

25

Perl and CGI (Common Gateway Interface)

Objectives

- To understand basic Perl programming.
- To understand the Common Gateway Interface.
- To understand string processing and regular expressions in Perl.
- To be able to read and write cookies.
- To be able to construct programs that interact with MySQL databases.

This is the common air that bathes the globe.

Walt Whitman

The longest part of the journey is said to be the passing of the gate.

Marcus Terentius Varro

Railway termini. . . are our gates to the glorious and the unknown. Through them we pass out into adventure and sunshine, to them, alas! we return.

E. M. Forster

There comes a time in a man's life when to get where he has to go—if there are no doors or windows—he walks through a wall.

Bernard Malamud

You ought to be able to show that you can do it a good deal better than anyone else with the regular tools before you have a license to bring in your own improvements.

Ernest Hemingway



Outline

- 25.1 Introduction
- 25.2 Perl
- 25.3 String Processing and Regular Expressions
- 25.4 Viewing Client/Server Environment Variables
- 25.5 Form Processing and Business Logic
- 25.6 Server-Side Includes
- 25.7 Verifying a Username and Password
- 25.8 Using DBI to Connect to a Database
- 25.9 Cookies and Perl
- 25.10 Operator Precedence Chart
- 25.11 Web Resources

Summary • Terminology • Self-Review Exercises • Answers to Self-Review Exercises • Exercises

25.1 Introduction

Practical Extraction and Report Language (Perl) is one of the most widely used languages for Web programming today. Larry Wall began developing this high-level programming language in 1987 while working at Unisys. His initial intent was to create a programming language to monitor large software projects and generate reports. Wall wanted to create a language that would be more powerful than shell scripting and more flexible than C, a language with rich text-processing capabilities and, most of all, a language that would make common programming tasks straightforward and easy. In this chapter, we discuss Perl 5.8 and examine several practical examples that use Perl for Internet programming.

The **Common Gateway Interface (CGI)** is a standard interface through which users interact with applications on Web servers. Thus, CGI provides a way for clients (e.g., Web browsers) to interface indirectly with applications on the Web server. Because CGI is an interface, it cannot be programmed directly; a script or executable program (commonly called a **CGI script**) must be executed to interact with it. While CGI scripts can be written in many different programming languages, Perl is commonly used because of its power, its flexibility and the availability of preexisting scripts.

Figure 25.1 illustrates the interaction between client and server when the client requests a document that references a CGI script. Often, CGI scripts process information (e.g., a search-engine query, a credit-card number) gathered from a form. For example, a CGI script might verify credit-card information and notify the client of the results (i.e., accepted or rejected). Permission is granted within the Web server (usually by the **Webmaster** or the author of the Web site) for specific programs on the server to be executed. These programs are typically designated with a certain filename extension (e.g., `.cgi` or `.pl`) or located within a special directory (e.g., `cgi-bin`). After the application output is sent to the server through CGI, the results may be sent to the client. Information received by the client is usually an HTML or XHTML document, but may contain images, streaming audio, Macromedia Flash files (see Chapters 17–18), XML (see Chapter 20), and so on.

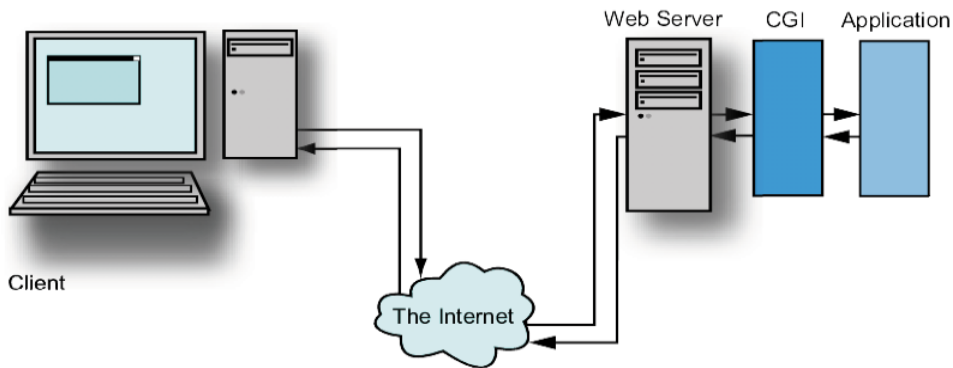


Fig. 25.1 Data path of a typical CGI-based application.

Applications typically interact with the user through **standard input** and **standard output**. Standard input is the stream of information received by a program from a user, typically through the keyboard, but also possibly from a file or another input device. Standard output is the information stream presented to the user by an application; it is typically displayed on the screen, but may be printed or written to a file.

For CGI scripts, the standard output is redirected (or **pip**ed) through the Common Gateway Interface to the server and then sent over the Internet to a Web browser for rendering. If the server-side script is programmed correctly, the output will be readable by the client. Usually, the output is an HTML or XHTML document that is rendered by a Web browser.

25.2 Perl

With the advent of the World Wide Web and Web browsers, the Internet gained tremendous popularity. This greatly increased the volume of requests users made for information from Web servers. It became evident that the degree of interactivity between the user and the server would be crucial. The power of the Web resides not only in serving content to users, but also in responding to requests from users and generating dynamic content. The framework for such communication already existed through CGI. Most of the information users send to servers is text; thus Perl was a logical choice for programming the server side of interactive Web-based applications. Perl possesses simple, yet powerful, text-processing capabilities and is arguably the most popular CGI scripting language. The Perl community, headed by Wall (who currently works for O'Reilly & Associates as a Perl developer and researcher), continuously works to evolve the language, keeping it competitive with newer server-side technologies, such as PHP (see Chapter 26).

To run the Perl scripts in this chapter, Perl must first be installed on your system. *ActivePerl* is the industry-standard Perl distribution for Windows, Solaris and Linux platforms and is available as a free download from www.activestate.com/Products/ActivePerl. Installation instructions can be found at aspn.activestate.com/ASPN/docs/ActivePerl/install.html. Please visit www.perl.com to find information on obtaining and installing Perl on other platforms.

Printing a Line of Text

Figure 25.2 presents a simple Perl program that writes the text "Welcome to Perl!" to the screen. The program does not interact with the Common Gateway Interface and therefore is not a CGI script. Our initial examples are command-line programs that illustrate fundamental Perl programming.

```
1  #!C:\Perl\bin\perl
2  # Fig. 25.2: fig25_02.pl
3  # A first program in Perl.
4
5  print( "Welcome to Perl!\n" );
```

```
Welcome to Perl!
```

Fig. 25.2 Simple Perl program.

Lines 2–3 use the Perl **comment character** (#) to instruct the interpreter to ignore everything on the current line following the #. This syntax allows programmers to write descriptive comments in their programs. The exception to this rule is the “**shebang**” **construct** (!) in line 1. On UNIX systems, this line indicates the path to the Perl interpreter (e.g., #!/usr/bin/perl). On other systems (e.g., Windows), the line may be ignored or may indicate to the server (e.g., Apache) that a Perl program follows the statement.



Good Programming Practice 25.1

While not all servers require the “shebang” construct (!), it is good practice to include it for program portability.




Common Programming Error 25.1

Some systems require that the shebang construct indicate the path to the Perl interpreter. If this path is incorrect, the program might not run. For Windows, this path is normally #!C:\Perl\bin\perl. For Unix, the path is normally #!/usr/bin/perl. If you are unsure where the Perl interpreter is, do a search for perl and use the path found in the shebang construct.

The comment (line 2) indicates that the file name of the program is fig25_02.pl. Perl program file names typically end with the .pl extension. The program can be executed by running the Perl interpreter from the command prompt (e.g., the Command Prompt in Windows 2000/XP or the MS-DOS Prompt in older versions of Windows). Refer to “Running a Perl Script” later in this section to learn more about how to execute this example.

Line 5 calls function **print** to write text to the screen. Note that because Perl is case sensitive, writing Print or PRINT instead of print yields an error. The text "Welcome to Perl!\n" is enclosed in quotes and is called a **string**. The last portion of the string—the newline **escape sequence**, \n—moves the output cursor to the next line. The semicolon (;) at the end of line 5 terminates the Perl statement. Note that the argument passed to function print (i.e., the string that we wish to print) is enclosed in parentheses (). The parentheses are not required; however, we suggest that you use parentheses as often as possible in your programs to maintain clarity. In this example, we use parentheses to indicate what we want printed. We will demonstrate the use of parentheses throughout the chapter.



Common Programming Error 25.2

Forgetting to terminate a statement with a ; is a syntax error in most cases.

Running a Perl Script

To run `fig25_02.pl`, we must first open the Windows command prompt. Click the **Start** button and click **Run....** Type `cmd` in the **Open:** text field. This opens the command-prompt window. At the command prompt, type

```
cd examplesFolder
```

where `examplesFolder` is the full path of the folder containing the Chapter 25 examples (e.g., `C:\IW3HTP3\examples\ch25examples`). This folder path should now appear to the left of the blinking cursor to indicate that any commands we type will be executed in that location. Type at the command prompt


```
perl fig25_02.pl
```

where `perl` is the interpreter and `fig25_02.pl` is the script. Alternatively, you could type

```
perl -w fig25_02.pl
```

which instructs the Perl interpreter to output warnings to the screen if it finds potential bugs in your code.

On Windows systems, a Perl script may also be executed by double clicking its program icon. The program window closes automatically once the script terminates, and any screen output is lost. For this reason, it is usually better to run a script from the command prompt.



Error-Prevention Tip 25.1

When running a Perl script from the command line, always use the `-w` option. Otherwise, the program may seem to execute correctly even though there is actually something wrong with the source code. The `-w` option displays warnings encountered while executing a Perl program.

Perl Variables and Data Types

Perl has built-in data types (Fig. 25.3) that represent different kinds of data. Note that each variable name has a specific character (i.e., `$`, `@` or `%`) preceding it. For example, the `$` character specifies that the variable contains a **scalar** value (i.e., strings, integer numbers, float-

Data type	Format for variable names of this type	Description
Scalar	<code>\$scalarname</code>	Can be a string, an integer number, a floating-point number or a reference.
Array	<code>@arrayname</code>	An ordered list of scalar variables that can be accessed using integer indices.
Hash	<code>%hashname</code>	An unordered set of scalar variables whose values are accessed using unique scalar values (i.e., strings) called keys .

Fig. 25.3 Perl data types.

ing-point numbers and references). The script `fig25_04.pl` (Fig. 25.4) demonstrates the manipulation of scalar variables.



Common Programming Error 25.3

Failure to place a preceding \$ character before a scalar variable name is a syntax error.

```

1  #!C:\Perl\bin\perl
2  # Fig. 25.4: fig25_04.pl
3  # Program to illustrate the use of scalar variables.
4
5  $number = 5;
6  print( "The value of variable \ $number is: $number\n\n" );
7
8  $number += 5;
9  print( "Variable \ $number after adding 5 is: $number\n" );
10
11 $number *= 2;
12 print( "Variable \ $number after multiplying by 2 is: " );
13 print( "$number\n\n\n" );
14
15 # using an uninitialized variable in the context of a string
16 print( "Using a variable before initializing: $variable\n\n" );
17
18 # using an uninitialized variable in a numeric context
19 $test = $undefined + 5;
20 print( "Adding uninitialized variable \ $undefined " );
21 print( "to 5 yields: $test\n" );
22
23 # using strings in numeric contexts
24 $string = "A string value";
25 $number += $string;
26 print( "Adding a string to an integer yields: $number\n" );
27
28 $string2 = "15charactersand1";
29 $number2 = $number + $string2;
30 print( "Adding $number to string \"$string2\" yields: " );
31 print( "$number2\n" );

```

The value of variable \$number is: 5

Variable \$number after adding 5 is: 10

Variable \$number after multiplying by 2 is: 20

Using a variable before initializing:

Adding uninitialized variable \$undefined to 5 yields: 5

Adding a string to an integer yields: 20

Adding 20 to string "15charactersand1" yields: 35

Fig. 25.4 Using scalar variables.

In Perl, a variable is created the first time it is encountered by the interpreter. Line 5 creates a variable with name `$number` and sets its value to 5. Line 6 calls function `print` to write text followed by the value of `$number`. Note that the actual value of `$number` is printed, rather than the string `"$number"`; when a variable is encountered inside a double-quoted (") string, Perl uses a process called **interpolation** to replace the variable with its associated data. Note that we avoid interpolation and actually print the string `"$number"` in the first part of line 6 by immediately preceding the dollar sign with a backslash character (`\`). The string `"\ $number"` is interpreted as the dollar-sign character followed by the string `number`, rather than as the variable `$number`. See the last paragraph of this section for more on printing special characters. Line 8 adds 5 to `$number`, which results in the value 10 being stored in `$number`. Note that we use an **assignment operator** (`+=`) to yield an expression equivalent to `$number = $number + 5`, which adds 5 to the value of `$number` and stores the result in `$number`. Assignment operators (i.e., `+=`, `-=`, `*=` and `/=`) are syntactical shortcuts. In line 11, we use the multiplication assignment operator, `*=`, to multiply `$number` by 2 and store the result in `$number`.



Error-Prevention Tip 25.2

Function `print` can be used to display the value of a variable at a particular point during a program's execution. This is often helpful in debugging a program.

In Perl, uninitialized variables have the value **undef**, which evaluates to different values depending on the variable's context. When `undef` is used in a numeric context (e.g., `$undefined` in line 19), it evaluates to 0. In contrast, when it is interpreted in a string context (e.g., `$variable` in line 16), `undef` evaluates to the empty string (").

Lines 24–31 show the results of evaluating strings in a numeric context. Unless a string begins with a digit, it is evaluated as `undef` in a numeric context. If it begins with a digit, every character up to, but not including, the first nondigit character is evaluated as a number, and the remaining characters are ignored. For example, the string "A string value" (line 24) does not begin with a digit and therefore evaluates to `undef`. Because `undef` evaluates to 0, variable `$number`'s value is unchanged. The string "15charactersand1" (line 28) begins with a digit and is interpreted as 15. The character 1 on the end is ignored, because there are nondigit characters preceding it. Evaluating a string in numeric context does not actually change the value of the string. This rule is shown by line 31's output, which prints the original string as "15charactersand1" and the result of adding this string to the integer value 20 as 35.

The programmer does not need to differentiate between numeric and string data types, because the interpreter's evaluation of scalar variables depends on the context in which they are used.



Common Programming Error 25.4

Using an uninitialized variable might make a numerical calculation incorrect. For example, multiplying a number by an uninitialized variable results in 0.



Error-Prevention Tip 25.3

While it is not always necessary to initialize variables before using them, errors can be avoided by doing so. Many Perl programmers place the statements `use strict` and `use warnings` at the top of their programs to catch such errors. For simplicity, we will not initialize variables before using them in this chapter.

Creating and Manipulating Arrays

Perl provides the capability to store data in arrays. Arrays are divided into **elements**, each containing a scalar value. The script `fig25_05.pl` (Fig. 25.5) demonstrates some techniques for array initialization and manipulation.

```

1  #!C:\Perl\bin\perl
2  # Fig. 25.5: fig25_05.pl
3  # Program to demonstrate arrays in Perl.
4
5  @array = ( "Bill", "Bobby", "Sue", "Michelle" );
6
7  print( "The array contains: @array\n" );
8  print( "Printing array outside of quotes: ", @array, "\n\n" );
9
10 print( "Third element: $array[ 2 ]\n" );
11
12 $number = 3;
13 print( "Fourth element: $array[ $number ]\n\n" );
14
15 @array2 = ( 'A' .. 'Z' );
16 print( "The range operator is used to create a list of\n" );
17 print( "all capital letters from A to Z:\n" );
18 print( "@array2 \n\n" );
19
20 $array3[ 3 ] = "4th";
21 print( "Array with just one element initialized: @array3 \n\n" );
22
23 print( 'Printing literal using single quotes: ' );
24 print( '@array and \n', "\n" );
25
26 print( "Printing literal using backslashes: " );
27 print( "\\@array and \\n\n" );

```

```

The array contains: Bill Bobby Sue Michelle
Printing array outside of quotes: BillBobbySueMichelle

Third element: Sue
Fourth element: Michelle

The range operator is used to create a list of
all capital letters from A to Z:
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Array with just one element initialized:    4th

Printing literal using single quotes: @array and \n
Printing literal using backslashes: @array and \n

```

Fig. 25.5 Using arrays.

Line 5 initializes array `@array` to contain the strings "Bill", "Bobby", "Sue" and "Michelle". In Perl, all array variable names must be preceded by the `@` symbol. Parentheses are necessary to group the strings in the array assignment; this group of elements sur-

rounded by parentheses is called a **list**. In assigning the list to `@array`, each person's name is stored in an individual array element with a unique integer index value, starting at 0. Like JavaScript arrays, arrays in Perl use a zero-based numbering scheme.



Common Programming Error 25.5

Although lists in Perl may seem similar to arrays, they are not. A list is simply a group of elements surrounded by parentheses and does not contain the entire functionality of an array. You can access individual elements in a list using square brackets (i.e., subscript notation), but you cannot assign a new value to an element; lists are immutable.

When printing an array inside double quotes (line 7), the array element values are printed with only one space separating them. The values are separated by whatever is in special variable `$"`, which, by default, is a space. If this value were changed to the letter "a", all the array elements would be printed with the character "a" between them. If the array name is not enclosed in double quotes when it is printed (line 8), the interpreter prints the element values without inserting spaces between them.

Line 10 demonstrates how individual array elements are accessed using square brackets (`[]`). As mentioned previously, if we use the `@` character followed by the array name, we reference the array as a whole. But if the name of the array is prefixed by the `$` character and followed by an index number in square brackets (as in line 10), it refers instead to an individual element of the array that is a scalar value. Line 13 demonstrates the use of a variable as the index number. The value of `$array[$number]` is used to get the value of the fourth element of the array.

Line 15 initializes array `@array2` to contain the capital letters A to Z inclusive. The **range operator** (`..`) specifies that all values between uppercase A and uppercase Z are to be placed in the array. The range operator can be used to create a consecutive series of values, such as 1 through 15 or a through z.

The Perl interpreter handles memory management. Therefore, it is not necessary to specify an array's size. If a value is assigned to a position outside the range of the array or to an uninitialized array, the interpreter automatically extends the array range to include the new element. Elements that are added by the interpreter during an adjustment of the range are initialized to the `undef` value. Line 20 assigns a value to the fourth element in the uninitialized array `@array3`. The interpreter recognizes that memory has not been allocated for this array and creates new memory for the array. The interpreter then sets the value of the first three elements to `undef` and the value of the fourth element to the string "4th". When the array is printed, the first three `undef` values are treated as empty strings and printed with a space between each. This accounts for the three extra spaces in the output before the string "4th".

To print special characters like `\`, `@`, `$` and `"` and not have the interpreter treat them as an escape sequence, array or variable, Perl provides two options. The first is to **print** (lines 23–24) the characters as a **literal string** (i.e., a string enclosed in single quotes). When strings are inside single quotes, the interpreter treats the string literally and does not attempt to interpret any escape sequence or variable substitution. The second choice is to use the backslash character (line 26–27) to **escape** special characters.

25.3 String Processing and Regular Expressions

One of Perl's most powerful capabilities is the processing of textual data easily and efficiently, which allows for straightforward searching, substitution, extraction and concatena-

tion of strings. Text manipulation in Perl is usually done with a **regular expression**—a series of characters that serves as a pattern-matching template (or search criterion) in strings, text files and databases. This feature allows complicated searching and string processing to be performed using relatively simple expressions.

Comparing Strings

Many string-processing tasks can be accomplished by using Perl's **equality** and **comparison** operators (Fig. 25.6, `fig25_06.pl`). Line 5 declares and initializes array `@fruits`. Operator `qw` ("quote word") takes the content inside the parentheses and creates a comma-separated list, with each element wrapped in double quotes. In this example, `qw(apple orange banana)` is equivalent to `("apple", "orange", "banana")`.

```
1  #!C:\Perl\bin\perl
2  # Fig. 25.6: fig25_06.pl
3  # Program to demonstrate the eq, ne, lt, gt operators.
4
5  @fruits = qw( apple orange banana );
6
7  foreach $item ( @fruits ) {
8
9      if ( $item eq "banana" ) {
10         print( "String '$item' matches string 'banana'\n" );
11     }
12
13     if ( $item ne "banana" ) {
14         print( "String '$item' does not match string 'banana'\n" );
15     }
16
17     if ( $item lt "banana" ) {
18         print( "String '$item' is less than string 'banana'\n" );
19     }
20
21     if ( $item gt "banana" ) {
22         print( "String '$item' is greater than string 'banana'\n" );
23     }
24 }
```

```
String 'apple' does not match string 'banana'
String 'apple' is less than string 'banana'
String 'orange' does not match string 'banana'
String 'orange' is greater than string 'banana'
String 'banana' matches string 'banana'
```

Fig. 25.6 Using the `eq`, `ne`, `lt` and `gt` operators.

Lines 7–24 demonstrate our first example of Perl **control statements**. The **foreach** statement (line 7) iterates sequentially through the elements in `@fruits`. Each element's value is assigned to variable `$item`, and the body of the `foreach` executes once for each element in the array. Note that a semicolon does not terminate the `foreach`.

Line 9 introduces the **if** statement. Parentheses surround the condition being tested, and mandatory curly braces surround the block of code that is executed when the condition

is true. In Perl, any scalar except the number 0, the string "0" and the empty string (i.e., undef values) is defined as true. In our example, when the `$item`'s content is tested against "banana" (line 9) for equality, the condition evaluates to true, and the **print** function (line 10) executes.

The remaining **if** statements (lines 13, 17 and 21) demonstrate the other string comparison operators. Operators **ne**, **lt** and **gt** test strings for inequality, less-than and greater-than, respectively. These operators are used only with strings. When comparing numeric values, operators **==**, **!=**, **<**, **<=**, **>** and **>=** are used.



Common Programming Error 25.6

Using **==** for string comparisons or **eq** for numerical comparisons can result in errors in the program.



Common Programming Error 25.7

While the number 0 and the string "0" evaluate to false in Perl **if** statements, other string values that might look like zero ("0.0") evaluate to true.

Regular Expressions

For more powerful string comparisons, Perl provides the **match operator** (**m/pattern/** or **/pattern/**), which uses regular expressions to search a string for a specified *pattern*. Figure 25.7 uses the match operator to perform a variety of regular-expression tests.

```

1  #!C:\Perl\bin\perl
2  # Fig 25.7: fig25_07.pl
3  # Searches using the matching operator and regular expressions.
4
5  $search = "Now is is the time";
6  print( "Test string is: '$search'\n\n" );
7
8  if ( $search =~ /Now/ ) {
9      print( "String 'Now' was found.\n" );
10 }
11
12 if ( $search =~ /^Now/ ) {
13     print( "String 'Now' was found at the beginning of the line." );
14     print( "\n" );
15 }
16
17 if ( $search =~ /Now$/ ) {
18     print( "String 'Now' was found at the end of the line.\n" );
19 }
20
21 if ( $search =~ /\b ( \w+ ow ) \b/x ) {
22     print( "Word found ending in 'ow': $1 \n" );
23 }
24
25 if ( $search =~ /\b ( \w+ ) \s ( \1 ) \b/x ) {
26     print( "Repeated words found: $1 $2\n" );
27 }
28

```

Fig. 25.7 Using the matching operator. (Part 1 of 2.)

```

29 @matches = ( $search =~ / \b ( t \w+ ) \b /gx );
30 print( "Words beginning with 't' found: @matches\n" );

```

Test string is: 'Now is is the time'

String 'Now' was found.

String 'Now' was found at the beginning of the line.

Word found ending in 'ow': Now

Repeated words found: is is

Words beginning with 't' found: the time

Fig. 25.7 Using the matching operator. (Part 2 of 2.)

We begin by assigning the string "Now is is the time" to variable `$search` (line 5). The expression in line 8 uses the match operator to search for the **literal characters** `Now` inside variable `$search`. The `m` character preceding the slashes of the `m//` operator is optional in most cases and thus is omitted here.

The match operator takes two operands. The first operand is the regular-expression pattern to search for (`Now`), which is placed between the slashes of the `m//` operator. The second operand is the string within which to search, which is assigned to the match operator using the `=~` operator. The `=~` operator is sometimes called a **binding operator** because it binds whatever is on its left side to a regular-expression operator on its right.

In our example, the pattern `Now` is found in the string "Now is is the time". The match operator returns true, and the body of the `if` statement is executed. In addition to literal characters like `Now`, which match only themselves, regular expressions can include special characters called **metacharacters**, which specify patterns or contexts that cannot be defined using literal characters. For example, the **caret metacharacter** (`^`) matches the beginning of a string. The next regular expression (line 12) searches the beginning of `$search` for the pattern `Now`.

The `$` metacharacter searches the end of a string for a pattern (line 17). The pattern `Now` is not found at the end of `$search`, so the body of the `if` statement (line 18) is not executed. Note that `Now$` is not a variable; it is a search pattern that uses `$` to search for `Now` at the end of a string.

The condition in line 21 searches (from left to right) for the first word ending with the letters `ow`. As in strings, backslashes in regular expressions escape characters with special significance. For example, the `\b` expression does not match the literal characters `"\b."` Instead, the expression matches any **word boundary**. A word boundary is a boundary between an **alphanumeric character** (0–9, a–z, A–Z and the underscore character) and something that is not an alphanumeric character. The expression inside the parentheses, `\w+ ow`, indicates that we are searching for patterns ending in `ow`. The first part, `\w+`, is a combination of `\w` (an escape sequence that matches a single alphanumeric character) and the `+` **modifier**, which is a **quantifier** that instructs Perl to match the preceding character one or more times. Thus, `\w+` matches one or more alphanumeric characters. The characters `ow` are taken literally. Collectively, the expression `/\b (\w+ ow) \b/` matches one or more alphanumeric characters ending with `ow`, with word boundaries at the beginning and end. See Fig. 25.8 for a description of several Perl regular-expression quantifiers and Fig. 25.9 for a list of regular-expression metacharacters.

Quantifier	Matches
{n}	Exactly n times
{m,n}	Between m and n times inclusive
{n,}	n or more times
+	One or more times (same as {1,})
*	Zero or more times (same as {0,})
?	One or zero times (same as {0,1})

Fig. 25.8 Some of Perl’s quantifiers.

Symbol	Matches	Symbol	Matches
^	Beginning of line	\d	Digit (i.e., 0 to 9)
\$	End of line	\D	Nondigit
\b	Word boundary	\s	Whitespace
\B	Nonword boundary	\S	Nonwhitespace
\w	Word (alphanumeric) character	\n	Newline
\W	Nonword character	\t	Tab

Fig. 25.9 Some of Perl’s metacharacters.

The parentheses that surround `\w+ ow` in line 21 (Fig. 25.7) indicate that the text matching the pattern is to be saved in a special Perl variable. The parentheses (line 21 of Fig. 25.7) cause `Now` to be stored in variable `$1`. Multiple sets of parentheses may be used in regular expressions, where each match results in a new Perl variable (`$1`, `$2`, `$3`, etc.). The value matched in the first set of parentheses is stored in variable `$1`, the value matched in the second set of parentheses is stored in variable `$2`, and so on.

Adding **modifying characters** after a regular expression refines the pattern-matching process. Modifying characters (Fig. 25.10) placed to the right of the forward slash that delimits the regular expression instruct the interpreter how to treat the preceding expression. For example, the `i` after the regular expression

```
/computer/i
```

tells the interpreter to ignore case when searching, thus matching `computer`, `COMPUTER`, `Computer`, `CoMpuTER`, etc.

When added to the end of a regular expression, the `x` modifying character indicates that whitespace characters in the regular expression are to be ignored. This allows programmers to add space characters to their regular expressions for readability without affecting the search. If the expression were written as

```
$search =~ /\b ( \w+ ow ) \b/
```

Modifying character	Purpose
<code>g</code>	Performs a global search; finds and returns all matches, not just the first one found.
<code>i</code>	Ignores the case of the search string (case insensitive).
<code>m</code>	The string is evaluated as if it had multiple lines of text (i.e., newline characters are not ignored).
<code>s</code>	Ignores the newline character and treats it as whitespace. The text is seen as a single line.
<code>x</code>	All whitespace characters in the regular expression are ignored when searching the string.

Fig. 25.10 Some of Perl's modifying characters.

without the `x` modifying character, then the script would be searching for a word boundary, two spaces, one or more alphanumeric characters, one space, the characters `ow`, two spaces and a word boundary. The expression would not match `$search`'s value.

The condition in line 25 uses the **memory function** (i.e., parentheses) in a regular expression. The first parenthetical expression matches any string containing one or more alphanumeric characters. The expression `\1` then evaluates to the word that was matched in the first parenthetical expression. The regular expression searches for two identical consecutive words separated by a whitespace character (`\s`), in this case `"is is."`

Line 29 searches for words beginning with the letter `t` in the string `$search`. Modifying character `g` indicates a global search—a search that does not stop after the first match is found. The array `@matches` is assigned the value of a list of all matching words.

25.4 Viewing Client/Server Environment Variables

Knowing information about a client's execution environment allows system administrators to provide client-specific information. **Environment variables** contain information about the execution environment in which a script is being run, such as the type of Web browser used, the HTTP host and the HTTP connection. A Web server might use this information to generate client-specific Web pages.

Until now, we have written simple Perl applications that output to the local user's screen. Through CGI, we can communicate with the Web server and its clients, allowing us to use the Internet as a method of input and output for our Perl applications. In order to run Perl scripts as CGI applications, a Web server must be installed and configured correctly for your system. See Chapter 21, Web Servers (IIS and Apache), and www.deitel.com for information on installing and setting up a Web server.

We place our CGI programs in the `cgi-bin` folder. If this directory does not exist, create it in the Web server's root directory. Recall that, by default, the Web server's root directory is `C:\Inetpub\wwwroot` under IIS and `C:\Program Files\Apache Group\Apache2` under Apache. The Web server must be configured correctly to allow files in the `cgi-bin` directory to run as scripts. The Web server also must be running for CGI programs to execute. Please consult your Web server's documentation for additional

configuration details. Copy the remaining .pl files and the password.txt file from the Chapter 25 examples directory to the cgi-bin directory. Other important files (.html files, .shtml files, etc.) normally are placed in the Web server's root directory. All image (.gif) files should be placed in an images folder within the Web server's root directory for the examples in this chapter to display properly.

CGI Script that Displays Environment Variables

In Fig. 25.11, we present our first CGI program. When creating dynamic Web pages in Perl, we output XHTML by using `print` statements. The XHTML generated in this program displays the client's environment variables. The **use directive** (line 5) instructs Perl programs to include the contents (e.g., functions) of predefined packages called **modules**. The **CGI module**, for example, contains useful functions for CGI scripting in Perl, including functions that return strings representing XHTML (or HTML) tags and HTTP headers. With the `use` directive, we can specify which functions we would like to import from a particular module. In line 5, we use the **import tag :standard** to import a predefined set of standard functions. We use several of these functions in the following examples.

```

1  #!C:\Perl\bin\perl
2  # Fig. 25.11: fig25_11.pl
3  # Program to display CGI environment variables.
4
5  use CGI qw( :standard );
6
7  $dtd =
8  "-//W3C//DTD XHTML 1.0 Transitional//EN"
9  "\"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd";
10
11 print( header() );
12
13 print( start_html( { dtd => $dtd,
14                    title => "Environment Variables..." } ) );
15
16 print( "<table style = \"border: 0; padding: 2;
17        font-weight: bold\">" );
18
19 print( Tr( th( "Variable Name" ),
20           th( "Value" ) ) );
21
22 print( Tr( td( hr() ), td( hr() ) ) );
23
24 foreach $variable ( sort( keys( %ENV ) ) ) {
25
26     print( Tr( td( { style => "background-color: #11bbff" },
27                  $variable ),
28              td( { style => "font-size: 12pt" },
29                  $ENV{ $variable } ) ) );
30
31     print( Tr( td( hr() ), td( hr() ) ) );
32 }
33
```

Fig. 25.11 Displaying CGI environment variables. (Part 1 of 2.)


```

34 print( "</table>" );
35 print( end_html() );

```

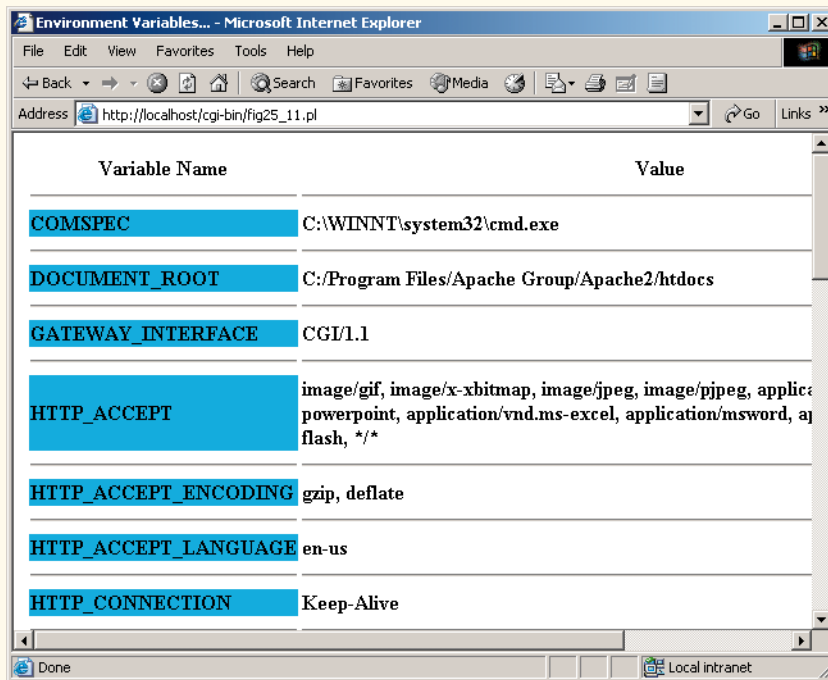


Fig. 25.11 Displaying CGI environment variables. (Part 2 of 2.)

Line 11 instructs the Perl script to print a valid **HTTP header**, using function **header** from the CGI library. Browsers use HTTP headers to determine how to handle incoming data. The **header** function returns the string "Content-type: text/html\n\n", indicating to the client that what follows is XHTML. The **text/html** portion of the header indicates that the browser must display the returned information as an XHTML document. Standard output is redirected when a CGI script executes, so the function **print** outputs to the user's Web browser.

In lines 13–14, we begin to write XHTML to the client by using the **start_html** function. This function prints the document type definition for this document, as well as several opening XHTML tags (<html>, <head>, <title>, etc., up to the opening <body> tag). Note that certain information is specified within curly braces ({}). In many CGI module functions, additional information (e.g., attributes) can be specified within curly braces. The **print** function in lines 13–14 displays the result returned by **start_html**. Each argument within the curly braces is in the form of a *key–value* pair. A **key** (or **value name**) is assigned a value using the **arrow operator** (**=>**), where the key is to the left of the arrow and the value is to the right. The first argument consists of the key **dtd** and the value **\$dtd**. When we include the **dtd** argument in the function **start_html**, the default document type definition is changed from HTML's DTD to the value of **\$dtd**. This adds the proper XHTML DTD (specified in lines 7–9) to this file. The **title** argument specifies the value that goes between the opening and closing <title> tags. In this example, the title of the Web page

is set to "Environment Variables...". The order of these key–value pairs is not important.

The function `start_html`, as well as many other Perl functions, can be used in a variety of ways. All of the arguments to `start_html` are optional, and some arguments can be specified differently than how we see in this program. A good way to find correct syntax is to consult the book *Official Guide to Programming with CGI.pm: The Standard for Building Web Scripts* by Lincoln Stein (the creator of the CGI library). Information about CGI is also available on the Internet. (See Section 25.11, Wide Web Resources, at the end of this chapter.)

In lines 19–20, we have two more CGI.pm functions—**Tr** and **th**. These functions place their arguments between table row tags and table header tags, respectively. The `print` statement outputs

```
<tr><th>Variable Name</th><th>Value</th></tr>
```

Function **th** is called twice, with the arguments "Variable Name" and "Value", causing both of these values to be surrounded by start and end table header tags. Function **Tr** places these two header tags inside `<tr>` start and end tags. [Note: This function has a capital "T" because Perl already contains an operator `tr`.] We call function **Tr** again in line 22 with the **hr** and **td** functions, in order to print a row of horizontal rules within `<td>` tags.

The **%ENV hash** is a built-in data structure in Perl that contains the names and values of all the environment variables. The `foreach` statement in lines 24–32 uses the **%ENV** hash. The **hash** data type is designated by the `%` character and represents an unordered set of scalar-value pairs. Unlike an array, which accesses elements through integer indices (e.g., `$array[2]`), each element in a hash is accessed using a unique string **key** that is associated with the element's value. For this reason, hashes are also known as **associative arrays**, because the keys and values are associated in pairs. Hash values are accessed using the syntax `$hashName{ keyName }`. In this example, each key in hash **%ENV** is the name of an environment variable name (e.g., `HTTP_HOST`). When this value is used as the key in the **%ENV** hash, the variable's value is returned.

Function **keys** returns an unordered array containing all the keys in the **%ENV** hash (line 24), as hash elements have no defined order. We call function **sort** to order the array of keys alphabetically. Finally, the `foreach` iterates sequentially through the array returned by `sort`, repeatedly assigning the current key's value to scalar `$variable`. Lines 26–31 execute for each element in the array of key values. In lines 26–29, we output a new row for the table, containing the name of the environment variable (`$variable`) in one column and the value for the variable (`$ENV{ $variable }`) in the next. We call function **td** in line 26 and use curly-brace notation to specify the value for the attribute `style`. In line 28, we use the hash notation again to specify a `style` attribute. Line 35 calls the function **end_html**, which returns the closing tags for the page (`</body>` and `</html>`).

25.5 Form Processing and Business Logic

XHTML forms enable Web pages to collect data from users and send it to a Web server for processing by server-side programs and scripts. This allows users to purchase products, send and receive Web-based e-mail, participate in polls and perform many other tasks. This type of Web communication allows users to interact with the server and is vital to Web development.

Figure 25.12 uses an XHTML form to collect information about users before adding them to a mailing list. This type of registration form could be used by a software company to obtain profile information before allowing the user to download software.

```

1  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4  <!-- Fig. 25.12: fig25_12.html -->
5
6  <html>
7    <head>
8      <title>Sample form to take user input in XHTML</title>
9    </head>
10
11    <body style = "font-face: arial; font-size: 12pt">
12
13      <div style = "font-size: 14pt; font-weight: bold">
14        This is a sample registration form.
15      </div>
16
17      <br />
18      Please fill in all fields and click Register.
19
20      <form method = "post" action = "/cgi-bin/fig25_13.pl">
21
22        <img src = "images/user.gif" /><br />
23
24        <div style = "color: blue" >
25          Please fill out the fields below.<br />
26        </div>
27
28        <img src = "images/fname.gif" />
29        <input type = "text" name = "fname" /><br />
30        <img src = "images/lname.gif" />
31        <input type = "text" name = "lname" /><br />
32        <img src = "images/email.gif" />
33        <input type = "text" name = "email" /><br />
34        <img src = "images/phone.gif" />
35        <input type = "text" name = "phone" /><br />
36
37        <div style = "font-size: 10pt">
38          Must be in the form (555)555-5555.<br /><br />
39        </div>
40
41        <img src = "images/downloads.gif" /><br />
42        <div style = "color: blue">
43          Which book would you like information about?<br />
44        </div>
45
46        <select name = "book">
47          <option>Internet and WWW How to Program 3e</option>
48          <option>C++ How to Program 4e</option>

```

Fig. 25.12 XHTML document with an interactive form. (Part 1 of 2.)

```

49         <option>Java How to Program 5e</option>
50         <option>XML How to Program 1e</option>
51     </select><br /><br />
52
53     <img src = "images/os.gif" /><br />
54     <div style = "color: blue">
55         Which operating system are you currently using?
56     </div><br />
57
58     <input type = "radio" name = "os"
59         value = "Windows XP" checked />
60     Windows XP<input type = "radio"
61         name = "os" value = "Windows 2000" />
62     Windows 2000<input type = "radio"
63         name = "os" value = "Windows 98/me" />
64     Windows 98/me<br /><input type = "radio"
65         name = "os" value = "Linux" />
66     Linux<input type = "radio" name = "os"
67         value = "Other" />
68     Other<br /><input type = "submit"
69         value = "Register" />
70 </form>
71 </body>
72 </html>

```

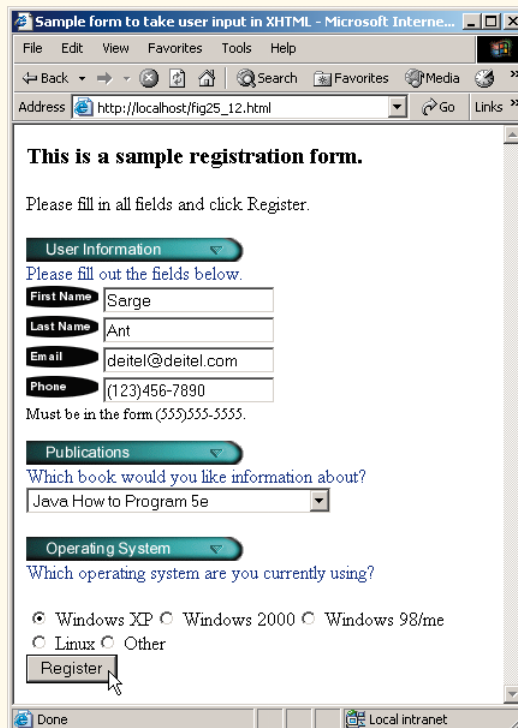


Fig. 25.12 XHTML document with an interactive form. (Part 2 of 2.)

Line 21 contains a form element which indicates that, when the user clicks **Register**, the form information is posted to the server. The attribute `action = "/cgi-bin/fig25_13.pl"` directs the server to execute the `fig25_13.pl` Perl script (located in the `cgi-bin` directory) to process the posted form data. We assign a unique name (e.g., `email` in line 33) to each of the form's input fields. When **Register** is clicked, each field's name and value are sent to the script `fig25_13.pl`, which can then access the submitted value for each specific field.



Good Programming Practice 25.2

Use meaningful XHTML object names for input fields. This practice makes Perl programs easier to understand when processing form data.

The program in Fig. 25.13 processes the data posted by `fig25_12.html` and sends a Web-page response back to the client. Function **param** (lines 8–13) is part of the Perl CGI module and retrieves values from a form field's value. For example, in line 35 of Fig. 25.12, an XHTML form text field is created with the name `phone`. In line 12 of Fig. 25.13, we access the value that the user entered for that field by calling `param("phone")` and assign the value returned to variable `$phone`.

Line 24 determines whether the phone number entered by the user is valid. In this case, (555)555-5555 is the only acceptable format. Validating information is crucial when you are maintaining a database, and a great way to do this is by using regular expressions. Validation ensures, for example, that data is stored in the proper format in a database and that credit-card numbers contain the proper number of digits before they are encrypted for submission to a merchant. This script implements the **business logic**, or **business rules**, of our application.

```

1  #!C:\Perl\bin\perl
2  # Fig. 25.13: fig25_13.pl
3  # Program to read information sent to the server
4  # from the form in the fig25_12.html document.
5
6  use CGI qw( :standard );
7
8  $os = param( "os" );
9  $firstName = param( "fname" );
10 $lastName = param( "lname" );
11 $email = param( "email" );
12 $phone = param( "phone" );
13 $book = param( "book" );
14
15 $dtd =
16 "-//W3C//DTD XHTML 1.0 Transitional//EN\"
17   \"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd";
18
19 print( header() );
20
21 print( start_html( { dtd => $dtd,
22                     title => "Form Results" } ) );
23
24 if ( $phone =~ / ^ \ ( \d{3} \) \d{3} - \d{4} $ /x ) {
25     print( "Hi " );

```

Fig. 25.13 Script to process user data from `fig25_12.html`. (Part 1 of 3.)

```

26     print( span( { style => "color: blue; font-weight: bold" },
27                  $firstName ) );
28     print( "!" );
29
30     print( "\nThank you for completing the survey." );
31     print( br(), "You have been added to the " );
32
33     print( span( { style => "color: blue; font-weight: bold" },
34                  $book ) );
35     print( " mailing list.", br(), br() );
36
37     print( span( { style => "font-weight: bold" },
38                  "The following information has
39                  been saved in our database: " ), br() );
40
41     print( table(
42         Tr( th( { style => "background-color: #ee82ee" },
43                 "Name" ),
44             th( { style => "background-color: #9370db" },
45                 "E-mail" ),
46             th( { style => "background-color: #4169e1" },
47                 "Phone" ),
48             th( { style => "background-color: #40e0d0" },
49                 "OS" ) ),
50
51         Tr( { style => "background-color: #c0c0c0" },
52             td( "$firstName $lastName" ),
53             td( $email ),
54             td( $phone ),
55             td( $os ) ) ) );
56
57     print( br() );
58
59     print( div( { style => "font-size: x-small" },
60               "This is only a sample form. You have not been
61               added to a mailing list." ) );
62 }
63 else {
64     print( div( { style => "color: red; font-size: x-large" },
65               "INVALID PHONE NUMBER" ), br() );
66
67     print( "A valid phone number must be in the form " );
68     print( span( { style => "font-weight: bold" },
69               "(555)555-5555." ) );
70
71     print( div( { style => "color: blue" },
72               "Click the Back button, and enter a
73               valid phone number and resubmit." ) );
74     print( br(), br() );
75     print( "Thank you." );
76 }
77
78 print( end_html() );

```

Fig. 25.13 Script to process user data from fig25_12.html. (Part 2 of 3.)

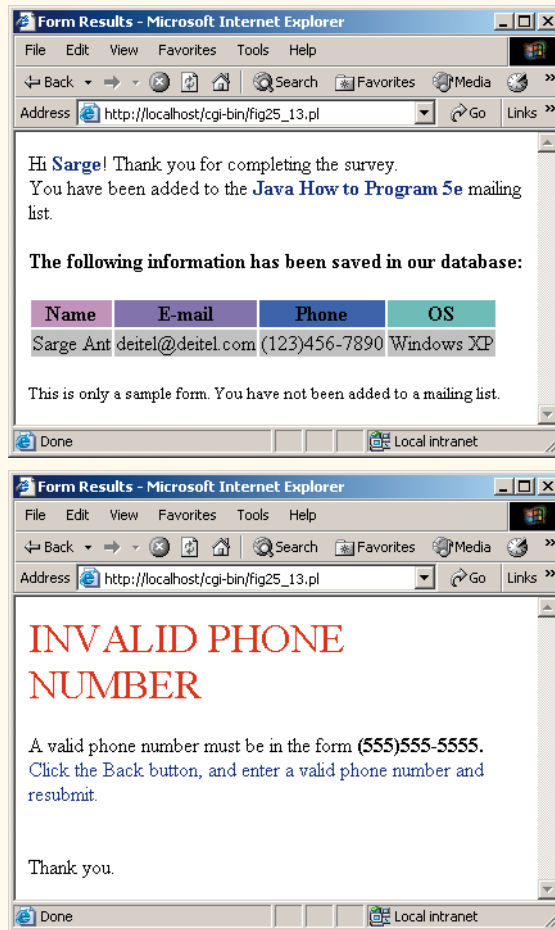


Fig. 25.13 Script to process user data from `fig25_12.html`. (Part 3 of 3.)

The `if` condition in line 24 uses a regular expression to validate the phone number. The expression `"\("` matches the opening parenthesis of the phone number. Because we want to match the literal character `(`, we must escape its normal meaning by using the `\` character. This sequence must be followed by three digits (`\d{3}`), a closing parenthesis, three digits, a hyphen and finally, four more digits. Note that we use the `^` and `$` metacharacters to ensure that there are no extra characters at the beginning or end of the string.

If the regular expression is matched, then the phone number is valid, and a Web page is sent, thanking the user for completing the form. If the user posts an invalid phone number, the `else` (lines 63–76) executes, instructing the user to enter a valid phone number.



Good Programming Practice 25.3

Use business logic to ensure that invalid information is not stored in a database.

The **br** function (line 31) adds a break (`
`) to the XHTML page, while functions **span** (line 26) and **div** (line 59) add `span` and `div` elements to the page, respectively.

25.6 Server-Side Includes

Dynamic content greatly improves the look-and-feel of a Web page. Pages that include the current date or time, rotating banners or advertisements, a daily message, special offers or company news are attractive because they are current. Clients see new information on every visit and thus will likely revisit the site in the future.

Server-Side Includes (SSIs) are commands embedded in XHTML documents to create simple dynamic content. SSI commands like **ECHO** and **INCLUDE** enable the inclusion on Web pages of content that is constantly changing (i.e., the current time) or information that is stored in a database. The command **EXEC** can be used to run CGI scripts and embed their output directly into a Web page.

Not all Web servers support the available SSI commands. Therefore, SSI commands are written as XHTML comments (e.g., `<!--#ECHO VAR="DOCUMENT_NAME" -->`). Servers that do not recognize these commands treat them as comments. Some servers do support SSI commands, but only if they are configured to do so. Check your server's documentation for instructions on how to configure your server to process SSI commands properly. Please refer to

httpd.apache.org/docs/howto/ssi.html

if you are running the Apache Web server, or

perl.xotechnologies.net/tutorials/SSI/SSI.htm

if you are running a version of IIS.

A document containing SSI commands is typically given the **.shtml** file extension (the **s** at the front of the extension stands for **server**). The **.shtml** files are parsed by the server. The server executes the SSI commands and writes output to the client.



Performance Tip 25.1

Parsing XHTML documents on a server can dramatically increase the load on the server. To increase the performance of a heavily loaded server, limit the use of server-side includes.

Figure 25.14 implements a **Web page hit counter**. Each time a client requests the document, the counter is incremented by 1. The Perl script `fig25_15.pl` manipulates the counter.

```

1  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
2    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
3
4  <!-- Fig. 25.14: fig25_14.shtml -->
5
6  <html>
7    <head>
```

Fig. 25.14 Incorporating a Web-page hit counter and displaying environment variables, using server-side includes. (Part 1 of 3.)

```

8      <title>Using Server-Side Includes</title>
9  </head>
10
11  <body>
12      <h3 style = "text-align: center">
13          Using Server-Side Includes
14      </h3>
15
16      <!--#EXEC CGI="/cgi-bin/fig25_15.pl" --><br />
17
18      The Greenwich Mean Time is
19      <span style = "color: blue">
20          <!--#ECHO VAR="DATE_GMT" -->.
21      </span><br />
22
23      The name of this document is
24      <span style = "color: blue">
25          <!--#ECHO VAR="DOCUMENT_NAME" -->.
26      </span><br />
27
28      The local date is
29      <span style = "color: blue">
30          <!--#ECHO VAR="DATE_LOCAL" -->.
31      </span><br />
32
33      This document was last modified on
34      <span style = "color: blue">
35          <!--#ECHO VAR="LAST_MODIFIED" -->.
36      </span><br />
37
38      Your current IP Address is
39      <span style = "color: blue">
40          <!--#ECHO VAR="REMOTE_ADDR" -->.
41      </span><br />
42
43      My server name is
44      <span style = "color: blue">
45          <!--#ECHO VAR="SERVER_NAME" -->.
46      </span><br />
47
48      And I am using the
49      <span style = "color: blue">
50          <!--#ECHO VAR="SERVER_SOFTWARE" -->
51          Web Server.
52      </span><br />
53
54      You are using
55      <span style = "color: blue">
56          <!--#ECHO VAR="HTTP_USER_AGENT" -->.
57      </span><br />
58
59      This server is using

```

Fig. 25.14 Incorporating a Web-page hit counter and displaying environment variables, using server-side includes. (Part 2 of 3.)

```

60     <span style = "color: blue">
61         <!--#ECHO VAR="GATEWAY_INTERFACE" -->.
62     </span><br />
63
64     <br /><br />
65     <div style = "text-align: center;
66                 font-size: xx-small">
67         <hr />
68         This document was last modified on
69         <!--#ECHO VAR="LAST_MODIFIED" -->.
70     </div>
71 </body>
72 </html>

```

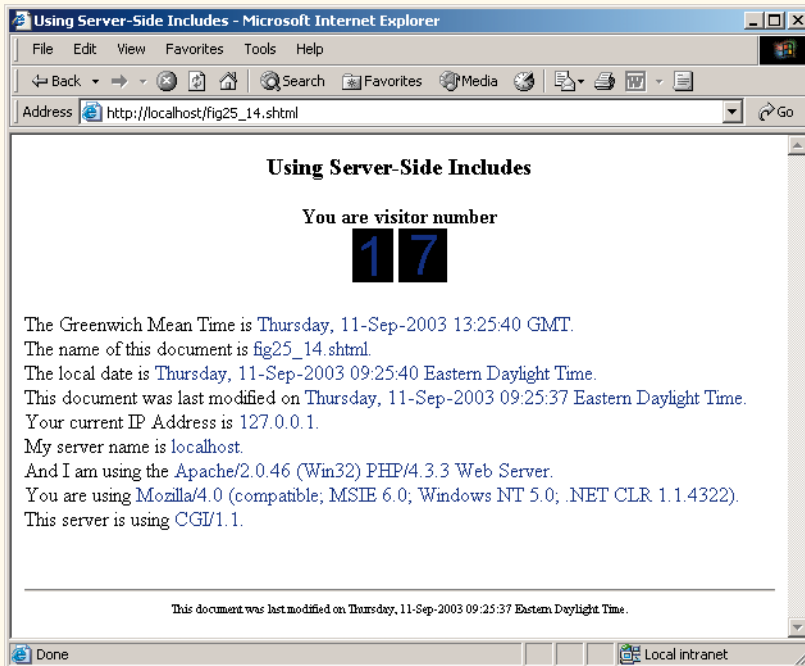


Fig. 25.14 Incorporating a Web-page hit counter and displaying environment variables, using server-side includes. (Part 3 of 3.)

Line 16 of the `fig25_14.shtml` script executes the `fig25_15.pl` script, using the `EXEC` command. Before the XHTML document is sent to the client, the SSI command is executed, and any script output is sent to the client.

Line 20 uses the **ECHO** command to display variable information. The `ECHO` command is followed by the keyword **VAR** and the name of the variable. For example, variable **DATE_GMT** contains the current date and time in Greenwich Mean Time (GMT). In line 25, the name of the current document is included in the XHTML page with the **DOCUMENT_NAME** variable. The **DATE_LOCAL** variable inserts the date in line 30 in local format—different formats are used around the world.

Figure 25.15 (fig25_15.pl) introduces file input and output in Perl. Line 8 opens (for input) the file `counter.dat`, which contains the number of hits to date for the `fig25_14.shtml` Web page. Function **open** is called to create a **filehandle** to refer to the file during the execution of the script. In this example, the file opened is assigned a filehandle named `COUNTREAD`.

```
1  #!C:\Perl\bin\perl
2  # Fig. 25.15: fig25_15.pl
3  # Program to track the number of times
4  # a Web page has been accessed.
5
6  use CGI qw( :standard );
7
8  open( COUNTREAD, "counter.dat" );
9  $data = <COUNTREAD>;
10 $data++;
11 close( COUNTREAD );
12
13 open( COUNTWRITE, ">counter.dat" );
14 print( COUNTWRITE $data );
15 close( COUNTWRITE );
16
17 print( header(), "<div style = \"text-align: center;
18                                     font-weight: bold\">" );
19 print( "You are visitor number", br() );
20
21 for ( $count = 0; $count < length( $data ); $count++ ) {
22     $number = substr( $data, $count, 1 );
23     print( img( { src => "images/$number.gif" } ), "\n" );
24 }
25
26 print( "</div>" );
```

Fig. 25.15 Perl script for counting Web page hits.

Line 9 uses the **diamond operator**, `<>`, to read one line of the file referred to by filehandle `COUNTREAD` and assign it to the variable `$data`. When the diamond operator is used in a scalar context, only one line is read. If assigned to an array, each line from the file is assigned to a successive element of the array. Because the file `counter.dat` contains only one line (in this case, only one number), the variable `$data` is assigned the value of that number in line 9. Line 10 then increments `$data` by 1. If the file does not yet exist when we try to open it, `$data` is assigned the value `undef`, which will be evaluated as 0 and incremented to 1 in line 10. In line 11, the connection to `counter.dat` is terminated by calling function **close**.

Now that the counter has been incremented for this hit, we write it back to the `counter.dat` file. In line 13, we open the `counter.dat` file for writing by preceding the file name with a `>` **character** (this is called **write mode**). This immediately truncates (i.e., discards) any data in that file. If the file does not exist, Perl creates a new one with the specified name. The first argument (`COUNTWRITE`) specifies the filehandle that will be used to refer to the file. Perl also provides an **append mode** (`>>`) for appending to the end of a file.

After line 13 executes, data can be written to the file `counter.dat`. Line 14 writes the counter number back to the file `counter.dat`. The first argument to `print` indicates the filehandle that refers to the file where data is written. If no filehandle is specified, `print` writes to standard out (STDOUT). Note that we need to use a space, rather than a comma, to separate the filehandle from the data. Line 15 closes the connect to the file `counter.dat`.

Lines 21–24 use a **for** statement to iterate through each digit of the number scalar `$data`. The **for** statement syntax consists of three semicolon-separated statements in parentheses, followed by a body delimited by curly braces. In our example, we iterate until `$count` is equal to `length($data)`. Function **length** returns the length of a character string, so the **for** iterates once for each digit in the variable `$data`. For instance, if `$data` stores the value "32", then the **for** iterates twice, first to process the value "3", and second to process the value "2". In the first iteration, `$count` equals 0 (as initialized in line 21), and the second time `$count` will equal 1. This is because the value of `$count` will be incremented for each loop (as specified by the statement `$count++`, also in line 21). For each iteration, we obtain the current digit by calling function **substr**. The first parameter passed to function **substr** specifies the string from which to obtain a substring. The second parameter specifies the offset, in characters, from the beginning of the string, so an offset of 0 returns the first character, 1 returns the second and so forth. The third argument specifies the length of the substring to be obtained (one character in this case). The **for** then assigns each digit (possibly from a multiple-digit number) to the scalar variable `$number`. Each digit's corresponding image is displayed using the **img** function (line 23).



Good Programming Practice 25.4

When opening a text file to read its contents, open the file in read-only mode. Opening the file in other modes increases the risk of overwriting the data accidentally.



Good Programming Practice 25.5

Always close files as soon as you are finished with them.

It is important in this example to think about file permissions and security. This program may not run correctly if the user's default settings do not allow scripts to manipulate files. To resolve this issue, the user can change the permissions in the folder where `counter.dat` resides so that all users have **Write** access. Check your Web server's documentation for instructions on configuring file permissions. The user should be aware that giving users **Write** access poses a security risk to the system. Security details are covered in Chapter 38, e-Business & e-Commerce.

25.7 Verifying a Username and Password

It is often desirable to have a **private Web site** that is visible only to certain people. Implementing privacy generally involves username and password verification. Figure 25.16 is an XHTML form that queries the user for a username and a password. It posts the fields `username` and `password` to the Perl script `fig25_17.pl` upon submission of the form. For simplicity, this example does not encrypt the data before sending it to the server.

The script `fig25_17.pl` (Fig. 25.17) is responsible for verifying the username and password of the client by crosschecking against values from a database. The database of valid users and their passwords is a simple text file: `password.txt` (Fig. 25.18).

```

1  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
2     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
3
4  <!-- Fig. 25.16: fig25_16.html -->
5
6  <html>
7      <head>
8          <title>Verifying a username and a password</title>
9      </head>
10
11     <body>
12         <p>
13             <div style = "font-family = arial">
14                 Type in your username and password below.
15             </div><br />
16
17             <div style = "color: #0000ff; font-family: arial;
18                 font-weight: bold; font-size: small">
19                 Note that the password will be sent as plain text.
20             </div>
21         </p>
22
23         <form action = "/cgi-bin/fig25_17.pl" method = "post">
24
25             <table style = "background-color: #dddddd">
26                 <tr>
27                     <td style = "font-face: arial;
28                         font-weight: bold">Username:</td>
29                 </tr>
30                 <tr>
31                     <td>
32                         <input name = "username" />
33                     </td>
34                 </tr>
35                 <tr>
36                     <td style = "font-face: arial;
37                         font-weight: bold">Password:</td>
38                 </tr>
39                 <tr>
40                     <td>
41                         <input name = "password" type = "password" />
42                     </td>
43                 </tr>
44                 <tr>
45                     <td>
46                         <input type = "submit" value = "Enter" />
47                     </td>
48                 </tr>
49             </table>
50         </form>
51     </body>
52 </html>

```

Fig. 25.16 XHTML form for submitting a username and a password to a Perl script. (Part 1 of 2.)

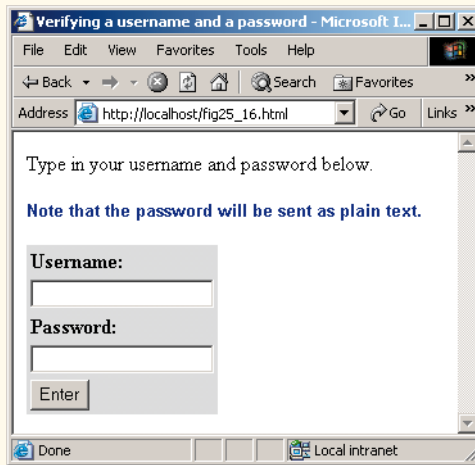


Fig. 25.16 XHTML form for submitting a username and a password to a Perl script. (Part 2 of 2.)

In line 14 of `fig25_17.pl`, we open the file `password.txt` (Fig. 25.18) for reading and assign it to the filehandle `FILE`. To verify that the file was opened successfully, a test is performed using the **logical OR operator** (`||`). Operator `||` returns true if either the left condition or the right condition evaluates to true. If the condition on the left evaluates to true, then the condition on the right is not evaluated. In this case, the function **die** executes only if `open` returns false, indicating that the file did not open properly. Function **die** displays an error message and terminates program execution.

```

1  #!C:\Perl\bin\perl
2  # Fig. 25.17: fig25_17.pl
3  # Program to search a database for usernames and passwords.
4
5  use CGI qw( :standard );
6
7  $dtd =
8  "-//W3C//DTD XHTML 1.0 Transitional//EN\"
9  "\"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd\"";
10
11 $testUsername = param( "username" );
12 $testPassword = param( "password" );
13
14 open( FILE, "password.txt" ) ||
15     die( "The database could not be opened." );
16
17 while ( $line = <FILE> ) {
18     chomp( $line );
19     ( $username, $password ) = split( ",", $line );
20

```

Fig. 25.17 Script to analyze the username and password submitted from an XHTML form. (Part 1 of 3.)


```
21     if ( $testUsername eq $username ) {
22         $userVerified = 1;
23     }
24     if ( $testPassword eq $password ) {
25         $passwordVerified = 1;
26         last;
27     }
28 }
29 }
30
31 close( FILE );
32
33 print( header() );
34 print( start_html( { dtd => $dtd,
35                     title => "Password Analyzed" } ) );
36
37 if ( $userVerified && $passwordVerified ) {
38     accessGranted();
39 }
40 elsif ( $userVerified && !$passwordVerified ) {
41     wrongPassword();
42 }
43 else {
44     accessDenied();
45 }
46
47 print( end_html() );
48
49 sub accessGranted
50 {
51     print( div( { style => "font-face: arial;
52                         color: blue;
53                         font-weight: bold" },
54               "Permission has been granted,
55               $username.", br(), "Enjoy the site." ) );
56 }
57
58 sub wrongPassword
59 {
60     print( div( { style => "font-face: arial;
61                         color: red;
62                         font-weight: bold" },
63               "You entered an invalid password.", br(),
64               "Access has been denied." ) );
65 }
66
67 sub accessDenied
68 {
69     print( div( { style => "font-face: arial;
70                         color: red;
71                         font-size: larger;
72                         font-weight: bold" },
```

Fig. 25.17 Script to analyze the username and password submitted from an XHTML form. (Part 2 of 3.)

```

73     "You have been denied access to this site." ) );
74 }

```

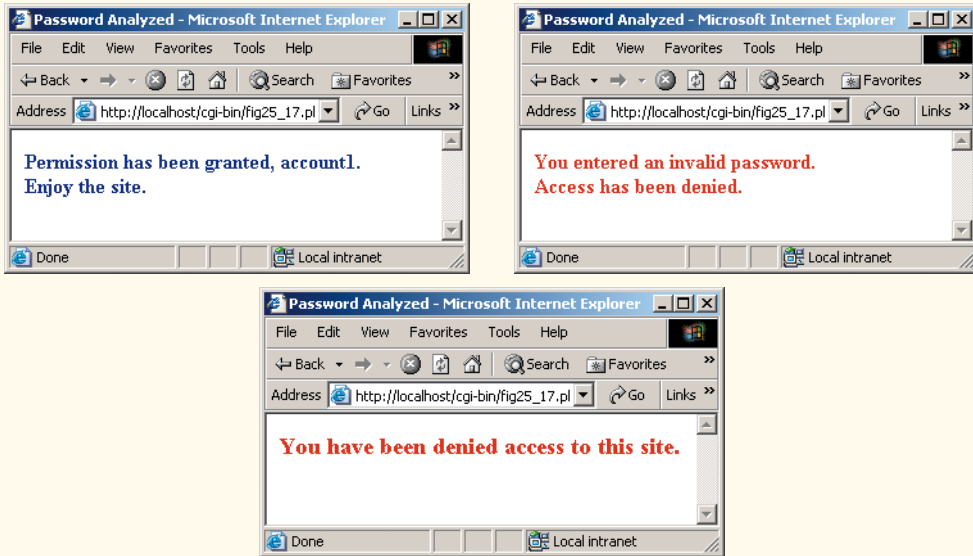


Fig. 25.17 Script to analyze the username and password submitted from an XHTML form. (Part 3 of 3.)

```

1  account1,password1
2  account2,password2
3  account3,password3
4  account4,password4
5  account5,password5
6  account6,password6
7  account7,password7
8  account8,password8
9  account9,password9
10 account10,password10

```

Fig. 25.18 Database `password.txt` containing usernames and passwords.



Good Programming Practice 25.6

Function `die` is useful to handle situations in which a program cannot continue. Rather than resulting in program errors, function `die` will cause the program to end with a message explaining the situation to the user.

The **while** statement (lines 17–29) repeatedly executes the code enclosed in curly braces until the condition in parentheses evaluates to false. In this case, the test condition assigns the next unread line of `password.txt` to `$line` and evaluates to true as long as a line from the file is successfully read. When the end of the file is reached, `<FILE>` returns false and the loop terminates.

Each line in `password.txt` (Fig. 25.18) consists of an account name and password pair, separated by a comma, and followed by a newline character. For each line read, function **chomp** is called (line 18) to remove the newline character at the end of the line. Then **split** is called to divide the string into substrings at the specified separator or **delimiter** (in this case, a comma). For example, the `split` of the first line in `password.txt` returns the list (`"account1"`, `"password1"`). The syntax

```
( $username, $password ) = split( ",", $line );
```

sets `$username` and `$password` to the first and second elements returned by `split` (`account1` and `password1`), respectively.

If the username entered is equivalent to the one we have read from the text file, the condition in line 21 returns true. The `$userVerified` variable is then set to 1. Next, the value of `$testPassword` is tested against the value in the `$password` variable (line 24). If the password matches, the `$passwordVerified` variable is set to 1. In this case, because a successful username-password match has been found, the **last** statement, used to exit a repetition structure prematurely, allows us to exit the `while` loop immediately in line 26.

We are finished reading from `password.txt`, so we **close** the file in line 31. Line 37 checks whether both the username and the password were verified, using the Perl **logical AND operator** (`&&`). If both conditions are true (i.e., if both variables evaluate to nonzero values), then the function `accessGranted` is called (lines 49–56), which sends a Web page to the client, indicating a successful login. Otherwise, the condition in the `elsif` (line 40) is tested to determine whether the user was verified but not the password. In this case, the function `wrongPassword` is called (lines 58–65). The unary **logical negation operator** (`!`) is used in line 40 to negate the value of `$passwordVerified` and test whether it is false. If the user is not recognized, function `accessDenied` is called, and a message indicating that permission has been denied is sent to the client (lines 67–74).

Perl allows programmers to define their own **functions** or **subroutines**. Keyword `sub` begins a function definition (lines 49, 58 and 67), and curly braces delimit the function body. To call a function (i.e., to execute the code within the function definition), use the function's name, followed by a pair of parentheses (line 38, 41 and 44).

25.8 Using DBI to Connect to a Database

Database connectivity allows system administrators to maintain information on user accounts, passwords, credit-card information, mailing lists, product inventory and similar matters. Databases allow companies to enter the world of electronic commerce and maintain crucial data. For more information on databases, please refer to Chapter 22.

To access various relational databases in Perl, we need an interface (in the form of software) that allows us to connect to and execute SQL operations (queries). The **Perl DBI (Database Interface)** allows us to do this. This interface was created to access different types of databases uniformly. In this section, we access and manipulate a MySQL database containing information on several of Deitel & Associates, Inc.'s publications.

The examples in this section require that MySQL (www.mysql.org) be installed. Please refer to the MySQL installation instructions posted at www.deitel.com/books/iw3HTP3/index.html. The Perl **DBI** module and the MySQL driver, **DBD::mysql** (specified in lines 6–7 of Fig. 25.19), are also required.

If you are using ActiveState Perl, you can download these files using the *Perl Package Manager* (PPM), which is part of ActiveState Perl. Using PPM, you can download and install Perl modules and packages (provided that you are connected to the Internet at the time you are running the program). To use PPM, type `ppm` at the command prompt. This command starts the package manager in **interactive mode**, providing you with the `ppm>` prompt. Type `install DBI` and press *Enter* to install DBI. To install the MySQL driver, type `install DBD-mysql` and press *Enter*. [Note: The specific package to install is named `DBD-mysql`, whereas the driver itself is referred to as `DBD:mysql`.] If you do not have the Perl Package Manager, you can search for the module or package on *CPAN*, the *Comprehensive Perl Archive Network* (www.cpan.org).

The final step in setting up a computer to run the following example is to copy the database `books` to the computer. This database is located in the Chapter 25 examples directory on the CD-ROM that accompanies this book. The examples directory contains a subfolder named `books`, which contains all the required database files. In your `mysql` directory (e.g., `C:\mysql`), a `data` directory exists, which contains MySQL databases. Each subfolder in the `data` directory represents a database and contains all the files that comprise that database. Copy the `books` folder into this `data` directory.

In Fig. 25.19, the client selects an author from a drop-down list (the authors are numbered by their ID value). When **Get Info** is clicked, the chosen author and the author's ID are posted to the Perl script in Fig. 25.20 that queries the database for all books published by that author. The results are displayed in an XHTML table. To create and execute SQL queries, we create DBI objects known as **handles**. Database handles create and manipulate a connection to a database, while **statement handles** create and submit SQL to a database.

```

1  #!C:\Perl\bin\perl
2  # Fig. 25.19: fig25_19.pl
3  # CGI program that generates a list of authors.
4
5  use CGI qw( :standard );
6  use DBI;
7  use DBD:mysql;
8
9  $dtd =
10 "-//W3C//DTD XHTML 1.0 Transitional//EN\"
11   \ "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd";
12
13 print( header() );
14
15 print( start_html( { dtd => $dtd,
16                     title => "Authors" } ) );
17
18 # connect to "books" database, no password needed
19 $databaseHandle = DBI->connect( "DBI:mysql:books",
20                                "root", "", { RaiseError => 1 } );
21
22 # retrieve the names and IDs of all authors
23 $query = "SELECT FirstName, LastName, AuthorID
24          FROM Authors ORDER BY LastName";

```

Fig. 25.19 Perl script that queries a MySQL database for authors. (Part 1 of 2.)

```

25
26 # prepare the query for execution, then execute it
27 # a prepared query can be executed multiple times
28 $statementHandle = $databaseHandle->prepare( $query );
29 $statementHandle->execute();
30
31 print( h2( "Choose an author:" ) );
32
33 print( start_form( { action => 'fig25_20.pl' } ) );
34
35 print( "<select name = \"author\">\n" );
36
37 # drop-down list contains the author and ID number
38 # method fetchrow_array returns a single row from the result
39 while ( @row = $statementHandle->fetchrow_array() ) {
40     print( "<option>" );
41     print( "$row[ 2 ]. $row[ 1 ], $row[ 0 ]" );
42     print( "</option>" );
43 }
44
45 print( "</select>\n" );
46
47 print( submit( { value => 'Get Info' } ) );
48 print( end_form(), end_html() );
49
50 # clean up -- close the statement and database handles
51 $databaseHandle->disconnect();
52 $statementHandle->finish();

```

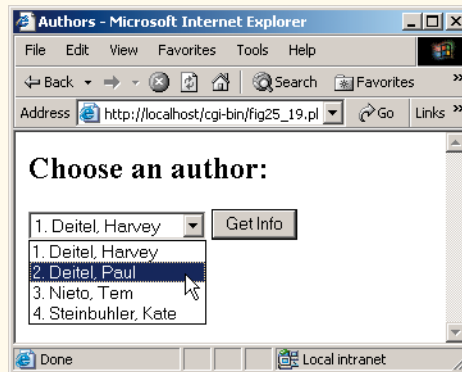


Fig. 25.19 Perl script that queries a MySQL database for authors. (Part 2 of 2.)

In lines 19–20, we connect to the database by calling DBI method **connect**. The first argument specifies the data source (i.e., the database). Note that we first specify the interface name (DBI), followed by a colon (:), then the database driver (`mysql`), followed by another colon and the name of the data source (`books`). The second argument specifies the user, and the third argument specifies the password for the database. This database does not require a username or password, so we simply use the empty string (`""`). The fourth argument (`{ RaiseError => 1 }`) is used for error checking. If an error occurs when trying to

connect to the database, function `die` is called and passed an error message. Setting this hash reference to 1 is like setting a variable to true—this value “turns on” the error checking, saving the programmer from writing extra code to handle the problem or from having the program crash unexpectedly. If the connection succeeds, function `connect` returns a database handle that is assigned to `$databaseHandle`.

In this example, we query the database for the names and IDs of the authors. We create this query in lines 23–24. In line 28, we use our database handle to prepare the query (using the method **prepare**). This method prepares the database driver for a statement that can be executed multiple times. The statement handle returned is assigned to `$statementHandle`. We execute the query by calling method **execute** in line 29.

Once the query has been executed, we can access the results by calling method **fetchrow_array** (line 39). Each call to this function returns the next set of data in the resulting table until there are no data sets left. A data set, or a row in the resulting table, contains one of the elements that satisfies the query. For example, in the first program, a query is executed that returns the ID and name of each author. This query creates a table that contains two columns, one for the author’s ID and one for the author’s name. A row contains the ID and name of a specific author. Each row is returned as an array and assigned to `@row`. We `print` these values as list options in lines 40–42. The option chosen is sent as the parameter “author” (line 35) to the Perl script in Fig. 25.20. In lines 51–52, we close the database connection (using method **disconnect**), and we specify that we are finished with this query by calling method **finish**. This function closes the statement handle and frees memory, especially if the resulting table is large.



Look-and-Feel Observation 25.1

Using tables to output fields in a database neatly organizes information into rows and columns.

Figure 25.20 presents the script `fig25_20.pl`, which queries the database for information about the specified author.

```

1  #!C:\Perl\bin\perl
2  # Fig. 25.20: fig25_20.pl
3  # CGI program to query a MySQL database.
4
5  use CGI qw( :standard );
6  use DBI;
7  use DBD: :mysql;
8
9  $dtd =
10 "-//W3C//DTD XHTML 1.0 Transitional//EN"
11   "\"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd\"";
12
13 print( header() );
14
15 # retrieve author's ID and name from the posted form
16 $authorID = substr( param( "author" ), 0, 1 );
17 $authorName = substr( param( "author" ), 3 );
18

```

Fig. 25.20 Perl script that queries a MySQL database for author information. (Part 1 of 3.)

```

19 print( start_html( { dtd => $dtd,
20                     title => "Books by $authorName" } ) );
21
22 $databaseHandle = DBI->connect( "DBI:mysql:books",
23                                 "root", "", { RaiseError => 1 } );
24
25 # use AuthorID to find all the ISBNs related to this author
26 $query1 = "SELECT ISBN FROM AuthorISBN
27           WHERE AuthorID = $authorID";
28
29 $statementHandle1 = $databaseHandle->prepare( $query1 );
30 $statementHandle1->execute();
31
32 print( h2( "$authorName" ) );
33
34 print( "<table border = 1>" );
35 print( th( "Title" ), th( "ISBN" ), th( "Publisher" ) );
36
37 while ( @isbn = $statementHandle1->fetchrow_array() ) {
38     print( "<tr>\n" );
39
40     # use ISBN to find the corresponding title
41     $query2 = "SELECT Title, PublisherID FROM titles
42             WHERE ISBN = \"'$isbn[ 0 ]'\"";
43     $statementHandle2 = $databaseHandle->prepare( $query2 );
44     $statementHandle2->execute();
45     @title_publisherID = $statementHandle2->fetchrow_array();
46
47     # use PublisherID to find the corresponding PublisherName
48     $query3 = "SELECT PublisherName FROM Publishers
49             WHERE PublisherID = \"'$title_publisherID[ 1 ]'\"";
50
51     $statementHandle3 = $databaseHandle->prepare( $query3 );
52     $statementHandle3->execute();
53     @publisher = $statementHandle3->fetchrow_array();
54
55     # print resulting values
56     print( td( $title_publisherID[ 0 ] ), "\n" );
57     print( td( $isbn[ 0 ] ), "\n" );
58     print( td( $publisher[ 0 ] ), "\n" );
59
60     print( "</tr>" );
61
62     $statementHandle2->finish();
63     $statementHandle3->finish();
64 }
65
66 print( "</table>" );
67
68 print( end_html() );
69
70 $databaseHandle->disconnect();
71

```

Fig. 25.20 Perl script that queries a MySQL database for author information. (Part 2 of 3.)


```
72 $statementHandle1->finish();
```

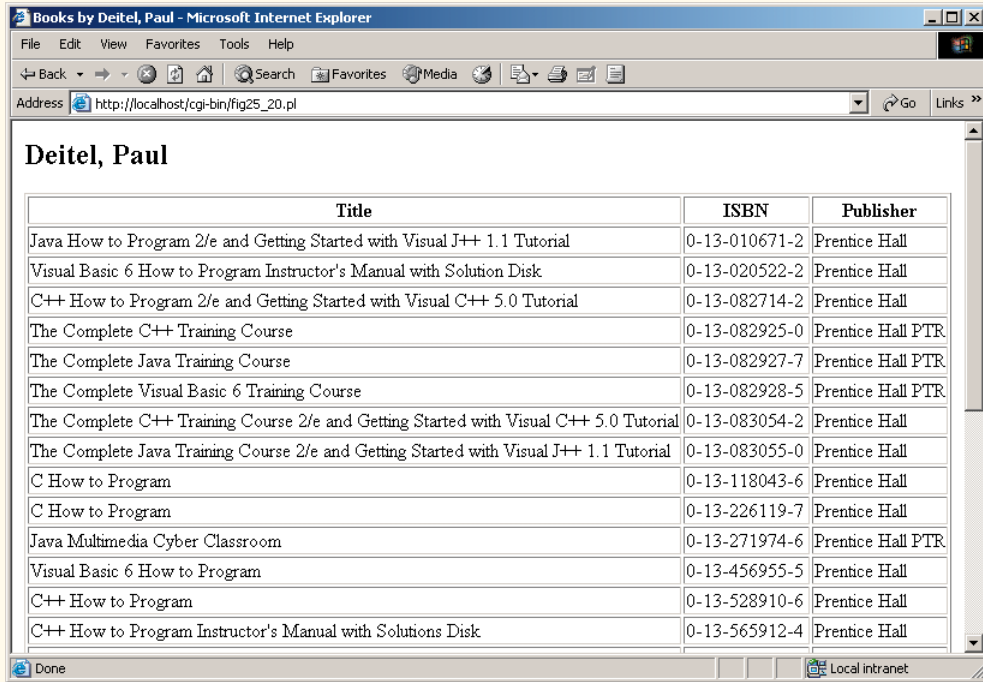


Fig. 25.20 Perl script that queries a MySQL database for author information. (Part 3 of 3.)

This program creates an XHTML page that displays the title of each book written by the current author, along with the ISBN number and book publisher. To obtain this information, we need the author's ID number, because the `AuthorISBN` table contains a field for the author's ID, not the author's name. Recall that the author's ID was submitted to this script by `fig25_19.pl`. The ID is the numerical value that precedes the author's name in the `author` parameter. To retrieve the ID and author name, we call method `substr` in lines 16–17. This statement returns the first character in the string (an offset of zero indicates the beginning of the string), which contains the ID value. In line 17, we specify an offset of three, because the author's name begins after the third character. Note that in this call we do not specify a length, because we want all the characters from the offset to the end of the string.

After connecting to the database, we specify and execute our first query in lines 26–30. This query returns all the ISBN numbers for the specified author. We place these values in a table. In line 37, we begin a `while` loop that iterates through each row matched by the query. The rows are retrieved by calling `fetchrow_array`, which returns the current data set as an array. When there are no more data sets to return, the condition evaluates to false. Within the loop, we use the ISBNs to obtain the title and publisher values for the current table row. The query in lines 41–45 uses the ISBN value to determine the book's title and the publisher's ID number. The next query (lines 48–53) uses the publisher's ID to determine the name of the publisher. These values are printed in lines 57–59.

25.9 Cookies and Perl

Cookies maintain **state information** for each client who uses a Web browser. Preserving this information allows data and settings to be retained even after the execution of a CGI script has ended. Cookies are used to record preferences (or other information) for the next time the client visits a Web site. For example, many Web sites use cookies to store each client's postal zip code. The zip code is used when the client asks a Web page to send, for instance, current weather information or news updates for the client's region. On the server side, cookies may be used to track information about client activity in order to determine which sites are visited most frequently or how effective certain advertisements and products are.

Microsoft Internet Explorer stores cookies as small text files saved on the client's hard drive. The data stored in the cookies is sent back to the server whenever the user requests a Web page from that server. The server can then serve XHTML content to the client that is specific to the information stored in the cookie.

Writing Cookies

Figure 25.21 uses a script to write a cookie to a client's machine. The `fig25_21.html` file is used to display an XHTML form that allows a user to enter a name, height and favorite color. When the user clicks the **Write Cookie** button, the `fig25_22.pl` script (Fig. 25.22) executes.



Good Programming Practice 25.7

Personal identifying information, such as credit-card numbers and passwords, should not be stored using cookies. Cookies cannot be used to retrieve information such as e-mail addresses or other data stored on the hard drive of a client's computer.

```

1  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
2    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
3
4  <!-- Fig. 25.21: fig25_21.html -->
5
6  <html>
7    <head>
8      <title>Writing a cookie to the client computer</title>
9    </head>
10
11   <body style = "font-face: arial">
12     <div style = "font-size: large;
13       font-weight: bold">
14       Click Write Cookie to save your cookie data.
15     </div><br />
16
17     <form method = "post" action = "cgi-bin/fig25_22.pl"
18       style = "font-weight: bold">
19       Name:<br />
20       <input type = "text" name = "name" /><br />
21       Height:<br />
22       <input type = "text" name = "height" /><br />
23       Favorite Color:<br />
24       <input type = "text" name = "color" /><br />

```

Fig. 25.21 XHTML document to read in cookie data from the user. (Part 1 of 2.)

```

25         <input type = "submit" value = "Write Cookie" />
26     </form>
27 </font>
28 </body>
29 </html>

```

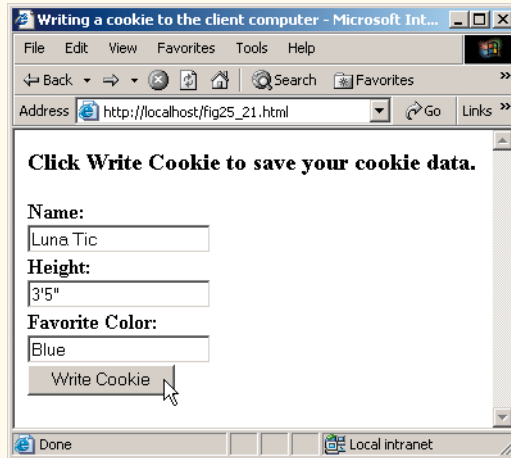


Fig. 25.21 XHTML document to read in cookie data from the user. (Part 2 of 2.)

```

1  #!C:\Perl\bin\perl
2  # Fig. 25.22: fig25_22.pl
3  # Program to write a cookie to a client's machine.
4
5  use CGI qw( :standard );
6
7  $name = param( "name" );
8  $height = param( "height" );
9  $color = param( "color" );
10
11  $expires = gmtime( time() + 86400 );
12
13  print( "Set-Cookie: Name=$name; expires=$expires; path=\n" );
14  print( "Set-Cookie: Height=$height; expires=$expires; path=\n" );
15  print( "Set-Cookie: Color=$color; expires=$expires; path=\n" );
16
17  $dtd =
18  "-//W3C//DTD XHTML 1.0 Transitional//EN"
19  "\http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd";
20
21  print( header() );
22  print( start_html( { dtd => $dtd,
23                      title => "Cookie Saved" } ) );
24
25  print <<End_Data;
26  <div style = "font-face: arial; font-size: larger">

```

Fig. 25.22 Writing a cookie to the client. (Part 1 of 2.)

```

27     The cookie has been set with the following data:
28     </div><br /><br />
29
30     <span style = "color: blue">
31     Name: <span style = "color: black">$name</span><br />
32     Height: <span style = "color: black">$height</span><br />
33     Favorite Color:</span>
34
35     <span style = "color: $color"> $color</span><br />
36     <br />Click <a href = "fig25_25.pl">here</a>
37     to read saved cookie.
38     End_Data
39
40     print( end_html() );

```

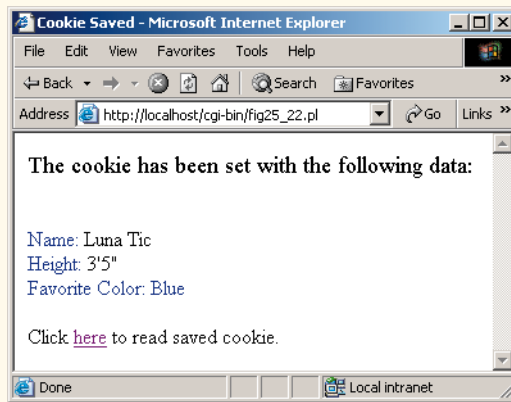


Fig. 25.22 Writing a cookie to the client. (Part 2 of 2.)

The `fig25_22.pl` script (Fig. 25.22) reads the data sent from the client in lines 7–9. Line 11 declares and initializes variable `$expires` to contain the expiration date of the cookie. The browser deletes a cookie after it expires. This script dynamically sets the cookie's expiration date to be one day in the future (i.e., the cookie will be deleted one day after it is created). Function `time` returns the current date and time as a measure of the number of seconds since the epoch (i.e., January 1, 1970 on most systems, but January 1, 1904 on Mac OS). Adding 86400 to this value produces a date and time that is exactly one day ahead of the current date and time (86400 seconds = 60 seconds/minute × 60 minutes/hour × 24 hours/day). A measure of the date and time as the number of seconds since the epoch cannot be used to specify a cookie's expiration date, however. Function `gmtime` converts this representation of the date and time to a string representation of the date and time (e.g., "Thu Aug 14 12:00:00 2003"), localized to Coordinated Universal Time (abbreviated UTC)—formerly Greenwich Mean Time (GMT). The script assigns this string to the variable `$expires` to set the cookie's expiration date to be exactly 24 hours from the moment the cookie is created.

Lines 13–15 call function `print` to output the cookie information. We use the **Set-Cookie:** header to indicate that the browser should store the incoming data in a cookie. The header sets three attributes for each cookie: A name-value pair containing the data to be stored, the expiration date and the URL path of the server domain over which the cookie

is valid. For this example, no path is given, making the cookie readable from anywhere within the server's domain. Lines 21–40 create a Web page indicating that the cookie has been written to the client.

In lines 25–38 we see our first **“here” document**. Line 25 instructs the Perl interpreter to print the subsequent lines verbatim (after variable interpolation) until it reaches the `End_Data` label (line 38). This label consists simply of the identifier `End_Data`, placed at the beginning of a line, with no whitespace characters preceding it, and followed immediately with a newline. “Here” documents are often used in CGI programs to eliminate the need to call function `print` repeatedly. Note that we use functions in the CGI library, as well as “here” documents, to create a clean program.

If the client is an Internet Explorer browser, cookies are stored in the `Cookies` directory on the client's machine. Figure 25.23 shows the contents of this directory on Windows prior to the execution of `fig25_22.pl`. After the cookie is written, an additional text file is added to this list. The file `luna tic@localhost[1].txt` can be seen in the `Cookies` directory in Fig. 25.24. The domain for which the cookie is valid is `localhost`. The username `luna tic`, however, is only part of name of the file Internet Explorer uses to store the cookie and is not actually a part of the cookie itself. Therefore, a remote server cannot access the username.

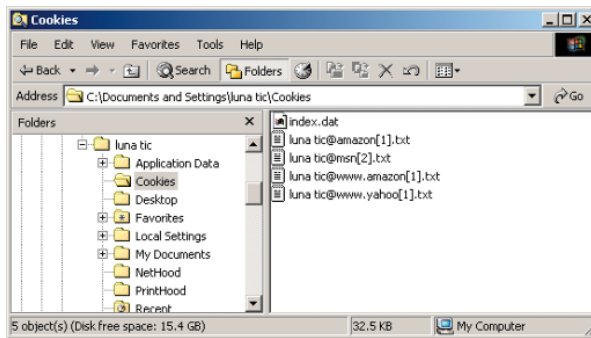


Fig. 25.23 Cookies directory before a cookie is written.

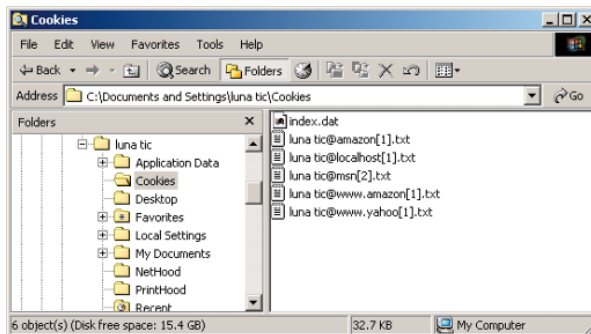


Fig. 25.24 Cookies directory after a cookie is written.

Reading an Existing Cookie

Figure 25.25 (fig25_25.pl) reads the cookie written in Fig. 25.22 and displays the information in a table. Environment variable **HTTP_COOKIE** contains the client's cookies. Line 25 calls subroutine `readCookies` (lines 37–47) and places the returned value into hash `%cookies`. The user-defined subroutine `readCookies` splits the environment variable containing the cookie information into separate cookies (using `split`) and stores them as distinct elements in `@cookieArray` (line 39). For each cookie in `@cookieArray`, we call `split` again to obtain the original name-value pair, which, in turn, is stored in `%cookieHash` in line 43.

```

1  #!C:\Perl\bin\perl
2  # Fig. 25.25: fig25_25.pl
3  # program to read cookies from the client's computer.
4
5  use CGI qw( :standard );
6
7  $dtd =
8  "-//W3C//DTD XHTML 1.0 Transitional//EN\"
9  \"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd";
10
11 print( header() );
12 print( start_html( { dtd => $dtd,
13                     title => "Read Cookies" } ) );
14
15 print( div( { style => "font-face: arial;
16                     font-size: larger;
17                     font-weight: bold" },
18         "The following data is saved in a
19         cookie on your computer." ), br() );
20
21 print( "<table style = \"background-color: #aaaaaa\"
22         border = 5 cellpadding = 10
23         cellspacing = 0>" );
24
25 %cookies = readCookies();
26 $color = $cookies{ Color };
27
28 foreach $cookieName ( "Name", "Height", "Color" ) {
29     print( Tr( td( { style => "background-color: $color" },
30                  $cookieName ),
31              td( $cookies{ $cookieName } ) ) );
32 }
33
34 print( "<table>" );
35 print( end_html() );
36
37 sub readCookies
38 {
39     @cookieArray = split( "; ", $ENV{ 'HTTP_COOKIE' } );
40
41     foreach ( @cookieArray ) {
42         ( $cookieName, $cookieValue ) = split( "=", $_ );

```

Fig. 25.25 Output displaying the cookie's content. (Part 1 of 2.)

```

43     $cookieHash{ $cookieName } = $cookieValue;
44 }
45
46     return %cookieHash;
47 }

```



Fig. 25.25 Output displaying the cookie's content. (Part 2 of 2.)

The `split` function in line 42 makes reference to a variable named `$_`. The special Perl variable `$_` is used as a default argument for many Perl functions. In this case, because no variable was provided in the `foreach` loop (line 41), `$_` is used by default. Thus, in this example, `$_` is assigned the value of the current element of `@cookieArray` as a `foreach` statement iterates through it.

Once `%cookieHash` has been created, line 46 returns it from the function (using the `return` keyword), and `%cookies` is assigned this value in line 25. A `foreach` loop (lines 28–32) then iterates through the hash with the given key names, printing the key and value for the data from the cookie in an XHTML table.

Be aware that users can disable cookies on their machines. There are a few ways to handle this issue, but the most basic is to create a file similar to a cookie that would be stored on the server's computer rather than the client's computer. For more information, the reader can visit the Web sites listed at the end of this chapter.

25.10 Operator Precedence Chart

This section contains the operator precedence chart for Perl (Fig. 25.26). The operators are shown in decreasing order of precedence, from top to bottom.

Operator	Type	Associativity
terms and list operators	<code>print @array</code> or <code>sort (4, 2, 7)</code>	left to right
<code>-></code>	member access	left to right

Fig. 25.26 Perl operator precedence chart. (Part 1 of 3.)

Operator	Type	Associativity
++	increment	none
--	decrement	
**	exponentiation	right to left
!	logical NOT	right to left
~	bitwise one's complement	
\	reference	
+	unary plus	
-	unary minus	
=~	matching	left to right
!~	negated match	
*	multiplication	left to right
/	division	
%	modulus	
x	repetition	
+	addition	left to right
-	subtraction	
.	string concatenation	
<<	left shift	left to right
>>	right shift	
named unary operators	unary operators—e.g., -e (filetest)	none
<	numerical less than	none
>	numerical greater than	
<=	numerical less than or equal to	
>=	numerical greater than or equal to	
lt	string less than	
gt	string greater than	
le	string less than or equal to	
ge	string greater than or equal to	
==	numerical equality	none
!=	numerical inequality	
<=>	numerical comparison (returns -1, 0 or 1)	
eq	string equality	
ne	string inequality	
cmp	string comparison (returns -1, 0 or 1)	
&	bitwise AND	left to right
	bitwise inclusive OR	left to right
^	bitwise exclusive OR	
&&	logical AND	left to right
	logical OR	left to right
..	range operator	none

Fig. 25.26 Perl operator precedence chart. (Part 2 of 3.)

Operator	Type	Associativity
<code>? :</code>	conditional operator	right to left
<code>=</code>	assignment	right to left
<code>+=</code>	addition assignment	
<code>-=</code>	subtraction assignment	
<code>*=</code>	multiplication assignment	
<code>/=</code>	division assignment	
<code>%=</code>	modulus assignment	
<code>**=</code>	exponentiation assignment	
<code>.=</code>	string concatenation assignment	
<code>x=</code>	repetition assignment	
<code>&=</code>	bitwise AND assignment	
<code> =</code>	bitwise inclusive OR assignment	
<code><<=</code>	bitwise exclusive OR assignment	
<code>>>=</code>	left shift assignment	
<code>&&=</code>	right shift assignment	
<code> =</code>	logical AND assignment	
	logical OR assignment	
<code>,</code>	expression separator; returns value of last expression	left to right
<code>=></code>	expression separator; groups two expressions	
<code>not</code>	logical NOT	right to left
<code>and</code>	logical AND	left to right
<code>or</code>	logical OR	left to right
<code>xor</code>	logical exclusive OR	

Fig. 25.26 Perl operator precedence chart. (Part 3 of 3.)

25.11 Web Resources

There is a strongly established Perl community online that has made available a wealth of information on the Perl language, Perl modules and CGI scripting.

www.perl.com

Perl.com: The Source for Perl is the first place to look for information about Perl. The home page provides up-to-date news on Perl, answers to frequent questions about Perl and an impressive collection of links to Perl resources on the Internet. The links include sites for Perl software, tutorials, user groups and demos.

www.pm.org

This site is the home page of *Perl Mongers*, a group dedicated to supporting the Perl community. The site is helpful in finding others in the Perl community with whom to converse; Perl Mongers has established Perl user groups around the globe.

www.perl.org

This Perl Mongers site is a great one-stop resource for developers. Resources include documentation, links to several other Perl sites and mailing lists.

www.activestate.com

From this site you can download ActivePerl—the Perl 5.8 implementation for Windows.

www.cpan.org

The *Comprehensive Perl Archive Network* includes an extensive listing of Perl-related information.

www.perl.com/CPAN/scripts/index.html

This site is the scripts index from the CPAN archive. You will find a wealth of scripts written in Perl.

www.speakeasy.org/~cgires

This site is a collection of tutorials and scripts that can provide a thorough understanding of CGI.

www.perlarchive.com

This site features a large number of scripts and guides, as well as a learning center that includes helpful articles.

www.cgi.resourceindex.com

General CGI site including scripts, a list of freelance CGI programmers, documentation, job listings and several other resources.

www.cgi101.com

CGI 101 is a site for those looking to improve their programming ability through familiarity with CGI. The site contains a six-chapter class outlining techniques for CGI programming in the Perl language. The class includes both basic and advanced scripts, with working examples. Also available at the site are script libraries and links to other helpful sources.

www.freeperlcode.com

This site provides a help guide and access to several Perl scripts that can be easily downloaded and installed.

www.jmarshall.com/easy/cgi

This site provides a good, brief explanation of CGI for those with programming experience.

www.wdvl.com/Authoring/Languages/Perl

This site contains many links to Perl resources.

www.wdvl.com/Authoring/CGI

The *Web Developer's Virtual Library* provides tutorials for learning both CGI and Perl.

www.perlmonth.com

Perlmonth is a monthly online periodical devoted to Perl, with featured articles from professional programmers. This site is a good source for those who use Perl frequently and wish to keep up on the latest developments.

tpj.com

The *Perl Journal* is a large magazine dedicated to Perl. Subscribers are provided with up-to-date Perl news and articles, on the Internet or in printed form.

www.1024kb.net/index.html?./texts/perlnet.html

This page provides a brief tutorial on Perl network programming for those who already know the language. The tutorial uses code examples to explain the basics of network communication.

www.w3.org/CGI

The World Wide Web Consortium page on CGI is concerned with CGI security issues. This page provides links to CGI specifications, as indicated by the National Center for Supercomputing Applications (NCSA).

SUMMARY

- The Common Gateway Interface (CGI) is a standard protocol through which applications interact with Web servers. CGI provides a way for clients to interface indirectly with applications on the Web server.
- Because CGI is an interface, it cannot be programmed directly; a script or executable program (commonly called a CGI script) must be executed to interact with it.

- CGI scripts often process information gathered from a form. These programs are typically designated with a certain filename extension (e.g., `.cgi` or `.pl`) or located within a special directory (e.g., `cgi-bin`). After the application output is sent to the server through CGI, the results may be sent to the client.
- Standard input is the stream of information received by a program from a user, typically through the keyboard, but also possibly from a file or another input device.
- Standard output is the information stream presented to the user by an application; it is typically displayed on the screen, but may also be printed by a printer, written to a file, etc.
- The Perl comment character (`#`) instructs the interpreter to ignore everything on the current line following the `#`. The exception to this rule is the “shebang” construct (`#!`). On Unix systems, this line indicates the path to the Perl interpreter. On other systems, the line may be ignored or may indicate to the server that a Perl program follows the statement.
- Perl program file names typically end with the `.pl` extension. Programs can be executed by running the Perl interpreter from the command-line prompt (e.g., the Command Prompt in Windows).
- Using the `-w` option when running a Perl program instructs the interpreter to output warnings to the screen if it finds bugs in your code.
- Function `print` is used to output text.
- Text surrounded by quotes is called a string.
- Escape sequences can be used to output special characters, such as newlines.
- Semicolons (`;`) are used to terminate Perl statements.
- Perl has built-in data types that represent different kinds of data, including scalar, hash and array.
- The `$` character specifies that a variable contains a scalar value.
- The `@` character specifies that a variable contains an array, while the `%` character specifies that a variable contains a hash.
- In Perl, a variable is created the first time it is encountered by the interpreter.
- When a variable is encountered inside a double-quoted (`""`) string, Perl uses a process called interpolation to replace the variable with its associated data.
- In Perl, uninitialized variables have the value `undef`, which can be evaluated differently depending on context. When `undef` is found in a numeric context, it evaluates to 0. When it is interpreted in a string context, `undef` evaluates to the empty string (`""`).
- Unless a string begins with a digit, it is evaluated as `undef` in a numeric context. If the string does begin with a digit, every character up to the first nondigit character is evaluated as a number, and the remaining characters are ignored.
- The programmer does not need to differentiate between numeric and string data types because the interpreter evaluates scalar variables depending on the context in which they are used.
- Several values can be stored in arrays, which are divided into elements, each containing a scalar value. Array variable names are preceded by the `@` symbol.
- When `printing` an array inside double quotes, the array element values are printed with only one space separating them.
- Individual array elements are accessed using square brackets (`[]`). If the array name is prefaced by the `$` character and followed by an index number in square brackets, it refers instead to an individual array element, which is a scalar value.
- The range operator (`..`) is used to specify all the values in a range.
- It is not necessary to specify an array’s size. The Perl interpreter recognizes that memory has not been allocated for this array and creates new memory automatically.

- When strings are inside single quotes, the interpreter treats the string literally and does not attempt to interpret any escape sequence or variable substitution.
- Text manipulation in Perl is usually done with regular expressions—a series of characters that serve as pattern-matching templates in strings, text files and databases.
- Operator `qw` (“quote word”) takes the contents inside the parentheses and creates a comma-separated list with each element wrapped in double quotes.
- The `foreach` statement iterates sequentially through the elements in a specified array or the elements in a range of values.
- The `if` statement is used to execute code depending on a specified condition.
- In Perl, anything except the number 0, the string “0” and the empty string (i.e., `undef` values) is defined as true.
- Operators `ne`, `lt` and `gt` test strings for equality, less than and greater than, respectively. These operators are used only with strings. When comparing numeric values, operators `==`, `!=`, `<`, `<=`, `>` and `>=` are used.
- Perl provides the match operator (`m/pattern/` or `/pattern/`), which uses regular expressions to search a string for a specified *pattern*.
- The match operator takes two operands. The first operand is the regular-expression pattern for which to search; it is placed between the slashes of the `m//` operator. The second operand is the string to search within; it is assigned to the match operator by using the `=~` (binding) operator.
- Regular expressions can include special characters, called metacharacters, that can specify patterns or contexts that cannot be defined using literal characters.
- The caret metacharacter (`^`) searches the beginning of a string for a pattern.
- The `$` metacharacter searches the end of a string for a pattern.
- The `\b` expression matches any word boundary.
- The `+` modifier is a quantifier that instructs Perl to match the preceding character one or more times.
- Parentheses indicate that the text matching the pattern is to be saved in a special Perl variable.
- Modifying characters placed to the right of the forward slash that delimits a regular expression instruct the interpreter to treat the expression in different ways.
- Placing an `i` after the regular expression tells the interpreter to ignore case when searching.
- Placing an `x` after the regular expression indicates that whitespace characters are to be ignored.
- Modifying character `g` indicates a global search—a search that does not stop after the first match is found.
- Environment variables contain information about the environment in which a script is being run.
- The `use` directive directs Perl programs to include the contents of predefined packages, called modules.
- The `CGI` module contains many useful functions for CGI scripting in Perl.
- With the `use` directive, we can specify an import tag to include a predefined set of functions.
- We usually specify the import tag `:standard` when importing the `CGI.pm` module to specify the standard CGI functions.
- Function `header` directs the Perl program to output a valid HTTP header.
- The `start_html` function begins the output of XHTML. This function will print the document type definition for this document, as well as several opening XHTML tags.
- When using many of the functions in the `CGI` module, attribute information can be specified within curly braces.

- Each argument within the curly braces is in the form of a key–value pair. A key (or value name) is assigned a value using the arrow operator (`=>`), where the key is to the left of the arrow, and the value is to the right.
- Function `Tr` contains its arguments within table row tags.
- Function `th` contains its arguments within table header tags.
- Function `hr` creates horizontal rules.
- Function `td` contains its arguments within table data tags.
- The hash data type is designated by the `%` character and represents an unordered set of scalar-value pairs.
- Each element in a hash is accessed by using a unique key that is associated with a value.
- Hash values are accessed by using the syntax `$hashName{ keyName }`.
- Function `keys` returns an array of all the keys from a specified hash in no specific order because elements have no defined order.
- We use function `sort` to order the array of keys alphabetically.
- The `%ENV` hash is a built-in table that contains the names and values of all the environment variables.
- Function `end_html` outputs the closing tags for a page (`</body>` and `</html>`).
- Function `param` is used to retrieve values from form field elements.
- Regular expressions can be used to validate information in a CGI script. The design of verifying information is called business logic (also called business rules).
- Function `br` adds a break (`
`) to the XHTML page.
- Function `span` adds `span` elements to a page.
- Function `div` adds `div` elements to a page.
- Server-side includes (SSIs) are commands embedded in HTML documents to allow simple dynamic content creation.
- The command `EXEC` can be used to run CGI scripts and embed their output directly into a Web page. Before the XHTML document is sent to the client, the SSI command `EXEC` is executed and any script output is sent to the client.
- A document containing SSI commands is typically given the `.shtml` file extension. The `.shtml` files are parsed by the server.
- The `ECHO` command is used to display variable information. It is followed by the keyword `VAR` and the name of the variable.
- The variable `DATE_GMT` contains the current date and time in Greenwich Mean Time (GMT).
- The name of the current document is specified as the `DOCUMENT_NAME` variable.
- The `DATE_LOCAL` variable inserts the date.
- Function `open` is called to open a file and create a filehandle to be associated with the file.
- The diamond operator (`<>`) is used to read input from the user or a file. When the diamond operator is used in a scalar context, only one line is read. When the operator is used in list context, all the input (or the entire file) is read and assigned to values in the list.
- We open a file for writing by preceding the file name with a `>` character. Perl also provides an append mode (`>>`) for appending to the end of a file.
- A `for` statement is similar to a `foreach` statement. It iterates through a set of values, specified in parentheses after the keyword `for`. Within the parentheses, three statements are used to indicate the values through which the structure will iterate.

- Function `length` returns the length of a character string.
- Function `substr` is used to identify a specified substring.
- The `img` function is used to display images.
- Function `die` displays an error message and terminates the program.
- Function `chomp` removes the newline character from the end of a string, if a newline exists.
- Function `split` divides a string into substrings at the specified separator or delimiter.
- The `last` statement is used to exit a loop structure once a desired condition has been satisfied.
- Perl allows programmers to define their own functions or subroutines. Keyword `sub` begins a function definition, and curly braces delimit the function body.
- Database connectivity allows system administrators to maintain crucial data.
- The Perl Database Interface (DBI) allows access to various relational databases in a uniform manner.
- The Perl DBI module and the MySQL driver, `DBD::mysql` are required to access and manipulate a MySQL database from a Perl program.
- The Perl Package Manager (PPM) is designed so that the user can easily download and install several Perl modules and packages. Perl modules and packages can also be found on the Comprehensive Perl Archive Network (CPAN).
- To create and execute SQL queries, we create DBI objects known as handles.
- Database handles create and manipulate a connection to a database.
- Statement handles create and submit SQL to a database.
- Method `connect` in module DBI sets up a database connection and returns a database handle.
- A database handle is used to prepare a database query (using the method `prepare` in module DBI). This method prepares the database driver for a query that can be executed multiple times.
- We execute a query by calling method `execute` in module DBI.
- Once a query has been executed, we can access the results using the method `fetchrow_array` in module DBI. Each call to this function returns the next set of data in the resulting table until there are no data sets left.
- A database connection can be closed using method `disconnect` in module DBI.
- We can indicate that we are no longer using a query by calling method `finish` in module DBI.
- Cookies maintain state information for each client who uses a Web browser. Microsoft Internet Explorer stores cookies as small text files saved on the client's hard drive.
- Function `time` returns the current date and time as a measure of the number of seconds since the epoch (i.e., January 1, 1970 on most systems, but January 1, 1904 on Mac OS).
- Function `gmtime` converts a date and time from a measure of seconds since the epoch to a string representation of the date and time, localized to Coordinated Universal Time (abbreviated UTC)—formerly Greenwich Mean Time (GMT).
- We use the `Set-Cookie:` header to indicate that the browser should store the incoming data in a cookie.
- A “here” document is used to output a string verbatim. The string is specified as all the text from the beginning of the document to the closing identifier.
- Environment variable `HTTP_COOKIE` contains the client's cookies.
- The special variable `$_` is used as a default argument for many Perl functions.

TERMINOLOGY

!= operator	cgi-bin directory
\$ metacharacter	chomp function
\$ type symbol	close function
\$_ special variable	comment character (#)
% type symbol	Common Gateway Interface (CGI)
* quantifier	comparison operator
? quantifier	Comprehensive Perl Archive Network (CPAN)
@ type symbol	connect method
\b metacharacter	cookie
\B metacharacter	Cookies directory
\d metacharacter	database connectivity
\D metacharacter	database handle
\n escape sequence	DATE_GMT variable
\n metacharacter	DATE_LOCAL variable
\s metacharacter	DBD::mysql driver
\S metacharacter	DBI module
\t metacharacter	delimiter
\w metacharacter	diamond operator (<>)
\W metacharacter	die function
\w pattern	disconnect method
^ metacharacter	div function
{ } curly braces in CGI.pm functions	DOCUMENT_NAME variable
{m,n} quantifier	dtd argument in start_html function
{n,} quantifier	ECHO command
{n} quantifier	elements in an array
+ quantifier	empty string
< operator	end_html function
<= operator	%ENV hash
<> diamond operator	environment variable
== operator	equality operators
> operator	escape sequence
> write mode	escaping special characters
>= operator	EXEC command
>> append mode	execute method
alphanumeric character	fetchrow_array method
Apache Web server	filehandle
append mode (>>)	finish method
array	for statement
assignment operator	foreach statement
associative array	forms
binding operator (=~)	function
br function	g modifying character
built-in metacharacter	gt operator
business logic	handle
business rules	hash
.cgi file extension	header function
CGI module	hr function
CGI script	HTTP connection
CGI tutorial	HTTP header

HTTP host	quantifier
HTTP_COOKIE environment variable	qw operator
HTTP_HOST environment variable	range operator (..)
i modifying character	read-only mode
if statement	redirection
img function	regular expression
import tag	return keyword
INCLUDE command	s modifying character
interactive mode	scalar
interpolation	scalar value
keys (in a hash)	semicolons (;) to terminate statement
keys function	Server-Side Include (SSI)
last statement	Set-Cookie: header
length function	shebang construct (!)
list	.shhtml file extension
literal character	sort function
logical AND (&&) operator	span function
logical negation (!) operator	split function
logical OR () operator	SQL query string
lt operator	SSI (Server-Side Include)
m modifying character	:standard import tag
m//	standard input (STDIN)
match operator (m//)	standard output (STDOUT)
metacharacter	start_html function
modifying character	state information
module	statement handle
MySQL database	string context
mysql directory	sub keyword
MySQL driver	subroutine
ne operator	substr method
numeric context	td function
open function	th function
param function	title argument in start_html function
Perl (Practical Extraction and Report Language)	tr operator
perl command	Tr function
.pl file extension	undef value
Perl interpreter	Unisys
Perl Package Manager (PPM)	use directive
piping	VAR keyword
post method	-w command-line option in Perl
ppm method	while statement
prepare method	word boundary
print function	write mode (>)
private Web site	x modifying character

SELF-REVIEW EXERCISES

25.1 Fill in the blanks in the following statements.

- The _____ Protocol is used by Web browsers and Web servers to communicate with each other.
- Typically, all CGI programs reside in directory _____.

- c) To output warnings as a Perl program executes, the _____ command-line option should be used.
- d) The three data types in Perl are _____, _____ and _____.
- e) _____ are divided into individual elements that can each contain an individual scalar variable.
- f) To test the equality of two strings, operator _____ should be used.
- g) Business _____ is used to ensure that invalid data are not entered into a database.
- h) _____ allow Webmasters to include the current time, date or even the contents of a different HTML document.
- i) The _____ statement iterates once for each element in a list or array.
- j) Many Perl functions take special variable _____ as a default argument.

25.2 State whether each of the following is *true* or *false*. If *false*, explain why.

- a) Documents containing server-side includes must have a file extension of .SSI in order to be parsed by the server.
- b) A valid HTTP header must be sent to the client to ensure that the browser correctly displays the information.
- c) The numerical equality operator, eq, is used to determine whether two numbers are equal.
- d) The ^ metacharacter is used to match the beginning of a string.
- e) Perl has a built-in binding operator, =, that tests whether a matching string is found within a variable.
- f) Cookies can read information, such as e-mail addresses and personal files, from a client's hard drive.
- g) An example of a valid HTTP header is Content-type text/html.
- h) CGI environment variables contain such information as the type of Web browser the client is running.
- i) The characters \w in a regular expression match only a letter or number.
- j) CGI is a programming language that can be used in conjunction with Perl to program for the Web.

ANSWERS TO SELF-REVIEW EXERCISES

25.1 a) Hypertext Transfer. b) cgi-bin. c) -w. d) scalar variable, array, hash. e) Arrays. f) eq. g) logic (or rules). h) Server-side includes. i) foreach. j) \$_.

25.2 a) False. Documents containing server-side includes usually have a file extension of .shtml. b) True. c) False. The numerical equality operator is ==. d) True. e) False. The built-in binding operator is =~. f) False. Cookies do not have access to private information, such as e-mail addresses or private data, stored on the hard drive. g) False. A valid HTTP header might be: Content-type: text/html. h) True. i) False. \w also matches the underscore character. j) False. CGI is an interface, not a programming language.

EXERCISES

25.3 How can a Perl program determine the type of browser a Web client is using?

25.4 Describe how input from an HTML form is retrieved in a Perl program.

25.5 How does a Web browser determine how to handle or display incoming data?

25.6 What is the terminology for a command that is embedded in an HTML document and parsed by a server prior to being sent?

25.7 Write a Perl program named `ex25_07.pl` that creates a scalar variable `$states` with the value "Mississippi Alabama Texas Massachusetts Kansas". Using only the techniques discussed in this chapter, write a program that does the following:

- a) Search for a word in scalar `$states` that ends in `xas`. Store this word in element 0 of an array named `@statesArray`.
- b) Search for a word in `$states` that begins with `k` and ends in `s`. Perform a case-insensitive comparison. Store this word in element 1 of `@statesArray`.
- c) Search for a word in `$states` that begins with `M` and ends in `s`. Store this in element 2 of the array.
- d) Search for a word in `$states` that ends in `a`. Store this word in element 3 of the array.
- e) Search for a word in `$states` at the beginning of the string that starts with `M`. Store this word in element 4 of the array.
- f) Output the array `@statesArray` to the screen.

25.8 In this chapter, we have presented CGI environment variables. Develop a program that determines whether the client is using Internet Explorer. If so, determine the version number, and send this information back to the client.

25.9 Modify the programs and documents in Figs. 25.12 and 25.13 to save information sent to the server in a text file.

25.10 Write a Perl program that tests whether an e-mail address is input correctly. A valid e-mail address contains a series of characters followed by the `@` character and a domain name.

25.11 Using CGI environment variables, write a program that logs the IP addresses (obtained with the `REMOTE_ADDR` CGI environment variable) that request information from the Web server.

25.12 Modify the programs in Figs. 25.19 and 25.20 so that there is another column in the resulting table. Each element in the column will be a button that, when clicked, will display a third Web page with a description of the current book. To do this in a straightforward manner, you should create a third program that will query the database for the book's description. This program will be called when one of the buttons is clicked.