INITCAP('this is lower') gives 'This Is Lower'.

INSTR(`string1`, `string2`)
: Returns the position at which *string2* is found within *string1*; if it is not found, returns 0.

  Several variations are available:

  INSTR(`string1`, `string2`, `start_position`)
  : Begins searching for *string2* at the column in *string1* indicated by *start_position*. The default start position is 1, so INSTR(*string1*, *string2*, 1) is equivalent to INSTR(*string1*, *string2*).

  INSTR(`string1`, `string2`, `negative_start_position`)
  : Begins searching from the end of *string1* rather than from the beginning.

  INSTR(`string1`, `string2`, `start_position`, `nth`)
  : Finds the *n*th occurrence of *string2* after the *start_position*.

  INSTR(`string1`, `string2`, `negative_start_position`, `nth`)
  : Finds the *n*th occurrence of *string2*, counting from the end of *string1*.

  INSTR treats a string as a sequence of characters. The variations INSTRB, INSTR2, and INSTR4 treat a string as a sequence of bytes, Unicode code units, and Unicode code points, respectively. The variation INSTRC treats a string as a series of complete Unicode characters. For example, 'a\0303', which is the decomposed equivalent of ‘\00E3’, or *ã*, is treated and counted as a single character. INSTR, however, sees ‘a\0303’ as two characters.

LEAST(`string1`, `string2`, ...)
: Takes one or more strings as input and returns the string that would come first (i.e., that is the least) if the inputs were sorted in ascending order. Also see GREATEST, which is the opposite of LEAST.

LENGTH(`string1`)
: Returns the number of characters in a string. The variations LENGTHB, LENGTH2, and LENGTH4 return the number of bytes, the number of Unicode code units, and the number of Unicode code points, respectively. The variation LENGTHC returns the number of complete Unicode characters, normalizing (e.g., changing ‘a\0303’ to ‘\00E3’) where possible.

  LENGTH typically does not return zero. Remember that the Oracle database treats an empty string (‘’) as a NULL, so LENGTH(‘’) is the same as trying to take the length of a NULL, and the result is NULL. The sole exception is when LENGTH is used against a CLOB. It is possible for a CLOB to hold zero bytes and yet not be NULL. In this one case, LENGTH returns zero.

LOWER(`string1`)
: Converts all letters in the specified string to lowercase. This is the opposite of UPPER. The return datatype is the same as the input datatype (CHAR, VARCHAR2, CLOB). See also NLS_LOWER.

LPAD(`string1`, `padded_length`)
: Returns the value from *string1*, but padded on the left with enough spaces to make the result *padded_length* characters long. There is one variation, shown next:

  LPAD(`string1`, `padded_length`, `pad_string`)
  : Appends enough full or partial occurrences of *pad_string* to bring the total length up to *padded_length*. For example, LPAD(‘Merry Christmas!’, 25, ‘Ho! ’) results in ‘Ho! Ho! HMerry Christmas!’.

  LPAD is the opposite of RPAD.

LTRIM(`string1`)
: Removes (trims) space characters from the left, or leading edge, of *string1*. Also see TRIM (ISO standard) and RTRIM. There is one variation:

  LTRIM(`string1`, `trim_string`)
  : Removes any characters found in *trim_string* from the left end of *string1*.

NCHR(`code_location`)
: Returns an NVARCHAR2 character (length 1) that corresponds to the location in the national character set collating sequence specified by the *code_location* param-

eter. The CHR function's USING NCHAR_CS clause provides the same functionality as NCHR.

NLS_INITCAP(`string1`)

Returns a version of *string1*, which should be of type NVARCHAR2 or NCHAR, setting the first letter of each word to uppercase and the remainder of the letters to lowercase. This is sometimes called *title case*. The return value is a VARCHAR2. A *word* is a set of characters separated by a space or nonalphanumeric character.

You may specify a linguistic sorting sequence that affects the definition of "first letter":

NLS_INITCAP(`string1`, 'NLS_SORT=`sort_sequence_name`')

When you're using this syntax, *sort_sequence_name* should be a valid, linguistic sort name as described in Appendix A of Oracle's *Globalization Support Guide* under the heading "Linguistic Sorts."

The following example illustrates the difference between INITCAP and NLS_INITCAP:

```
BEGIN
   DBMS_OUTPUT.PUT_LINE(INITCAP('ijzer'));
   DBMS_OUTPUT.PUT_LINE(NLS_INITCAP('ijzer','NLS_SORT=XDUTCH'));
END;
```

The output is:

```
Ijzer
IJzer
```

In the Dutch language, the character sequence "ij" is treated as a single character. NLS_INITCAP correctly recognizes this as a result of the NLS_SORT specification and uppercases the word *ijzer* (Dutch for "iron") appropriately.

NLS_LOWER(`string1`) and NLS_LOWER(`string1`, 'NLS_SORT=`sort_sequence_name`')

Returns *string1* in lowercase in accordance with language-specific rules. See NLS_INITCAP for a description of how the NLS_SORT specification can affect the results.

NLS_UPPER(`string1`) and NLS_UPPER(`string1`, 'NLS_SORT=`sort_sequence_name`')

Returns *string1* in uppercase in accordance with language-specific rules. See NLS_INITCAP for a description of how the NLS_SORT specification can affect the results.

NLSSORT(`string1`) and NLSSORT(`string1`, 'NLS_SORT=`sort_sequence_name`')
: Returns a string of bytes that can be used to sort a string value in accordance with language-specific rules. The string returned is of the RAW datatype. For example, to compare two strings using French sorting rules:

```
IF NLSSORT(x, 'NLS_SORT=XFRENCH') > NLSSORT(y, 'NLS_SORT=XFRENCH') THEN...
```

When you omit the second parameter, the function uses the default sort sequence that you have established for your session. For a list of sort sequences, see Appendix A of Oracle's *Globalization Support Guide* under the heading "Linguistic Sorts."

REGEXP_COUNT, REGEXP_INSTR, REGEXP_LIKE, REGEXP_REPLACE, REGEXP_SUBSTR
: Refer to Appendix A of this book for information on these regular expression functions.

REPLACE(`string1`, `match_string`, `replace_string`)
: Returns a string in which all occurrences of *match_string* in *string1* are replaced by *replace_string*. REPLACE is useful for searching a pattern of characters, and then changing all instances of that pattern in a single function call. It has one variation:

REPLACE(`string1`, `match_string`)
: Returns *string1* with all occurrences of *match_string* removed.

RPAD(`string1`, `padded_length`)
: Returns the value from *string1*, but padded on the right with enough spaces to make the result *padded_length* characters long. There is one variation:

RPAD(`string1`, `padded_length`, `pad_string`)
: Appends enough full or partial occurrences of *pad_string* to bring the total length up to *padded_length*. For example, RPAD('Merry Christmas! ', 25, 'Ho! ') results in 'Merry Christmas! Ho! Ho!'.

RPAD pads on the right, while its complement, LPAD, pads on the left.

RTRIM(`string1`)
: Removes (trims) space characters from the right, or trailing edge, of *string1*. See also TRIM (ISO standard) and LTRIM. There is one variation:

RTRIM(`string1`, `trim_string`)
: Removes any characters found in *trim_string* from the trailing edge of *string1*.

SOUNDEX(`string1`)
: Returns a character string that is the "phonetic representation" of the argument. For example:

```
SOUNDEX ('smith') --> 'S530'
SOUNDEX ('SMYTHE') --> 'S530'
SOUNDEX ('smith smith') --> 'S532'
SOUNDEX ('smith z') --> 'S532'
SOUNDEX ('feuerstein') --> 'F623'
SOUNDEX ('feuerst') --> 'F623'
```

Keep the following SOUNDEX rules in mind when using this function:

- The SOUNDEX value always begins with the first letter in the input string.
- SOUNDEX uses only the first five consonants in the string to generate the return value.
- Only consonants are used to compute the numeric portion of the SOUNDEX value. Except for leading vowels, all vowels are ignored.
- SOUNDEX is not case sensitive; uppercase and lowercase letters return the same SOUNDEX value.

The SOUNDEX function is useful for ad hoc queries, and any other kinds of searches where the exact spelling of a database value is not known or easily determined.

The SOUNDEX algorithm is English-centric and may not work well (or at all) for other languages.

SUBSTR(`string1`, `start`, `length`)

Returns a substring from *string1*, beginning with the character at position *start* and going for *length* characters. If the end of *string1* is encountered before *length* characters are found, then all characters from *start* onward are returned. The following variations exist:

SUBSTR(`string1`, `start`)

Returns all characters beginning from position *start* through to the end of *string1*.

SUBSTR(`string1`, `negative_start`, `length`)

Counts backward from the end of *string1* to determine the starting position from which to begin returning *length* characters.

SUBSTR(`string1`, `negative_start`)

Returns the last ABS(*negative_start*) characters from the string.

SUBSTR treats a string as a sequence of characters. The variations SUBSTRB, SUBSTR2, and SUBSTR4 treat a string as a sequence of bytes, Unicode code units, and Unicode code points, respectively. The variation SUBSTRC treats a string as a series of complete Unicode characters. For example, 'a\0303', which is the decomposed equivalent of '\00E3', or *ã*, is treated and counted as a single character. SUBSTR, however, sees 'a\0303' as two characters.

TO_CHAR(national_character_data)
: Converts data in the national character set to its equivalent representation in the database character set. See also TO_NCHAR.

> TO_CHAR may also be used to convert date and time values, as well as numbers, into human-readable form. These uses of TO_CHAR are described in Chapter 9 (for numbers) and Chapter 10 (for dates and times).

TO_MULTI_BYTE(string1)
: Translates single-byte characters to their multibyte equivalents. Some multibyte character sets, notably UTF-8, provide for more than one representation of a given character. In UTF-8, for example, letters such as *G* can be represented using one byte or four bytes. TO_MULTI_BYTE lets you convert from the single-byte to the multibyte representation. TO_MULTI_BYTE is the opposite of TO_SINGLE_BYTE.

TO_NCHAR(database_character_data)
: Converts data in the database character set to its equivalent representation in the national character set. See also TO_CHAR and TRANSLATE...USING.

> TO_NCHAR may also be used to convert date and time values, as well as numbers, into human-readable form. These uses of TO_NCHAR are described in Chapter 9 (for numbers) and Chapter 10 (for dates and times).

TO_SINGLE_BYTE(string1)
: Translates multibyte characters to their single-byte equivalents. This is the opposite of TO_MULTI_BYTE.

TRANSLATE(string1, search_set, replace_set)
: Replaces every instance in *string1* of a character from *search_set* with the corresponding character from *replace_set*. For example:

```
TRANSLATE ('abcd', 'ab', '12') --> '12cd'
```

If the search set contains more characters than the replace set, then the "trailing" search characters that have no match in the replace set are not included in the result. For example:

```
TRANSLATE ('abcdefg', 'abcd', 'zyx') --> 'zyxefg'
```

The letter *d* is removed, because it appears in *search_set* without a corresponding entry in *result_set*. TRANSLATE swaps individual characters, while REPLACE swaps strings.

TRANSLATE(`text` USING CHAR_CS) and TRANSLATE(`text` USING NCHAR_CS)
Translates character data to either the database character set (CHAR_CS) or the national character set (NCHAR_CS). The output datatype will be either VARCHAR2 or NVARCHAR2, depending on whether you are converting to the database or the national character set, respectively.

> TRANSLATE...USING is an ISO standard SQL function. Starting with Oracle9*i* Database Release 1, you can simply assign a VARCHAR2 to an NVARCHAR2 (and vice versa), and the database will handle the conversion implicitly. If you want to make such a conversion explicit, you can use TO_CHAR and TO_NCHAR to convert text to database and national character sets, respectively. Oracle Corporation recommends the use of TO_CHAR and TO_NCHAR over TRANSLATE...USING, because those functions support a greater range of input datatypes.

TRIM(FROM `string1`)
Returns a version of *string1* that omits any leading and trailing spaces. Variations include:

TRIM(LEADING FROM ...)
Trims only leading spaces.

TRIM(TRAILING FROM ...)
Trims only trailing spaces.

TRIM(BOTH FROM ...)
Explicitly specifies the default behavior of trimming *both* leading and trailing spaces.

TRIM(...`trim_character` FROM `string1`)
Removes occurrences of *trim_character*, which may be any one character that you want to specify.

Oracle added the TRIM function in Oracle8*i* Database to increase compliance with the ISO SQL standard. TRIM comes close to combining the functionality of LTRIM and RTRIM into one function. The difference is that with TRIM, you can specify only one trim character. When using LTRIM or RTRIM, you can specify a set of characters to trim.

UNISTR(string1)

Returns *string1* converted into Unicode. This is the opposite of ASCIISTR. You can represent nonprintable characters in the input string using the \XXXX notation, where *XXXX* represents the Unicode code point value for a character. For example:

```
BEGIN
   DBMS_OUTPUT.PUT_LINE(
      UNISTR('The symbol \20AC is the Euro.')
   );
END;

The symbol € is the Euro.
```

The output of the code is:

```
The symbol € is the Euro.
```

UNISTR gives you convenient access to the entire universe of Unicode characters, even those you cannot type directly from your keyboard. Chapter 25 discusses Unicode in more detail.

UPPER(string1)

Returns a version of *string1* with all letters made uppercase. The return datatype is the same as the datatype of *string1* (CHAR, VARCHAR2, or CLOB). UPPER is the opposite of LOWER. See also NLS_UPPER.