APPENDIX B

CGI PROGRAMMING TECHNIQUES IN PERL

Perl (Practical Extraction and Reporting Language) is not a CGI-specific programming language. In fact, it is a powerful language with many applications far beyond the needs of CGI. Thus, as a CGI programmer, your mastery of Perl initially need only extend to a small subset of the Perl universe.

In this appendix, we will try to identify the most commonly appearing Perl functions used with the CGI applications in this book to give the beginner a quick-and-dirty, but by no means all-inclusive, introduction to Perl. If you are a beginner to Perl as well as to CGI, this appendix should give you the very basic foundation you will need in order to understand the scripts in this book. However, intermediate and advanced readers should only selectively browse this appendix as needed. Most of the information here should already be familiar to you.

If you would like more than a cheat-sheet, we strongly recommend that find *Learning Perl* by Randall Schwartz and *Programming Perl* by Randall Schwartz and Larry Wall, both of which are published by O' Reilly and Associates Inc. Both these books outline Perl completely and, as such, are invaluable resources for the CGI programmer.

In the meantime, let this appendix be your guide.

Discussion

Sending Text to the Web Browser

Every CGI application must output information. For example, both the HTTP header and the HTML code necessary to generate whatever graphical user interface (GUI) the client will be using to navigate must be sent to the Web browser.

USING THE PRINT FUNCTION

The most basic method for sending text to the Web browser is the **print** function in Perl. The print function uses the following syntax:

```
print "[string to print]";
```

By default, the **print** function outputs data to standard output "<STDOUT>" which, in the case of a CGI application, is the Web browser. Thus, whatever you tell Perl to print will be sent to the Web browser to be displayed.

For example, the following line sends the phrase, "Hello Universe" to the Web browser:

```
print "Hello Universe";
```

Of course, in order to comply with HTTP protocol, you must first send the HTTP header when communicating with a browser using the following syntax:

```
print "Content-type: text/html\n\n";
```

However, **print** does have some limitations. For example, it is limited in its ability to handle Perl special characters within an output string. For example, suppose we want to print the following HTML code:

```
<A HREF = "mailto:selena@foobar.com">selena@foobar.com</A>
```

You might extrapolate from the syntax above that you would use the following Perl code to display the hyperlink:

```
print "<A HREF = "mailto:selena@foobar.com">selena@foobar.com</A>";
```

Unfortunately, this would yield a syntax error. Additionally, because this is a very common line of HTML, it is a common source of Perl CGI customization errors. The problem lies in the incorporation of the at sign (@) and double-quote (“) characters within the code.

As it so happens, these characters are “special” Perl characters. In other words, each haas a special meaning to Perl and, when displaying them, you must take precautions so that Perl understands what you are asking for. For example, consider the double quote marks in the “mailto” hyperlink. How would Perl know that the double quote marks in the “mailto” hyperlink are supposed to be part of the string to be printed and not actually the end of the string to be printed? Recall that we use the double quote marks to delineate the beginning and the ending of a text string to be printed. Similarly, the at sign (@) is used by Perl to name list arrays.


NOTE

Many other “special” characters exist and are discussed in other Perl references.

One solution to this problem is to escape the Perl special characters with a backslash (\). The backslash character tells Perl that whatever character follows should be considered a part of the string and not a special character. Thus, the correct syntax for the **mailto** hyperlink would be

```
print “<A HREF =
\”mailto:selena\@foobar.com\”>selena\@foobar.com</A>”;
```

USING “HERE DOCUMENTS”

Unfortunately, much of what your CGI applications will be sending to the Web browser will include the double-quote mark special character. It becomes tedious, especially for long blocks of HTML code, to make **print** statements for every line of HTML and to escape every occurrence of a double-quote with a backslash. Consider the following table definition:

```
print “<TABLE BORDER = \”1\” CELLPADDING = \”2\”
              CELLSPACING = \”2\”>”;
print “<TR>”;
print “<TD ALIGN = \”center\”>Email</TD>”;
```

```
print "<TD ALIGN = \"center\">
       <A HREF = \"mailto:selena\@foobar.com\">
       selena\@foobar.com</A></TD>";
print "</TR>";
print "</TABLE>";
```

If any one of those backslashes is missing, the whole script breaks down. And this is a very small block of code!

One solution to the sending of large blocks of HTML code that incorporate the double-quote is to use the "here document" method of printing. The "here document" method tells Perl to print everything within a certain block of boundaried code. The "here document" uses the generic format:

```
print <<[TEXT_BOUNDARY_MARKER];
[Text to be printed]
[TEXT_BOUNDARY_MARKER]
```

For example, this code will print out the basic HTML header:

```
print <<end_of_html_header;
<HTML>
<HEAD>
<TITLE>My Title</TITLE>
</HEAD>
<BODY>
end_of_html_header
```

In short, the "here document" method of printing tells the Perl interpreter to print out everything it sees (**print <<)** from the **print** line until it finds the text boundary marker specified in the print line (**end_of_html_header**). The text boundary marker can be anything you like, of course, but it is useful to make the flag descriptive.

Further, the ending flag must be *exactly* the same as the flag definition. Thus, the following code will fail because the final flag is not indented correctly:

```
print <<"    end_of_header";
  <HTML><HEAD><TITLE>$title</TITLE></HEAD><BODY>
  end_of_header
```

The final **end_of_header** tag should have been indented four spaces, but it was only indented two.

Though the "here document" method of printing does avoid having to escape double quotes within the block to print, the at sign (@) and other special Perl characters still need escaping.

USING QQ

qq is another Perl trick that helps a programmer solve the double-quote problem by allowing her to change the double-quote delimiter in a print statement.

Normally, as we said, double-quotes (") are used to delimit the characters in a print statement. However, by replacing the first quote with two *q*s followed by another character, that final character becomes the new print statement delimiter. Thus, by using **qq!**, we tell Perl to use an exclamation point (!), also known as "bang,"to delimit the string instead of the double quotes.

For example, without using **qq**, a print statement that outputs, "She said, 'hi'." would be written as:

```
print "She said, \"hi\".";
```

But with the **qq** making bang (!) the new delimiter, the same statement can be written as:

```
print qq!She said, "hi"!;
```

Why would we do this? Readability. If the print statement was surrounded with the normal double quotes, then every double-quote would have to be escaped with a backslash whenever it was used within a string. The backslashes clutter the readability of the string. Thus, we choose a different character to delimit the string in the print statement so that we do not have to escape the double-quotes with backslashes.

USING THE PRINTF AND SPRINTF FUNCTIONS

The Perl **printf** is much like the **printf** function in C and awk in that it takes a string to be formatted and a list of format arguments, applies the formatting to the string, and then typically prints the formatted string to standard output, which in our case, is the Web browser.

The **printf** syntax uses a double-quoted string that includes special format markers followed by a comma-delimited list of arguments to be applied to those markers. The format markers are typically in the form of a percent sign followed by a control character.

For example, the generic format of **printf** might look like the following code:

```
printf ("[some text] %[format] [other text]", [argument to be formatted]);
```

In use, we might use the **%s** formatting argument specifying a string and the **%d** formatting argument specifying a digit using the following syntax:

```
$name = "Selena Sol";
$age = 27;
printf ("My name is %s and my age is %d.\n", $name, $age);
```

The code above would produce the following output in the Web browser window:

```
My name is Selena Sol and my age is 27.
```

In reality, the **printf** function is rarely used in Perl CGI since, unlike C which almost demands the use of **printf**, Perl has much easier ways of printing. However, the **printf** routines are essential for another, more useful (to CGI developers) function, **sprintf**.

Unlike **printf**, **sprintf** takes the formatted output and assigns it to a variable, rather than outputting it to standard output (<STDOUT>), using the following generic syntax:

```
$variable_name = sprintf ("[some text] %[format] [other text]",
[string to be formatted]);
```

A good example of using **sprintf** comes from the **format_price** subroutine in **web_store.cgi**. In this subroutine, we need to format subtotals and grand totals to two decimal places so that prices come out to numbers like "$99.00" or "$98.99" rather than "99" or "98.99876453782". Below is a snippet of code that uses **sprintf** to format the price string to two decimal places.

```
$formatted_price = sprintf ("%.2f", $round_price);
```

In this example, the variable, **$round_price** is formatted using the **"%.2f"** argument which formats (%) the string to two decimal places (**.2f**) and assigned to the **$formatted_price** string.

There are a multitude of formatting arguments besides **"%s"**, **"%d"**, and **"%f"**, however. Table B.1 lists several useful ones.

Table B.1 printf and sprintf Formats

Format Character	Description
c	Character
s	String
d	Decimal Number
x	Hexadecimal Number
o	Octal Number
f	Floating Point Number

Scalar Variables, List Arrays, and Associative Arrays

WHAT IS A SCALAR VARIABLE?

You can think of a variable as a place holder or a name that represents one or more values. The generic syntax for defining scalar variables (also known as *variables* for short) is as follows:

```
$variable_name = value;
```

Thus, for example, we might assign the value of 27 to the scalar variable named "age" with the syntax:

```
$age = 27;
```

The dollar sign (**$**) is used to let Perl know that we are talking about a scalar variable. From then on, unless we change the value of **$age**, the script will translate it to **27**.

So if we then say:

```
print "$age\n";
```

Perl will send the value **27** to standard output, which in our case, will be the Web browser.

If we are assigning a word or a series of words to a scalar variable rather than just a number, we must mark the boundary of the value with single or double quotes so that Perl will know exactly what should be assigned to the scalar variable.

We use single quotes to mark the boundary of a plain text string and we use double quotes to mark the boundary of a text string that can include scalar variables to be "interpolated." For example, we might have the following lines:

```
$age = 27;
$first_name = 'Selena';
$last_name = 'Sol';
$sentence = "$first_name $last_name is $age";
print "$sentence\n";
```

The routine would print the following line to standard output:

```
Selena Sol is 27
```

Notice that the scalar variable **$sentence** is assigned the actual values of **$first_name** and **$last_name**. This is because they were "interpolated" since we included them within double quotes in the definition of **$sentence**. There is no interpolation inside single quotes. Thus, if we had defined **$sentence** using single quotes as follows:

```
$sentence = '$first_name $last_name is $age';
```

Perl would print the following to standard output:

```
$first_name $last_name is $age
```

USING SCALAR VARIABLES

The benefit of substituting a scalar variable name for a value is that we can then manipulate its value. For example, you can autoincrement a scalar variable using the **++** operator:

```
$number = 1;
print "$number\n";
$number++;
print "$number\n";
```

Perl would send the following to standard output:

```
1
2
```

You can also perform arithmetic such as:

```
$item_subtotal = $item_price * $quantity;
$shipping_price = 39.99 * $quantity;
$grand_total = $item_subtotal + $shipping_price;
```

Scalar variables are the meat and potatoes of CGI. After all, translating between the client and the Web server is essentially the formatting and the reformatting of variables. Be prepared to see them used a lot.

USING THE DOT OPERATOR

Another cool Perl trick is the use of the "dot" (.) operator, which "appends" a value to an already existing scalar variable. Thus, the following code would print out **Selena Sol**:

```
$name = "Selena" . " Sol";
print "$name";
```

An alternative shorthand for appending to scalar variables is using the **.=** operator. For example, the following code does the same thing as the code above:

```
$name = "Selena";
$name .= " Sol";
print "$name\n";
```

CROPPING SCALAR VARIABLES WITH THE CHOP FUNCTION

Sometimes, you do not want the entire value that has been assigned to a scalar variable. For example, it is often the case that the lines you retrieve from a data file will incorporate a newline character at the end of the line. In this book, data files often take advantage of the newline character as a "database row delimiter." That is, every line in a database file is a new database item. For example, here is a snippet from sample Cart data file:

```
3|0011|Vowels|12.98|The letter E|12.98|100
2|0010|Vowels|15.98|The letter A|15.98|101
```

When the script reads each line, it also reads in the newline information. Thus, the first line is actually represented as:

```
3|0011|Vowels|12.98|The letter E|12.98|100\n
```

The final **\n** is a new line. Since we do not actually want the **\n** character included with the last database field, we use the chop function. The chop function chops off the very last character of a scalar variable using the syntax:

```
chop ($variable_name);
```

Thus, we would take off the final newline character as follows:

```
$database_row = "3|0011|Vowels|12.98|The letter
                 E|12.98|100\n";
chop ($database_row);
```

FINDING THE LENGTH OF A SCALAR VARIABLE WITH THE LENGTH FUNCTION

Finding the length of a scalar variable is incredibly easy using the **length** function. The syntax of length is as follows:

```
length ([$variable_name]);
```

Thus, if the scalar variable **$name** equals "Selena," then the scalar variable **$length_of_name** will be assigned the value of **6** in the following line:

```
$length_of_name = length ($name);
```

MANIPULATING SUBSTRINGS WITH THE SUBSTR FUNCTION

Sometimes, you want to work with just part of a string that has been assigned. The **substr** function can be used for that and follows the syntax:

```
$substring = substr([string you want to extract from],
                    [beginning point of extraction],
                    [length of the extracted value]);
```

For instance, to assign **Sol** to the scalar variable **$last_name** you would use the following code:

```
$name = "Selena Sol";
$last_name =  substr ($name, 7, 3);
```

The **substr** function takes the scalar variable **$name**, and extracts three characters beginning with the seventh:

As in array indexing, the substr function counts from zero, not from one. Thus, in the string **Gunther**, the letter *t* is actually referenced as **3** not **4**.

The final number (length of extracted value) is not necessary when you want to grab everything "after" the beginning character. Thus, the following code will do just what the previous did since we are extracting the entire end of the variable $name:

```
$last_name =  substr ($name, 7);
```

List Arrays

What Is a List Array?

List arrays (also known simply as *arrays* for short) take the concept of scalar variables to the next level. Whereas scalar variables associate one value with one variable name, list arrays associate one array name with a "list" of values.

A list array is defined with the following syntax:

```
@array_name = ("element_1", "element_2"..."element_n");
```

For example, consider the following list array definition:

```
@available_colors = ("red", "green", "blue", "brown");
```

As you might have guessed, the at sign (@) is used to communicate to Perl that a list array is being named much as the dollar sign ($) is used to denote a scalar variable name.

In this example, the list array **@available_colors** is filled with four color "elements" in the specific order: red, green, blue, brown. It is important to see that the colors are not simply dumped into the list array at random. Each list element is placed in the specific order in which the list array was defined. Thus list arrays are also considered to be *ordered.*

USING A LIST ARRAY

The benefit of ordering the elements in a list array is that we can easily grab one value out of the list on demand. To do this, we use Perl's subscripting operator using the format:

```
$array_name[list_element_number]
```

When pulling an element out of a list array, we create a scalar variable with the same name as the array, prefixed with the usual dollar sign denoting scalar variables.

For example, the first element of the array **@available_colors** is accessed as:

```
$available_colors[0].
```

Notice that the first element is accessed with a zero. This is important. List arrays begin counting at zero, not one. Thus, **$available_colors[0]** is a variable place holder for the word *red*. Likewise, **$available_colors[1]** equals *green* and **$available_colors[2]** equals *blue*.

FIGURING OUT HOW MANY ELEMENTS ARE IN AN ARRAY

Fortunately, Perl provides an easy way to determine how many elements are contained in an array. When used as a scalar, the list array name will be equal to the number of elements it contains. Thus, if the list array **@available_colors** contains the elements: red, green, blue, and brown, then the following line would set **$number_of_colors** equal to 4:

```
$number_of_colors = @available_colors;
```


WARNING

Be careful when using this value in your logic. The number of elements in an array is a number counting from 1. But when accessing an array, you must access starting from 0. Thus, the last element in the array **@available_colors** is not **$available_colors[@available_colors]** but rather **$available_colors[@available_colors - 1]**.

ADDING ELEMENTS TO A LIST ARRAY

Likewise, you can add to or modify the values of an existing array by simply referencing the array by number. For example, to add an element to **@available_colors**, you might use the following line:

```
$available_colors[4] = "orange";
```

Thus, **@available_colors** would include the elements: red, green, blue, brown, and orange.

You can also use this method to overwrite an element in a list array. To change a value in **@available_colors**, you might use the syntax:

```
$available_colors[0] = "yellow";
```

Now, the elements of **@available_colors** would be: yellow, green, blue, brown, orange.

DELETING AND REPLACING LIST ELEMENTS WITH THE SPLICE FUNCTION

The splice function is used to remove or replace elements in an array and uses the following syntax:

```
splice ([array to modify], [offset], [length],
        [list of new elements]);
```

The **array** argument is the array to be manipulated. **offset** is the starting point where elements are to be removed. **length** is the number of elements from the offset number to be removed. The **list** argument consists of an ordered list of values to replace the removed elements with. Of course, if the **list** argument is null, the elements accessed will be removed rather than replaced.

Thus, for example, the following code will modify the **@numbers** list array to include the elements, ("1" , "2", "three", "four", "5").

```
@numbers = ("1", "2", "3", "4", "5");
splice (@numbers, 2, 2, "three", "four");
```

A more common usage of the splice is simply to remove list elements by not specifying a replacement list. For example, we might modify **@numbers** to include only the elements "1", "2", and "5" by using the following code:

```
splice (@numbers, 2, 2);
```

ADVANCED LIST ARRAY MANIPULATION WITH THE PUSH, POP, SHIFT, AND UNSHIFT FUNCTIONS

Of course, once we have created a list array, we can do much more than just access the elements. We can also manipulate the elements in many ways. Throughout this book, list arrays are most often manipulated using the operators **push**, **pop**, **shift**, and **unshift**:

push is used to add a new element on the right-hand side of a list array.

Thus, the following code would create a list array of ("red", "green", "blue"):

```
@colors = ("red", "green");
push (@colors, "blue");
```

In other words, the **push** operator adds an element to the end of an existing list.

pop does the exact same thing as push, but in reverse. It extracts the right-side element of a list array using the following syntax:

```
$popped_variable_name = pop (@array_name);
```

Thus, we might pop out the value blue from **@colors** with the following syntax:

```
$last_color_in_list = pop (@colors);
```

Thus, the **@colors** array now contains only "red" and "green" and the variable **$last_color_in_list** is equal to "blue".

unshift does the exact same thing as **push**, but it performs the addition to the left side of the list array instead of to the right. Thus, we would create the list ("blue", "red", "green") with the following syntax:

```
@colors = ("red", "green");
unshift (@colors, "blue");
```

Similarly, **shift** works the same as **pop**, but to the left side of the list array. Thus, we reduce **@colors** to just "red" and "green" by shifting the first element blue with the following syntax:

```
$first_color_in_list = shift(@colors);
```

Thus, **@colors** again contains only "red" and "green" and **$first_color_in_list** equals blue.

Though **push**, **pop**, **shift**, and **unshift** are the most common list array manipulation functions used in this book, there are many others covered in more complete references. Table B.3 Summarizes some of the common array manipulating operators.

Table B.2 Array Manipulation Operators

Operator	Description
shift(@array)	Removes the first element in **@array**
unshift (@array, $element)	Adds **$element** to the beginning of **@array**
pop (@array)	Removes the last element of **@array**
push (@array, $element)	Adds $element to the end of **@array**
sort (@array)	Sorts the elements in **@array**
reverse(@array)	Reverses the order of the elements in**@array**
chop (@array)	chops off the last character of every element in **@array**
split (/delimiter/, string)	Creates an array by splitting a string
join (delimiter, @array)	Creates a scalar of every element in **@array** joined by the delimiter.

Associative Arrays

WHAT IS AN ASSOCIATIVE ARRAY?

Associative arrays add the final degree of complexity allowing ordered lists to be associated with other values. Unlike list arrays, associative arrays have index values that are not numbers. You do not reference an associative array as **$associative_array_name[0]** as you did for the list array. Instead, associative arrays are indexed with arbitrary scalar variables. Consider the following associative array definition:

```
%CLIENT_ARRAY = ('full_name', 'Selena Sol',
                 'phone', '213-456-7890',
                 'age', '27');
```

In this example, we have defined the associative array %CLIENT_ARRAY to have three sets of associations.


NOTE

the percent sign (%) denotes the associative array name just as the dollar sign ($) did for variables and the at sign (@) did for list arrays.

Thus, **full_name** is associated with **Selena Sol** as **age** is associated with **27**. This association is discussed in terms of **keys** and **values**. Each key is associated with one value. Thus, we say that the key **full_name** is associated with the value **Selena Sol**.

ACCESSING AN ASSOCIATIVE ARRAY

If we want to extract a value from the associative array, we reference it with the following syntax:

```
$variable_equal_to_value = $ASSOCIATIVE_ARRAY_NAME{'[key]'};
```

Thus, to pull out the value of the **name** key from **%CLIENT_ARRAY**, we use the following syntax:

```
$full_name = $CLIENT_ARRAY{'full_name'}
```

The variable **$full_name** would then be equal to **Selena Sol**. Think of it as using a **key** to unlock a **value**.


NOTE

When accessing an associative array using a scalar variable as a key, you should not surround the key with single quotes because the scalar variable will not be interpolated. For example, the following syntax generates the value for the age key:

```
$key_name = "age";
$age = $CLIENT_ARRAY{$key_name};
```

Accessing an associative array is one of the most basic CGI functions and is at the heart of the **ReadParse** routine in **cgi-lib.pl**, which creates an associative array from the incoming form data. By accessing this associative array (usually referred to in this book as **%form_data**), **web_store.cgi** is able to determine

what it is that the client has asked of it since HTML form variables are formed in terms of administratively defined NAMES and client-defined VALUES using syntax such as the following:

```
<INPUT TYPE = “text” NAME = “full_name” SIZE = “40”>
```

The “key” of the associative array generated by **ReadParse** will be **full_name** and the **value** will be whatever the client typed into the text box.

USING THE KEYS AND VALUES FUNCTIONS

Perl also provides a convenient way to get a list of all the keys or of all the values in an associative array if you are interested in more than just one key/value pair. keys and values are accessed with the keys and values functions using the following formats:

```
@associative_array_keys = keys (%ASSOCIATIVE_ARRAY_NAME);
```

and

```
@associative_array_values = values (%ASSOCIATIVE_ARRAY_NAME);
```

Thus, the keys and values list of the associative array **%CLIENT_ARRAY** defined above can be generated with the following syntax:

```
@client_array_keys = keys (%CLIENT_ARRAY);
@client_array_values = values (%CLIENT_ARRAY);
```

In this example **@client_array_keys** would look like (**full_name**, **phone**, **age**) and **@client_array_values** would look like (**Selena Sol**, **213-456-7890**, **27**).

ADDING TO AND DELETING FROM AN ASSOCIATIVE ARRAY

Like list arrays, associative arrays can be internally modified. The most common function, other than defining an associative array, is adding to it. Adding to an associative array simply involves telling Perl which key and value to add using the format:

```
$ARRAY_NAME{‘key’} = “value”;
```

or, using our example above:

```
$CLIENT_ARRAY{'favorite_candy'} = "Hershey's with Almonds";
```

%CLIENT_ARRAY now includes **full_name**, **phone**, **age** and **favorite_candy** along with their associated values.

Similarly, you can easily use the delete function to delete a key/value pair in an associative array. The **delete** function follows the syntax:

```
delete ($ASSOCIATIVE_ARRAY_NAME{'key'});
```

or for our %CLIENT_ARRAY example:

```
delete ($CLIENT_ARRAY{'age'});
```

Thus, **%CLIENT_ARRAY** would contain only **full_name**, **phone**, and **favorite_candy**.

Manipulating Strings

Another important function provided by CGI is the manipulation of strings of data. Whether called upon to display or manipulate the contents of a data file, to reformat some text for Web-display, or simply to use in some logical routine or external program, Perl has a diverse array of string modification functions at its disposal.

EQUALITY OPERATORS

One of the most important string manipulation functions is that of matching or testing of equality. It is an important tool because you can use it as the basis of complex logical comparisons necessary for the intelligence demanded of a CGI application.

For example, consider one of the most basic methods of pattern matching, the **ne** operator, which is used as the basis of the decision making process in **web_store.cgi**:

```
if (the user has hit a specific submit button)
  {
  execute a specific routine.
  }
```

Consider this code snippet:

```
if ($return_to_frontpage_button ne “”)
  {
  &display_frontpage;
  }
```

If you are confused about the usage of the “if” test, it is explained in greater detail in the “Control Structures” section later in this appendix.

The **ne** operator asks if the value of the variable **$return_to_frontpage_button** is not equal to an empty string. This logic takes advantage of the fact that the HTTP protocol specifies that if a FORM Submit button is pressed, its NAME is set equal to the VALUE specified in the HTML code. For example, the Submit button may have been coded using the following HTML:

```
<INPUT TYPE = “submit”
       NAME = “return_to_frontpage_button”
       VALUE = “Return to the Frontpage”>
```

Thus, if the NAME in the associative array has a VALUE, the script knows that the client pushed the associated button. The script determines which routines it should execute by following the logic of these pattern matches.

Similarly, you can test for equality using the **eq** operator. An example of the **eq** operator in use is shown below:

```
if ($name eq “Selena”)
  {
  print “Hi, Selena\n”;
  }
```

When comparing numbers instead of strings however, Perl uses a second set of operators. For example, to test for equality, you use the double equal (==) operator as follows:

```
if ($number == 11)
  {
  print “You typed in 11\n”;
  }
```

Never use the single equal sign (=) for comparison. Perl interprets the equal sign in terms of assignment rather than comparison. Thus the line:

```
$number = 11;
```

actually assigns the value of 11 to **$number** rather than comparing **$number** to 11.

There are many other types of comparison operators, but they are better researched in more comprehensive texts. However, we do include several important ones in Table B.5

Table B.3 Numeric and String Comparison Operators

Numeric Operator	String Operator	Description
==	eq	Equal
!=	ne	Not equal
<	lt	Less than
>	gt	Greater than
<=	le	Less than or equal to
>=	ge	Greater than or equal to

REGULAR EXPRESSIONS

Regular expressions are one of the most powerful, and hence, most complicated tools for matching strings. You can think of a regular expression as a "pattern," which can be used to match against some string. Regular expressions are far more versatile than the simple **eq** and **ne** operators and include a wide variety of modifiers and tricks. Other books have detailed chapters focusing on the use of regular expressions, so we will only touch upon a few common uses of regular expressions found this book.

Pattern Matching with //

Perl invokes a powerful tool for pattern matching that gives the program great flexibility in controlling matches. In Perl, a string is matched by placing it between two slashes as follows:

```
/[pattern_to_match]/
```

Thus, **/eric/** matches for the string "eric." You may also match according to whole classes of characters using the square brackets ([]). The pattern match will then match against any of the characters in the class. For example, to match for any single even-numbered digit, you could use the following match:

```
/[02468]/
```

For classes including an entire range of characters, you may use the dash (-) to represent the list. Thus, the following matches any single lower-case letter in the alphabet:

```
/[a-z]/
```

Likewise, you may use the caret (^) character within the square brackets to match every character that is "not" in the class. The following code matches any single character which is not a digit:

```
/[^0-9]/
```

Matching Operators

Further, the **//** operator can be modified to include complex pattern matching routines. For example, the period (.) matching operator is used to stand for "any" character. Thus, **/eri./** would match any occurrences of "eric" as well as "erik."

Another commonly used matching operator is the asterisk (*). The asterisk matches zero or more occurrences of the character preceding it. Thus, **/e*ric/** matches occurrences of "eeeeeric" as well as "eric."

Table B.4 includes a list of useful matching operators.

Table B.4 Commonly Used Matching Operators

Operator	Description
\n	Newline
\r	Carriage return
\t	Tab
\d	Digit (same as [0–9])
\D	Any non-digit (same as [^0-9])
\w	A Word Character (same as [0-9a-zA-Z_])
\W	A nonword character
\s	Any whitespace character (\t, \n, \r, or \f)
\S	A nonwhitespace character
*	Zero or more occurrences of the preceding character
+	One or more occurrences of the preceding character
.	Any character
?	Zero or one occurrences of the preceding character

Anchors

Regular expressions also take advantage of anchoring patterns which help match the string in relationship to the rest of the line. For example, the **\b** anchor is used to specify a word boundary. That is, **/\beric\b/** matches "eric", but it does not match **generic**.

Similarly, the caret (^) anchor will match a string to the beginning of the line. Thus, **/^eric/** will match the following line:

```
eric is my name
```

but it will not match:

```
my name is eric
```


WARNING

The caret (^) can be confusing since it is used as an anchor when included "outside" of the square brackets ([]) but is used as the **not** operator for a class when used "within."

Table B.5 summarizes a few of the most common anchors.

Table B.5 Common Anchors

Anchor	Description
^	Matches the beginning of the string
$	Matches the end of the string
\b	Matches a word boundary (between \w and \W)
\B	Matches on non-word boundary

String Modifiers

Finally, pattern matching can be used to modify strings of text. One of the most common methods of modification is substitution. Substitution is performed using the format:

```
s/[pattern_to_find]/[pattern_to_replace_with]/
```

Thus, for example, the line:

```
s/eric/selena/
```

would change the line

```
eric is my name
```

to

```
selena is my name
```

The substitution function is modified most commonly with the **/i** and the **/g** arguments. The **/i** argument specifies that matching should be done with case insensitivity and the **/g** specifies that the match should occur globally for the entire string of text rather than just for the first occurrence.

Thus, the line:

```
s/eric/selena/gi
```

would change the line:

```
I am Eric, eric I am
```

to:

```
I am selena, selena I am
```

without the **/i**, you would get:

```
I am Eric, selena I am
```

and without **/g** but with the **/i**, you would get:

```
I am selena, eric I am
```

There are many, many different kinds of matching operators, anchors, and string modifiers. If you want a more detailed explanation we recommend that you find a good reference source on regular expressions. Otherwise, the above discussion shouldbe sufficient to explain how we use operators and anchors in this book.

THE =~ OPERATOR

Pattern matching can also be used to manipulate variables. In particular, the scripts in this book take advantage of the **=~** operator in conjunction with the substitution operator using the format:

```
$variable_name =~ s/[string_to_remove]/[string_to_add]/gi;
```

For example, if we want to censor every occurrence of the word *Frack* from the client-defined input field "comment," we might use the line:

```
$form_data{'comments'} =~ s/frack/censored/gi;
```

USING THE SPLIT AND JOIN FUNCTIONS

Finally, regular expressions can be used to split a string into separate fields. To do so, we use the **split** function with the format:

```
@split_array = split (/[pattern_to_split_on]/, [string_to_split]);
```

For example, the applications in this book often use the **split** function to read the fields of database rows. Consider the following code snippet:

```
$database_row = "Selena Sol|213-456-7890|27";
@database_fields = split (/|/, $database_row);
```

Now **@database_fields** will include the elements **Selena Sol**, **213-456-7890**" and **27**. Each of these fields can then be processed separately if need be.

The reverse operation is performed with the **join** function, which uses the following format:

```
$joined_string = join ("[pattern_to_join_on]", [list_to_join]);
```

Thus, we might recreate the original database row using:

```
$new_database_row = join ("\|", @database_fields);
```


NOTE

Notice that in the above line, the pipe (|) symbol must be escaped with a backslash (\) because the pipe is a special Perl character.

Control Structures

Some of the most powerful tools of Perl programming are control structures. Control structures are used to create the basic logic that drives many of the routines used in CGI applications. These control structures use Boolean logic to imbue your script with the intelligence necessary to manage the diverse needs of the clients with the abilities and requirements of the server.

STATEMENT BLOCKS

All control structures are divided into the control statement (which we will explain below) and the statement block. The *statement block* is simply a group of commands that are executed together. This block is grouped by enclosing the commands within curly braces ({}). For example, the following is a simple statement block:

```
{
  statement one
  statement two
  statement three
}
```

Perl will execute each statement in a statement block from beginning to end as a group. When, how, or if the script will execute the commands, however, is determined by the control statement.

USING THE IF, ELSIF, ELSE AND UNLESS CONTROL STATEMENTS

The most common control statement used throughout the scripts in this book is the "if" test. The if test checks to see if some expression is true, and if so, executes the routines in the statement block. Perl uses a simple binary comparison as a test of truth. If the result of some operation is true, the operation returns a one and the statement block is executed. If the result is false, it returns a 0, and the statement block is not executed. For example, consider the following code:

```
if ($name eq "Selena Sol")
  {
  print "Hello Selena.\n";
  }
```

In this example, Perl checks to see if the scalar variable **$name** has the value of **Selena Sol**. If the patterns match, the matching operation will return true and the script will execute the print statement within the statement block. If Perl discovers that **$name** is not equal to **Selena Sol**, however, the print will not be executed.


WARNING

Be careful with your usage of **eq** versus **=.** Within an if test, if you write **$name = Selena Sol**, you will actually be *assigning* **Selena Sol** to the variable **$name** rather than *comparing* it to the value **Selena Sol**. Since this action will be performed successfully, the "if" test will always test to true and the statement block will always be performed even if **$name** did not initially equal **Selena Sol**.

The "if" test also provides for alternatives: the **else** and the **elsif** control statements. The **elsif** alternative adds a second check for truth and the **else** alternative defines a final course of action for every case of failed if or elsif tests. The following code snippet demonstrates the usage of if, **elsif**, and **else:**

```
if ($name eq "Selena Sol")
  {
  print "Hi, Selena.\n";
  }
elsif ($name eq "Gunther Birznieks")
  {
  print "Hi, Gunther\n";
  }
else
```

```
  {
  print "Who are you?\n";
  }
```

Obviously, the else need not perform a match since it is a catch-all control statement.

The **unless** control statement works like an inverse **if** control statement. Essentially it says, "execute some statement block unless some condition is true". The unless control statement is exemplified in the code below:

```
unless ($name eq "Selena")
  {
  print "You are NOT Selena!\n";
  }
```

FOREACH

Another very useful control statement is the **foreach** loop. The **foreach** loop iterates through some list and execute a statement block for each iteration. In this book, the **foreach** loop is most commonly used to iterate through a list array. For example, the following code snippet will print out the value of every element in the list array **@names:**

```
foreach $name (@names)
{
print "$name\n";
}
```

WHILE

The **while** loop also performs iteration and is used in this book primarily for reading lines in a file. The while loop can be used to read and print out every line of a file with the following syntax:

```
open ([FILE_HANDLE_NAME], "[filename]");
while (<[FILE_HANDLE_NAME]>)
  {
  print "$_";
  }
close ([FILE_HANDLE_NAME]);
```

The script would print out every line in the file **filename** because the **$_**, the Perl "default" variable, represents "the current line" in this case.


NOTE

The process of opening and closing files is covered in the "File Management" section later in this appendix.

FOR LOOPS

The **for** loop is another excellent control statement tool. The basic syntax of a **for** loop follows:

```
for ([initial condition]; [test]; [incrementation])
  {
  [action to perform]
  }
```

The **initial condition** defines where the loop should begin. The **test** defines the logic of the loop by letting the script know the conditions that determine the scripts actions. The **incrementation** defines how the script should perform the loop. For example, we might produce a visible countdown with the following for loop:

```
for ($number = 10; $number >= 0; $number—)
  {
  print "$number\n";
  }
```

The script would initially assign 10 to the scalar variables **$number**. It would then test to see if **$number** was greater than or equal to 0. Since 10 is greater than 0, the script would decrement **$number** by subtracting 1 from the value of **$number**.


NOTE

To decrement, you use **$variable_name—**. To increment, you use **$variable_name++**.

Executing the statement block, the script would then print out the number nine. Then, it would go back through the loop again and again, printing each decremented numbers until **$number** was less than zero. At that point, the test would fail and the for loop would exit.

USING LOGICAL OPERATORS (&& AND ||)

Control statements can also be modified with a variety of logical operators that extend the breadth of the control statement truth test using the following syntax:

```
[control statement] (([first condition])
                      [logical operator]
                    ([second condition]))
{
[action to be performed]
}
```

For example, the **&&** operator can be translated as "and". In usage, it takes the format used in the following example:

```
if (($first_name eq "Selena") && ($last_name eq "Sol"))
{
print "Hello Selena Sol";
}
```

Translating the logic goes something like this: if the first name is Selena AND the last name is Sol, then print **Hello Selena Sol**. Thus, if **$first_name** was equal to "Selena" but **$last_name** was equal to "Flintstone", the control statement would test as false and the statement block would not be executed.

Notice that we use parentheses to denote conditions. Perl evaluates each expression inside the parentheses independently and then evaluates the results for the entire group of conditions. If either returns false, the entire test returns false. The use of parentheses are used to determine precedence. With more complex comparisons, in which there are multiple logical operators, the parentheses help to determine the order of evaluation.

Similarly, you may wish to test using the double pipe (||) operator. This operator is used to denote an **or**. Thus, the following code would execute the statement block if $first_name was Selena *or* Gunther.

```
if (($first_name eq "Selena") || ($first_name eq "Gunther"))
{
print "Hello humble CGI book author!";
}
```

File Management

OPENING AND CLOSING FILES

One of the main resources that your server provides is a file management system. The scripts in this book, for example, use a multitude of supporting files in the server's file system such as temporary files, counter files, user files, data files, setup files, and libraries. Perl includes several excellent tools for working with these files.

First, Perl gives your scripts the ability to open files using the **open** function. The **open** function allows you to create a *filehandle* with which to manipulate a file. A filehandle is another name for a connection between the script and the server. Often, filehandles manage connections between the script and standard input, output, or error, however, in the case of **open**, any file can be read into a filehandle using the syntax:

```
open ([FILE_HANDLE_NAME], "[filename]");
```

For example, we might open a data file for reading using:

```
open (DATA_FILE, "inventory.dat");
```

In this case, all of the lines of **inventory.dat** will be read into the filehandle **DATA_FILE,** which Perl can then use within the program. However, you must also close a file once you are done with it. The syntax for closing a file is as follows:

```
close ([FILE_HANDLE_NAME]);
```

Finally, Perl gives you the ability to execute an error routine if there is a problem opening a file. The logical operator is sometimes discussed in terms of a short circuit. For instance, the logic of the **or** operator is such that if the first expression evaluates to true, there is no need to evaluate the next. On the other hand, if the first expression evaluates to false, the second expression is executed. Thus, using the double pipe (||) operator, you can specify the default action to perform if an "open" fails. In CGI applications, the alternate action executed is usually something like the subroutine, **CgiDie** located in **cgi-lib.pl**. For example, the following routine would execute the **CgiDie** subroutine if there was a problem opening **data.file**:

```
open (DATA, "data.file") || &CgiDie("Cannot open data.file");
```

Thus, if the script has a problem opening a needed file, the double pipe (||) operator provides a convenient and elegant way to quit the program and report the problem.

READING A FILE LINE BY LINE

An often-used technique in this book for the manipulation of files is the reading of each line of a file. Perhaps we want to check each line for a keyword, or find every occurrence of some marker tag on a line and replace it with some other string. This process is done using a **while** loop as discussed previously. Consider this routine which will print out every line in a data file:

```
open (DATA, "data.file") ||
      &CgiDie ("Cannot open data.file");
while (<ADDRESSES>)
  {
  print "$_";
  }
close (ADDRESSES);
```

Thus, the script would print out every line in the file address.dat because **$_** is Perl's special name for "the current line" in this case:

You can also manipulate the **$_** variable in other ways such as applying pattern matching on it or adding it to an array.

WRITING AND APPENDING TO FILES

You can do more than just read a file of course. You can also open a filehandle for writing with the greater than sign (>) using the syntax:

```
open ([FILE_HANDLE_NAME], ">[filename]");
```

or for appending using the double-greater-than symbol (>>) with the syntax:

```
open ([FILE_HANDLE_NAME], ">>[filename]");
```

The difference between appending and writing is that when you *write* to a file, you overwrite whatever was previously there. When you append to a file, you simply add the new information to the end of whatever text was already there.

If the file that Perl is asked to write or append to does not already exist, Perl will create the file for you.

Typically, when writing to a file, you use the print function. However, instead of printing to standard output, you would specify the filename to print to. Consider the following example:

```
open (TEMP_FILE, ">temp.file") ||
     &CgiDie ("Cannot open temp.file");
print TEMP_FILE "hello there\n";
close (TEMP_FILE);
```

The file **temp.file** will now have the solitary line:

```
hello there
```

DELETING, RENAMING AND CHANGING THE PERMISSIONS OF FILES

Perl also provides you with all the file-management functions typically offered by your operating system. In our experience, the three most utilized functions in CGI scripts are unlink, rename and chmod. unlink is Perl's function for deleting a file from the file system. The syntax is pretty straight forward.

```
unlink ("[filename]");
```

This line of Perl code will delete the file called filename provided that the script has permissions to delete the file.

Your Perl script can also rename a file using the rename function:

```
rename ("[old_filename]", "[new_filename]");
```

In this case, the file's name will be replaced with the new filename specified.

Finally, Perl gives you the ability to affect the permissions of files in the file system using the chmod function. The syntax is also fairly straight forward as follows:

```
chmod (0666, "filename");
```

In this case, "filename" will be made readable and writable by user, group and world.

FILE TESTS

Finally, Perl provides many methods for determining information about files on the file system using "File tests." For the purposes of this appendix, there are too many types of file tests to cover them all in depth. Further, they are covered extensively elsewhere. However, we will note the most frequent syntax of file tests used in this book, which follows the form:

```
if ([filetest] [filename] && [other filetest] [filename])
  {
  do something
  }
```

Consider the following example which checks to see if a file exists (**-e**) and is writable (**-w**) by us, and if so deletes it:

```
if ((-e "temp.file") && (-w "temp.file"))
{
unlink ("temp.file");
}
```

Table B.6 lists several common file tests.

Table B.6 Common File Tests

Test	Description
-r	File or directory is readable
-w	File or directory is writable
-x	File or directory is executable
-o	File or directory is owned by user
-e	File or directory exists
-z	File exists and has zero size
-s	File or directory exists and has non-zero size
-f	Entry is a plain file
-d	Entry is a directory
-T	File is text
-B	File is binary
-M	Modification age in days
-A	Access age in days

OPENING, READING,AND CLOSING DIRECTORIES

As with files, Perl gives you the ability to manage directories. Specifically, Perl allows you to open a directory as a directory handle, read in the current contents of the directory, and then close it again .

To open a directory, you use the following syntax:

```
opendir ([FILE_HANDLE_NAME], "[directory_location]") || &CgiDie
("Can't open [directory_location]");
```

Thus, for example, you might open the directory **/usr/local/etc/www/** with the syntax:

```
opendir (WWW, "/usr/local/etc/www/") || &CgiDie ("Can't open www");
```

As you can see, as with opening files, Perl allows the program to die elegantly in case there is a problem opening the directory. Also, as with file manipulation, you must close a directory after you are through with it using the syntax:

```
closedir ([FILE_HANDLE_NAME]);
```

For example, to close the directory opened above, you use the command:

```
closedir(WWW);
```

Once you have opened a directory, you can also read the contents of the directory with the **readdir** function. For example, the following code snippet assigns all of the filenames in the **WWW** directory to **@filenames**:

```
opendir (WWW, "/usr/local/etc/www/") ||
           &CgiDie ("Can't open www");
@filenames = readdir (WWW);
closedir (WWW);
```


NOTE

If you want to avoid including the "." (current directory) and ".." (root directory) files you can use the **grep** function to avoid including them in the **readdir** function using the syntax:

```
@filenames = grep (!/^\.\.?$/, readdir (FILE_HANDLE_NAME);
```