Chapter 13

Perl and CGI Programming

We have seen some Perl programs in Chapter 8 where we introduced HTML forms and their connection to server-side support programs. But we had only a cursory coverage of Perl and its support for CGI programming.

Perl is a popular language to script CGI programs. Important Perl features are introduced to help you write better CGI programs. Perl variables, conditionals, I/O, loops, function definition, patterns, the taint mode, and more will be described. Enough information is included for you to use Perl comfortably for CGI programming.

The `CGI.pm` module and how it helps CGI programming is emphasized. Cookies and their application in multi-step user transactions are explained and demonstrated with examples.

Materials presented here give a concise and practical introduction to Perl CGI programming. With this background the reader can learn more about Perl and CGI with ease.

13.1 What is Perl

Perl is the *Practical Extraction and Report Language* and is freely available for downloading from the *Comprehensive Perl Archive Network* (`www.perl.com/CPAN/`). Perl is a portable, command line driven, interpreted programming/scripting language. Written properly, the same Perl code will run identically on UNIX, Windows, and Mac OS systems.

A Perl program (or script) consists of a sequence of commands and the source code file can be named arbitrarily but usually uses the `.pl` suffix. A Perl interpreter reads the source file and executes the commands in the order given. You may use any text editor to create Perl scripts. These scripts will work on any platform where the Perl interpreter has been installed.

The Perl scripting language is usually used in the following applications areas:

- Web CGI programming
- DOS and UNIX shell command scripts
- Text input parsing
- Report generation
- Text file transformations, conversions

Although Perl is not Web specific, our coverage of Perl is focused on the application of Perl in Web CGI programming.

13.2 A Brief History of Perl

Perl was created in the UNIX tradition of open source software. Perl 1.0 was released 18 December 1987 (the Perl birthday) by Larry Hall with the following description:

> Perl is a interpreted language optimized for scanning arbi- trary text files, extracting information from those text files, and printing reports based on that information. It's also a good language for many system management tasks. The language is intended to be practical (easy to use, efficient, complete) rather than beautiful (tiny, elegant, minimal). It combines (in the author's opinion, anyway) some of the best features of C, sed, awk, and sh, so people familiar with those languages should have little difficulty with it. (Language historians will also note

> some vestiges of csh, Pascal, and even BASIC—PLUS.) Expression syntax corresponds quite closely to C expression syntax. If you have a problem that would ordinarily use sed or awk or sh, but it exceeds their capabilities or must run a little faster, and you don't want to write the silly thing in C, then perl may be for you. ...

In 1989 Perl 3.0 was released and distributed for the first time under the *GNU Public License* with its now well-known *copy left* philosophy. Released in 1994 was Perl 5.0, a complete rewrite of Perl adding objects and a modular organization. The modular structure makes it easy for everyone to develop *Perl modules* to extend the functionalities of Perl. CGI.pm (CGI Perl Module) is just such a library (1995 by Lincoln Stein). This module makes CGI programming in Perl much easier and more powerful.

Many other Perl modules have been created. The CPAN (Comprehensive Perl Archive Network, `www.cpan.org`) was established to store and distribute Perl and Perl related software.

Because of its text processing ease, wide availability (it runs on all major platforms), and CGI programming abilities, Perl has became one of the most popular languages for CGI programming.

13.3 Perl Programming ABC

To create a Perl program, you may simply use your favorite text editor. The very first line, before any other characters in the source code file, indicates the command to invoke the Perl interpreter. For example,

```
#!/usr/local/bin/perl
```

It indicates the location of the Perl interpreter which will execute the rest of the file. This line can be different on different computers because the **perl** command can be installed anywhere on the hard disk. To move a Perl program to another computer, make sure the

first line is adjusted to reflect the installation location of **perl**. This line may also specify any options, such as `-T` (*taint mode*, Section 13.18), for the **perl** command.

The Perl program file must be executable. On UNIX do

chmod `a+rx` *program_name*

On Windows, run Perl programs from the MS-DOS prompt. As a Web CGI program, a Perl script must be placed in special `cgi-bin` directories configured by the Web server.
In a Perl script:

- Comments starts with the `#` character and continues to the end of the line.
- Each Perl statement ends with a semicolon (`;`).
- Statements are executed sequentially.
- The statement `exit(0);` (`exit(1);`) terminates the program normally (abnormally).

To run a Perl program use either one of

program_name *arg1 arg2 ...*
perl *program_name arg1 arg2 ...*

As a CGI script, it must be placed in the Web server's `cgi-bin`.

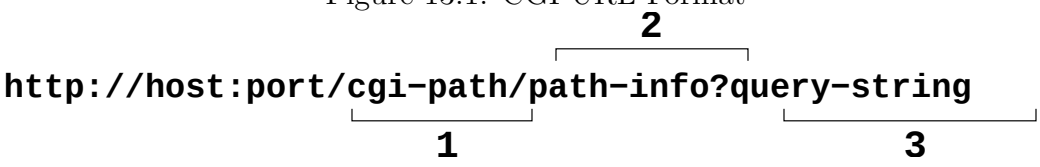
Here is a short Perl program (Ex: **PerlCmdLine**) that simply displays the command-line arguments:

```
#!/usr/bin/perl
###  displaying the command line              ## (a)

print "@ARGV\n";                               ## (b)
print "First arg: $ARGV[0]\n";                 ## (c)
print "Second arg: $ARGV[1]\n";
print "Third arg: $ARGV[2]\n";                 ## (d)
```

The mandatory first line is usually followed by one or more comments (line `a`) documenting the purpose, usage, author, and other key information for the program.

Figure 13.1: CGI URL Format



The first statement (line `b`) calls the `print` function with a string that is the value of the built-in array variable `@ARGV` followed by the NEWLINE character (`\n`). The `print` function outputs strings to *standard output* or to specified destinations. The standard output is normally the terminal screen but becomes the Web server when the program executess under CGI. Array subscripting is used to display the first three command line arguments (lines `c-d`).

On a UNIX system, enter this program into the file `cmdline.pl` and do

chmod `a+rx cmdline.pl`

to make it executable. Then issue the command

./cmdline.pl `a b c d e`

to run the program which is in the current directory (`./`). You should see the display

```
a b c d e
First arg: a
Second arg: b
Third  arg: c
```

You can run Perl programs similarly under MS/DOS. Remeber to use the *backslash* `\` as directory separator.

Web CGI programs usually do not depend on command-line arguments and instead use form data sent via HTTP `POST` or `GET` requests. Or a CGI program can be invoked directly by a hyperlink URL independent of HTML forms. The general form of a CGI-invoking URL is shown in Figure 13.1:

1. The *cgi-path* usually starts with a prefix `cgi-bin` which indicates a server-defined directory for placing CGI programs. The remainder is a path, relative to the `cgi-bin`, leading to the CGI program.

2. The optional *path-info* is a UNIX-style file pathname given by first placing a `/` after item 1. This value is transmitted to the CGI program via the `PATH_INFO` environment variable.

3. The optional *query-string* is given after first placing a question mark. It is a url-encoded string. If *query-string* contains no `=` in it, then it is transmitted to the CGI program as command line arguments. Otherwise, it is becomes the value of the `QUERY_STIRG` environment variable.

Accessing a CGI program directly via such a URL is supported by the HTTP GET query.

Let's take a closer look at Perl before returning to CGI programming.

13.4 Perl Varibles

Perl provides three types of variables: *scalar*, *array* (list), and *association array* (hash).

Scalars

A scalar variable has a `$` prefix and can take on any string or numerical values. For example,

```
$var = 'a string';        ## a quoted string
$n = length $var;         ## is 8
$x = 12;
$abc = "$var$x";          ## a string12  (A)
```

Characters enclosed in single quotes are taken literally while variables are meaningful inside double quotes (line `A`).

Arrays

The Perl array varible uses the `@` prefix. For example (Ex: **PerlArray**),

```
#!/usr/bin/perl
```

```
@arr = ("aa", "bb", "cc", "dd");   ## creating an array
print "$arr[0]\n";                 ## first array element is aa  (B)
$arr[2]=7;                         ## third element set to 7     (C)
$m = $#arr;                        ## 3, last index of @arr      (D)
$n = @arr;                         ## n is 4 length of @arr      (E)
print "@arr\n";                    ## aa bb 7 dd                 (F)

push(@arr, "xyz");                 ## put on end of array        (G)
print "@arr\n";                    ## aa bb 7 dd xyz
$last = pop(@arr);                 ## pop off end of array       (H)
print "@arr\n";                    ## aa bb 7 dd
```

Note we use the scalar notation to retrieve or set values on an array using indexing (lines `B-C`). The special prefix `$#` returns the index of the last array element (line `D`) and `-1` if the array has no elements. Hence, `$#arr+1` is the array length. Assigning an array to a scalar produces its length (line `E`). Displaying the entire array is as easy as printing it (line `F`).

Use the Perl built-in function `push` to append a new element to the end of the array (line `G`) and the `pop` function to remove the last element from the array (line `H`). The function `shift` (`unshift`) deletes (adds) an element at the beginning of an array.

Executing this program produces the following output

```
aa
aa bb 7 dd
aa bb 7 dd xyz
aa bb 7 dd
```

Association Arrays

An association array, also known as a *hash array*, is an array with even number of elements. Elements come in pairs, a *key* and a *value*. You can create an hash arrays with the notation:

`(` *key1* `=>` *value1*`,` *key2* `=>` *value2*`,` ... `)`

The keys serve as symbolic indices for the corresponding values on the association array.

Perl association array variables use the `%` prefix. For example (Ex: **PerlAsso**),

```
%asso = ( "a" => 7, "b" => 11 );   ##                (1)
print "$asso{'a'}\n";              ## displays 7     (2)
print "$asso{'b'}\n";              ## displays 11    (3)
print "@asso{'a', 'b'}\n";         ## displays 7, 11 (4)
```

The symbol `=>` (line 1) makes the association perfectly clear. But a comma works just as well.

```
%asso = ( "a", 7, "b", 11 );
```

To retrieve a value from an association array, use its key (lines 2-3). Note the `$` prefix is used with the key enclosed in curly braces (`{}`). To obtain a list of values from an association list, the `@` prefix can be used (line 4). Use a non-existent key or a value as a key (`$asso{'z'}`, `$asso{7}` for example) and you'll get an undefined value (`undef`).

Assign a new value with a similar assignment where the key may or may not already be on the association array:

```
$asso{'c'} = 13;
```

To remove key-value pairs from a hash, use calls like:

```
delete( $asso{'c'} );          (deletes one pair)
delete( @asso{'a', 'c'} );     (deletes a list of pairs)
```

The `keys` (`values`) function produces an array of keys (values) of a given hash:

```
@all_keys = keys( %asso )          ## ('a', 'b', 'c')
@all_values = values ( %asso )     ## (7, 11, 13)
```

You may turn a hash into a regular array (line 5) and use numerical indexing (lines 6-7):

```
@myarr = %asso;                    ##          (5)
print "$myarr[0]\n";               ## a        (6)
print "$myarr[1]\n";               ## 7
print "$myarr[2]\n";               ## b        (7)
```

The built-in association array `%ENV` contains all the environment variables transmitted to the Perl program. We have already seen how `%ENV` is used to access CGI related environment values (Section 8.18).

13.5 Arithmetic and String Operators

Perl arithmetic operators are similar to those in C:

```
$a = 1 + 2;     # adds 1 and 2 and stores in $a
$a = 3 - 4;     # subtracts 4 from 3 and stores in $a
$a = 5 * 6;     # multiplies 5 and 6
$a = 7 / 8;     # divides 7 by 8 to give 0.875
$a = 2 ** 8;    # raises 2 to the power 8 (not in C)
$a = 5 % 2;     # gets remainder of 5 divided by 2
++$a;           # increments $a and then returns it
$a++;           # returns $a and then increments it
--$a;           # decrements $a and then returns it
$a--;           # returns $a and then decrements it
```

Perl strings can be easily concatenated:

```
$a = $b . $c;   # Dot operator concatenates $b and $c
$a = $b x $c;   # x operator repeats $b $c times
```

The usual assignment operations are supported:

```
$a = $b;              # assigns $b to $a
$a += $b;             # same as $a = $a + $b
$a -= $b;             # same as $a = $a - $b
$a .= $b;             # same as $a = $a . $b

($a, $b) = ($c, $d);  # same as $a=$c; $b=$d;
($a, $b) = @food;     # same as $a = $food[0]; $b = $food[1]
```

13.6 True or False

In Perl Boolean values are scalar values interpreted in the Boolean context. The numerical zero, empty string, and undefined value are Boolean false. All other scalar values are Bookean true.

We test *logical conditions* with *relational operators*:

```
$a == $b      ## is $a numerically equal to $b?
              ##    also >, <, >=, <=
$a != $b      ## is $a numerically unequal to $b?

$a eq $b      ## are $a and $b equal as strings?
$a ne $b      ## are $a and $b unequal as strings?

($a && $b)    ## are both $a and $b true?
($a || $b)    ## is either $a or $b true?
!($a)         ## is $a false?
```

Also any non-zero number and any non-empty string is considered true. The number zero, zero by itself in a string, and the empty string are considered false.

13.7 Automatic Data Context

Perl makes programming easier by detecting the context within which a variable is used and automatically converts its value appropriately.

For example you can use strings as numbers and vice versa.

```
$str1 = "12.5";
$str2 = "2.5";
$sum = $str1 + $str2;          ## adding as numbers  (A)
print "$sum\n";                ## displaying         (B)
```

We used the strings as numbers on line `A` and the `$sum`, which is a number, as a string on line `B`.

The assignment `$len = @arr` uses `@arr` in a scalar context and turns its value to the length of the array. The automatic conversion of a hash to an array on line `5` is another example.

In a CGI program, the need to convert strings to numbers arises often. For example:

```
$total = param('payment');     ## a string representing a number
if ( $total > 50 )             ## number context
{   ...  }
```

The variable `$total` gets the string associated to the HTML form input `payment`. And we can compare it directly with the number `50`. Converting a string that does not represent a number to a number results in the number zero. Thus if `$str != 0` is true then `$str` is a valid nonzero number.

13.8 Conditional Statements

True or false tests are used in conditional statements that execute statements only when specified conditions are met:

```
if ( test )
{   ...   }
else         ## optional
{   ...   }
```

The `elseif` is also available as part of an `if` statement:

```
if ( test )
{   ...   }
elsif ( test2 ) ## note spelling
{   ...   }
else    ## Everything before was false
{   ...   }
```

13.9 Perl I/O

Standard I/O

In Perl, the *file handles* `STDIN`, `STDOUT`, and `STDERR` stand for the standard input (from keyboard), standard output (to screen), and standard error output (to screen, no buffering). When running as a CGI program, a Perl script receives posted form data by reading `STDIN`. The code

```
if ( $ENV{'REQUEST_METHOD'} eq "POST" )
{   read(STDIN, $input, $ENV{'CONTENT_LENGTH'}); }
elsif ( $ENV{'REQUEST_METHOD'} eq "GET" )
{   $input = $ENV{'QUERY_STRING'}; }
else
{   $input = ""; }
```

detects the request method and sets `$input` to the form data.

The notation <*input-source*> is handy to read lines from an input source. For example,

```
$line = <STDIN>;    /* or simply $line = <> */
```

Reads one line from standard input. Repeated execution of this statement will let you read line by line. The value of `$line` has the line termination character at the end. A handy function to remove any line terminator is the Perl built-in function `chomp`:

```
$str = chomp($line);
```

File I/O

The following program opens a file on the local file system for input, reads it, prints the data read to standard output, and closes the input file (Ex: **PerlCat**).

```
$file = $ARGV[0];                                  # file name, a string
open(IN, $file) || die("can't open $file:$!");  # opens or fails
@lines = <IN>;                                     # reads into an array
close(IN);                                         # closes input file
print @lines;                                      # outputs the array
```

The file handle `IN` is set up by `open` to be used for I/O to the file. The function `die` prints a message before terminating the program. It is executed if `open` does not return true (failed). The special variable `$!` is the system error string.

Once the handle `IN` has been opened, you may use

```
read(IN, $input, 20});       # Reads 20 characters
$line = <IN>;                # Reads one line
@lines = <IN>;               # Reads whole file into array
```

Table 13.1: File Checks

Check	Meaning	Check	Meaning
`if ( (-f $f) )`	is a plain file	`if ( (-d $f) )`	is a directory
`if ( (-r $f) )`	is readable	`if ( (-w $f) )`	is writable
`if ( (-x $f) )`	is executable	`if ( (-T $f) )`	is a text file
`if ( (-e $f) )`	exists	`if ( (-z $f) )`	is empty (zero size)
`if ( (-s $f) )`	is file size not 0	`if ( (-l $f) )`	is a symbolic link

to read in a specific number of characters, to read in one line, or to read in the whole file.

To open a file for output, use one of

```
open(OUT, ">$file");          # opens file for output
open(OUT, ">>$file");         # opens file for appending
```

Now you can sent output to the file with

```
print OUT "This line goes to the file.\n";
```

Again you may close the output with

```
close(OUT);                   # closes the output file
```

Let `$f` be a file name or file handle. You can use the checks listed in Table 13.1 before opening it for reading or writing.

Inter-process I/O

From a Perl program, you can execute any shell-level command (as another process on the same computer) and obtain its output with

`$result = ` *`command string`*

where the *command string* is enclosed in BACKQUOTES (`). For example,

```
$files = `ls -l`;
```

You can open another process for reading or writing. For example,

```
open(MAIL, "| /usr/sbin/sendmail") || die("fork failed") ;
```

gives `MAIL` for writing to the `sendmail` process. We have seen such uses in Section 8.17.

Whereas, the code

```
open(RESULT, "| ls -l") || die("fork failed") ;
```

allows you to read the results from `ls -l`.

13.10 Perl Iterations

An iteration is the repeated execution of a set of statements. Powerful programs often require iterations to perform tasks. Perl iteration constructs include: `foreach`, `while`, `do ... while`, and `for`.

The `foreach` Loop

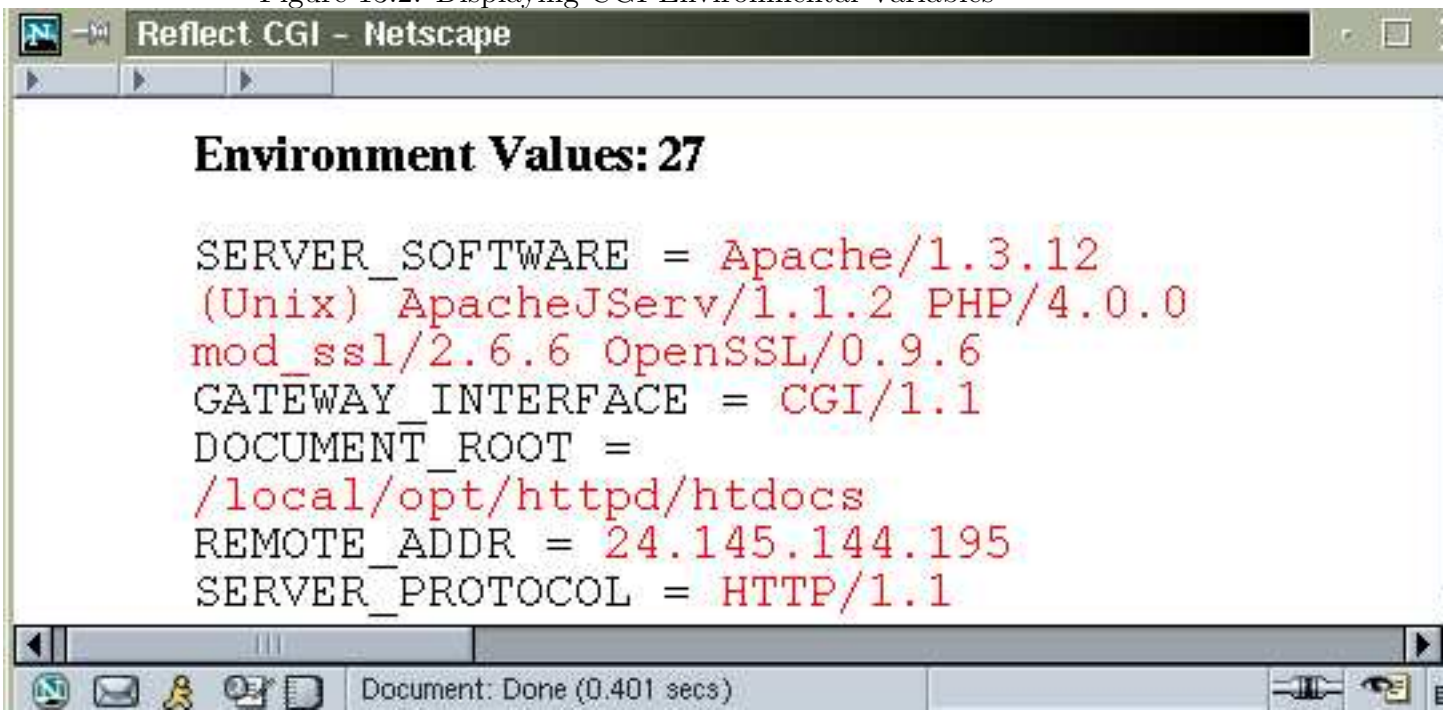
The syntax is:

```
foreach $var ( @array )
{   statements   }
```

In a CGI program, we can display the name and value of each environmental variable (Figure 13.2) with this iteration:

```
@keys = keys(%ENV);
$n = @keys;
print "<h2>Environment Values: $n</h2>";
print "<p style=\"font-size: larger\"><code>";
foreach  $key ( @keys )
{   print "$key = <span style=\"color: #c33\">
       $ENV{$key}</span><br />";
}
print "</code></p>";
```

Figure 13.2: Displaying CGI Environmental Variables



You can list the names of all `.html` files with the following:

```
print "<ul>";
foreach $file ( <*.html> )      ## file name matching
{    print "<li>$file</li>";
}
print "</ul>";
```

The notation `<*.html>` allows UNIX shell-like file matching.

The `while` Loop

The `while` iteration has the general form:

```
while ( test condition )
{     statements
}
```

The `while` loop tests a condition and, if true, executes a set of statements. The cycle is repeated until the test condition becomes false. If the condition is false in the beginning, the body of `while` may not be executed even once.

Let `amounts.txt` be a file with one amount figure per line. This `while` look computes the total:

```
$sum = 0;
open(IN, "amounts.txt");        # opens file for reading
while ( defined($amt = <IN>) )  # reads one line   (A)
{   $sum += $amt;  }
print "$amt\n";
```

The condition (line `A`) becomes false only after all lines are read from the file.

An easy to iterate over an associative array is

```
while ( ($person, $grade) = each(%grades) )
{    ...    }
```

The Perl function `each` returns the next two-element list of a hash for each iteration.

The `do-while` Loop

The `do-while` is very similar to `while` but it executes the enclosed statements before checking the condition for loop continuation:

```
do
{
      statements


} while (  condition );
```

The `for` Loop

The `while`, `do-while` and `for` loops all follow the C language. Here is an example `for` loop:

```
for ($i = 0; $i < 10; ++$i)      # starts with $i = 0
                                 # continues while $i < 10
                                 # increments $i before repeating
{   print "$i\n";   }
```

13.11 Defining Functions

Most serious programs require the definition of functions that can be called from anywhere. You define functions with the `sub` keyword. A function can be placed anywhere in your Perl source code file. Usually all functions are placed at the end of the file. For substantial programming, functions and objects can be placed in separate *packages* or *modules* and then imported into a program with the `use` statement (Section 13.19).

The general form of a function is:

`sub` *functionName*
`{`
    a sequence of statements
`}`

A call to the above may take any of these forms:

functionName`();` `## no arg`
functionName`($a);` `## one arg`
functionName`($aa, $bb);` `## two args`
functionName`(@arr);` `## array elements as args`
functionName`($aa, @arr);` `## $aa and array elements as args`

A function gets in-coming arguments in the special array `@_`. In a function definition, the notations `$_[0]`, `$_[1]`, `$_[2]` are used to access the individual arguments.

Arguments, scalars, arrays and hashes, in a function call are passed to the receiving function as a *flat list*. Consider the function call

```
myfunc($total, @toys)
```

In `myfun`, `$_[0]` is `$x`; `$_[1]` is `$toys[0]`; `$_[2]` is `$toys[1]`; and so on. Furthermore, each `$_[i]` is a reference to the argument passed and modifying it will alter the data in the calling program.

To obtain a local copy of the passed arguments use for example

```
sub myfun
{   my($a, $b, $c) = @_;       // $a, $b, $c local to myfun
    ...                        // and have copies of passed data
}
```

Here `$a` gets a copy of `$_[0]`, `$b` a copy of `$_[1]` and so on.

Use `return` *value*`;` to return a value for a function. If a function returns without executing a `return`, then the value is that of the last statement.

Sometimes you need to include arrays and hashes in a function call *un-flattened*. This can be done by passing *references* to the arrays and hashes. In general, a *reference* is a symbol that leads to the construct to which it refers (like a pointer). References are scalars and are passed in function calls as such.

References are not hard to understand. The following points will help.

- Put a '\' in front of a variable to obtain a reference: `$ref_x = \$`*x*, `ref_x = \@`*x*, or `ref_x = \%`*x*.
- Now `$ref_x` is a reference and can be used just like the symbol *x* to which it refers.
- The notation `$$ref_x` is the same as `$x`, `@$ref_x` is the same as `@x`, and `%$ref_x` is the same as `%x`.

The following Perl program (Ex: **PerlRef**) shows a scalar `$x`, an array `@arr` and a hash `%asso`, how their references are obtained, and used.

```
#!/usr/bin/perl
```

```
my $x = 3.1416, @arr = ("a", "b");
my %asso = ("one" => 7, "two" => 11);

my $ref_x = \$x;              // three references
my $ref_foo = \@arr;
my $ref_bar = \%asso;

// using references
print "$$ref_x\n";            // 3.1416
print "$$ref_foo[1]\n";       // b
print "$$ref_bar{'two'}\n";   // 11
```

When references are passed in function calls, they can be used in the called function exactly the same way.

Local Variables in Functions

In Perl, all variables are global within its module (source code file), unless declared local. In a subroutine local variables are declared with either the keywords `my` or `local`.

- `local(var1, var2, ...);` (dynamic nesting)
- `my(var1, var2, ...);` (static lexical scoping)

A variable declared by `my` is known only within the function or source code file, in the same sense as local variables in C, C++, or Java. A variable declared by `local` is known within the function and other function it calls at run-time, in the same sense as `prog` variables in Common LISP. For most purposes, the `my` declaration will suffice.

As an example, let's write a function `htmlBegin`. This can be a handy CGI utility that sends out the leading part of the HTML code for an HTTP response (Ex: **HtmlBegin**).

The function receives two arguments: the name of a page title and a filenname `$frontfile`. The `$frontfile` is a partial HTML template that can be customized for the look and feel of a particular website.

```
sub htmlBegin
{    my $title=$_[0];          ##  page title
```

```
    my $frontfile=$_[1];     ##  HTML template
    my $line;
    print "Content-type: text/html\r\n\r\n";
    if ( defined($frontfile) )
    {   open(IN, $frontfile) || die "Can't open $frontfile";
        while ( $line=<IN> )
        {  if ( $line =~ /XYZZZ/ )
           {   $line =~ s/XYZZZ/$title/;  }             ## (1)
           if  ( $line =~ /XHTMLFRONT/ )
           {   $line =~ s/XHTMLFRONT/$xhtml_front/;  }  ## (2)
           print  $line;
        }
        close(IN);
    }
    else
    {   print ( "$xhtml_front"                          ## (3)
                . "<head><title>$title</title>"
                . "</head><body>\n" );
    }
}
```

The variable `$xhtml_front` is set to the first lines needed for an XHTML file earlier in this program.

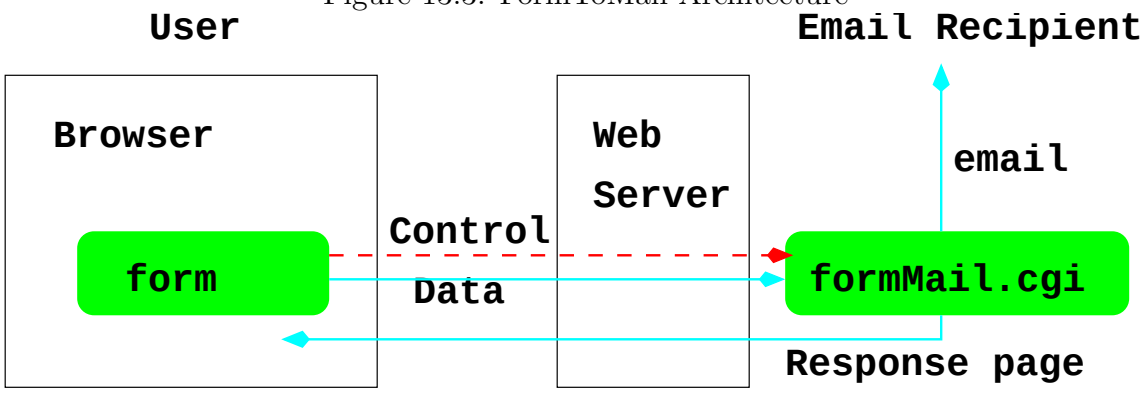
A sample call to this function is

```
htmlBegin("Request Confirmation", "webtong.front");
```

where the page title and an "HTML template" are supplied. The file `webtong.front`

```
XHTMLFRONT
<head><title>Webtong - XYZZZ</title>
<base href="http://www.webtong.com/feedback.html" />
<link rel="stylesheet" type="text/css" href="webtong.css" />
</head><body>
<table class="layout" border="0" cellspacing="0"
       cellpadding="0" width="100%">
<!-- top banner begin -->
   ...
<!-- top banner end -->
<h2>XYZZZ</h2></tr><tr>
<!-- left nav bar begin -->
```

Figure 13.3: FormToMail Architecture




```
   ...
<!-- left nav bar end -->

<td class="content">
<!-- CONTENT BEGIN -->
```

is an HTML fragment for the style and layout of a website with `XYZZZ` and `XHTMLFRONT` as *parts to be replaced* by the `htmlBegin` function. Perl pattern matching is used (lines `1-2`) to make the replacement. See Section 13.13 for a description of Perl patterns.

In case `$frontfile` is not defined a generic HTML opening is used (line `3`).

13.12 A Form-to-Email Program

For many websites, forms are used to collect information for off-line processing. Such forms can be supported by a well-designed server-side program that takes the form-collected data and sends email to designated persons. Furthermore, such a program can be made to serve forms located on various authorized sites. Hidden fields in the form can be used to customize and control the behavior of the program to suit diverse needs (Ex: **FormToEmail**). As early as 1995, such a CGI program was created and placed on the Web by Matt Wright. The **FormToMail** program described here has many new features and is also simpler because it uses the Perl CGI module. Figure 13.3 shows the FormToMail architecture. The form-supplied configuration parameters (dashed arrow) controls how the CGI program works. The

form-supplied email content (solid arrow) is sent to the target recipient and reflected in the confirmation response page.

Program Configuration

Our CGI program (Ex: **FormToMail**) begins with the customization part

```
#!/usr/bin/perl
### formMail.cgi
use CGI qw(:standard);                  ## uses CGI
my $formid, $front, $back, $recipient;  ## global vars

#### Customization begin
$mailprog = '/usr/lib/sendmail -t';     ## location of sendmail

@referrers = ();                        ## (1)
#### Customization end
```

The variable `mailprog` indicates the location of the *sendmail* command used for sending email. The array `@referrers` (line `1`) lists all domains (or domain prefixes) whose website may deploy forms to access this program. Example settings are,

```
my @referrers = ("symbolicnet.org", "131.123.35.90");
my @referrers = ('sofpower.com', 'cs.kent.edu', 'rooster.localdomain');
```

To avoid illegitimate use of the email program, we insist on forms on known hosts (the referrers) and requests via the `POST` method. Checking is done by the function `checkReferer`:

```
sub checkReferer
{   if ( $ENV{'REQUEST_METHOD'} eq "GET" ) // disallowed
    {   error('get request not allowed');   }
    my $url= $ENV{'HTTP_REFERER'};                        # (2)
    if ( $url )
    {  foreach $referrer (@referrers)
       {   if ($url =~ m|https?://([^/0-9]*)$referrer|i) # (3)
           {  return; }
       }
    }
    error('bad_referrer');                                # (4)
}
```

If the URL given by the environment variable `HTTP_REFERER` (line 2) does not contain (line 3) any domain or domain prefix listed on `@referrer`, an error results (line 4). The *pattern matching* used on line 3 is discussed in Section 13.13.

The `Config` associative array lists keys and some default values used in the `formMail.cgi` program. Values for these keys can be supplied by input fields, in the form.

```
## Configuration values
my %Config=('formid' => '',  'sender_name' => '',
            'email' => '',   'subject' => 'Form Email',
            'redirect' => '', 'sort' => '',
            'print_blank_fields' => '',
            'page_title' => 'Thank You',
            'required' => 'formid,email,sender_name');   # (5)
```

Form-supplied values are required for the first three keys: `formid` identifies the HTML form, `sender_name` give the name who filled out the form, and `email` is the senders email address. The variable `required` lists form keys whose values must be supplied. Its initial value lists three items (line 5). But the submitting form can name other required form entries. The email `subject` and reply `page_title` default to `Form Email` and `Thank You` respectively.

The form can also elect to redirect to a different response page, to request ordering of form entries in the response and in the email listing, and to list or ignore empty form fields.

Each valid *formid* has a recipient list `$Recipient{$formid}` and an optional emall cc list defined by the configuration association lists

```
my %Recipient=('webtong_hosting' => 'sales@webtong.com',  # (6)
               'webtong' => 'info@webtong.com',
               'wdp_MailForm' => 'test');

my %Cc = ( 'webtong_hosting' => 'DomainMaster@webtong.com' );
```

Thus, the program works only for predefined *formid*s (three here in line 6) and hard-coded recipients.

Optionally, each *formid* can also define two files

```
?$Config{'formid'}.front
$Config{'formid'}.back
```

used to build a customized response page for a particular form.

Let's take a look at the HTML form before returning to the rest of the Perl program.

Form for FormToMail

A Web hosting form, for example, may supply these values by hidden form fields:

```
<form method="post" action="/cgi-bin/formMail.cgi">
<input type="hidden" name="formid" value="webtong_hosting" />
<input type="hidden" name="subject" value="Web Hosting" />
<input type="hidden" name="page_title"
       value="Hosting Request Received" />
<input type="hidden" name="required"
       value="domain,phone,email,sender_name" />

...

</form>
```

Other values such as `email` and `sender_name` are input fields to be filled by the end user.

The form must be placed on the correct referrer site and indicate the correct formid to work. An email will only be sent to recipients prescribed in the CGI program.

The CGI Program

The program `formMail.cgi` follows a few simple steps to do the job:

```
checkReferer();       # checks referring URL
$date = getDate();    # retrieves current date
formData();           # obtains data sent from form
checkData();          # checks data for required fields etc.
response();           # returns response or redirects
sendMail();           # sends email
exit(0);              # terminates program
```

Calling `getDate()` sets up a well-formatted date and time string used in the email and response page as a timestamp.

The `formData()` function retrieves the form-supplied data and sets values in the `Config` and `Form` associative arrays for later use. The call `param()` (line `A`) gives all keys sent by the form

```
sub formData
{   my ($name, $value);
    foreach $name ( param() )    # for each name-value pair   (A)
    {   $value = param($name);
        if (defined($Config{$name}))  ## set Config values
        {  if ($name eq 'required')
           {   $Config{$name} = $Config{$name} . "," . $value;  }
           else
           {   $Config{$name} = $value;  }
           if ($name eq 'email' || $name eq 'sender_name')  # (B)
           {    push(@Field_Order,$name);
                $Form{$name} = $value;
           }
        }
        else                          ## set Form values
        {  if ($Form{$name} && $value)
           {
               $Form{$name} = "$Form{$name}, $value";
           }
           elsif ($value)
           {   push(@Field_Order,$name);
               $Form{$name} = $value;
           }
        }
    }
    ## removes white spaces and obtains required fields
    $Config{'required'} =~ s/(\s+|\n)?,(\s+|\n)?/,/g;       # (C)
    $Config{'required'} =~ s/(\s+)?\n+(\s+)?//g;
    @Required = split(/,/,$Config{'required'});             # (D)
}
```

Each form value is obtained and stored in either the `Config` or the `Form` associative array. The `sender_name` and `email` are set in both (line `B`). The required `Form` and `Config` values are set up in the `@Required` array (line`C-D`). The `checkData` function sets up global variables (lines `E-F`) and uses `@Required` to check for all required fields.

```
sub checkData
{   my ($require, @error, $formid);
    $formid = $Config{'formid'};         ## formid            (E)
    $recipient=$Recipient{$formid};      ## mail recipient
    if ( (-e "$formid.front") && (-e "$formid.back") )
    {  $front="$formid.front";   ## response front file
       $back ="$formid.back";    ## response back file        (F)
    }
    if (!$recipient) { error('no_recipient') }
    foreach $require (@Required)
    {   # email address must be valid
        if ($require eq 'email' )
        {   if ( !checkEmail($Config{$require}))
            { push(@error,$require); }
        }

        # check required config values
        elsif (defined($Config{$require}))
        {   if (!$Config{$require})
            { push(@error,$require); }
        }
        # check required form data
        elsif (!$Form{$require})
        { push(@error,$require); }
    }
    # If error
    if (@error) { error('missing_fields', @error) }
}
```

After user input has been received and checked, the program proceeds to send email and produce an HTTP response. The function `sendMail` sends email:

```
sub sendMail
{   if ( $recipient eq "test" ) { return; }
    open(MAIL,"|$mailprog")               # opens mail program  (G)
        || die("open $mailprog failed") ;
    mailHeaders();                        # email headers       (H)
    print MAIL "This is a message from the " .
               "$Config{'site'}.  It was submitted by\n";
    print MAIL "$Config{'sender_name'} " .
               "($Config{'email'}) on $date\n";
    print MAIL "-" x 75 . "\n\n";         #                     (I)
```

```
    if ($Config{'sort'} eq 'alphabetic')   # alphabetical order  (J)
    {   mailFields(sort keys %Form); }
    elsif( getOrder() )                   # specific order
    {   mailFields(@sorted_fields); }
    else                                   # no ordering
    {   mailFields(@Field_Order); }
    print MAIL "-" x 75 . "\n\n";
    close (MAIL);
}
```

It opens the mail program (line G) and sends output to it with the `print` function. The `mailHeaders` function (line H) sends the `To:`, `From:`, `Subject:`, and `Cc:` headers. A confirmation copy is sent to the form sender at the given email address. The `"-" x 75` is Perl notation to get 75 consecutive dashes and the Perl operator `.` concatenates strings (line I). The form fields are sent on separate lines in three different possible orderings (line J) controlled by the `Config{'sort'}` value.

The `formMail` program also uses the `htmlBegin` (Section 13.11) and a similar `htmlEnd` function to allow easy site integration.

```
sub htmlBegin
{    my $title=$_[0];
     my $ln;
     print "Content-type: text/html\n\n";
     if ( defined($front) && $front ne ""
          && open(FF, $front) )             ## front file
     {   while ( $ln=<FF> )
         {  if ( $ln =~ /XYZZZ/ )
            {   $ln =~ s/XYZZZ/$title/;  }  ## page title
            print  $ln;
         }
         print "<h2>$title</h2>";           ## %%NEW XXXxxx
         close(FF);
     }
     else
     {   print ("<html><head><title>$title</title>" .
                "</head><body bgcolor=white>\n");
     }
}
```

The function `htmlBegin` is called as follows

```
htmlBegin("$Config{'page_title'}");
```

The program also contains `getDate` and other functions, including several for email checking to make sure the email address of the user is well-formed. The complete `formMail.cgi` program is available in the example package.

13.13 Pattern Matching in Perl

Matching of string patterns is important in practice. This is especially true for both client-side javascript and Perl CGI programming. We have seen some patterns and pattern matching in Javascript (Section 9.10) already. This is a big boost for pattern matching in Perl because almost identical regular expression notations are used. In fact, Javascript has borrowed the Perl regular expression syntax an semantics almost without change because Perl has excellent support for pattern matching and related manipulations.

The Perl relational operators `=~` (match) and `!~` (non-match) are used for pattern matching. In Perl, patterns are specified as *extended regular expressions* similar to that used for Javascript and the UNIX **egrep** utility.

The following are some simple matching examples involving the string `$line`:

```
if ( $line =~ /kent/ )   ## contains kent

if ( $line =~ /Kent/ )   ## contains Kent

if ( $line =~ /Kent/i )  ## contains kent, ignoring case

if ( $line =~ /^Kent/ )  ## Kent at beginning of line

if ( $line =~ /Kent$/ )  ## Kent at end of line
```

Note patterns are given inside `/`'s (the pattern delimiter). If the pattern contains `/` then it is convenient to use a leading `m` which allows you to use any non-alphanumeric character as the pattern delimiter:

```
if ( $url =~ m|http://| )              (A)
```

Or you can use \ to escape the / in the pattern:

```
if ( $url =~ /http:\/\// )             (Same as A)
```

Table 9.2 illustrates use of patterns.

The following is a pattern we applied in Ex: **FormToMail**.

```
if ($url =~ m|https?://([^/0-9]*)$referrer|i)          (B)
```

The pattern starts with `http`, then an optional `s`, followed by the three characters `://`, followed by zero or more characters each not a digit or the `/`, ending at the `$referrer` string. The `i` at the end (line `B`) is a *match option* indicating *case-insensitive* matching.

Special characters in Perl patterns include:

`\n` (A newline)
`\t` (A tab)
`\w` (Any alphanumeric (word) character, same as `[a-zA-Z0-9_]`)
`\W` (Any non-word character, same as `[^a-zA-Z0-9_]`)
`\d` (Any digit. The same as `[0-9]`)
`\D` (Any non-digit. The same as `[^0-9]`)
`\s` (Any whitespace character: space, `\t`, `\n`, etc)
`\S` (Any non-whitespace character)
`\b` (A word boundary, outside `[]` only)
`\B` (No word boundary)
`\x` (escapes x)

13.14 Substitutions

Often we look for a pattern in a string for the purpose of replacing it. This can be done easily with the string matching operator `=~`:

- `$line =~ s/http/HTTP/;`—replaces first occurrence of `http` in `$line` by `HTTP`.
- `$line =~ s/http/HTTP/g;`—replaces all occurrences *globally* in `$line`.
- `$line =~ s/http/HTTP/gi;`—ignores case in global matching.
- `$line =~ s/`*pattern*`/`*cmd*`/e;`—uses the replacement string obtained by executing *cmd*.

Pattern Matching Memory

You can store the parts of the string that matches parts of a pattern and use these matched parts in substitutions or for other purposes.

You use parenthesis in a pattern to call for memory. Matched strings get remembered in the variables `$1`,...,`$9`. These strings can also be used in the same regular expression (or substitution) by using the alternative codes `\1`,...,`\9`.

For example:

- `/(\w)\1/` matches repeated words
- `/(['"]).*\1/` words within single or double quotes
- `s/(Art)/$1s/` Art becomes Arts

Perl automatically uses these built-in variables `$‘` (string before the matched string), `$&` (the matched string), `$’` (string after the matched string), to remember the three parts for a match. Thus the following works:

`s/Art/$&s/` (Art becomes Arts)

Let’s look at some substitution Examples. If `$line` is

```
$line = 'BB    MM   GG   LL';
```

Then

```
$line =~ s/(\S+)\s+(\S+)/\2 \1/;   ## switches BB with MM
$line =~ s/s+(\S+)\s+(\S+)/\2 \1/; ## switches BB with GG
$line =~ s/(\S+)\s+(\S+)/$2 $1/g;  ## switches first two,
                                   ##   then next two ...
```

The flag `g` matches the pattern repeatedly.

The flag `e` allows you to execute a Perl expression and use the result of that expression as the substitution string. The following statement

```
$form_data =~ s/%(..)/pack("C", hex($1))/ge;
```

decodes the hex codes in the URL-encoded `$form_data`. The built-in function `hex` turns a string into a hex number and the built-in function `pack` turns a hex number into a corresponding ASCII character.

13.15 Simple Page Search

Let's see how string manipulation, file I/O, function definition, and pattern matching can be put to use in a CGI program.

The website `SymbolicNet.org` has an email directory for people in *Symbolic Computation*, an area of research. The email directory page offers a simple page-search function (Ex: **PageSearch**) allowing users to enter a text string to obtain all email listings matching the given string. Figure 13.4 shows the form at the beginning of the email directory page (`listing.html`).

The HTML source code for the form is:

```
<p>To look for email entries in this page,
   please enter text to find.</p>
<form method="post"
      action="/cgi-bin/pagesearch.pl">            <!-- a -->
  <input type="hidden" name="page"
         value="/email/listing.html" />           <!-- b -->
  <p>Find text:<br />
  <input name="pattern" size="20" />  
  <input type="submit" value="Find All" /> </p>
</form>
```

Figure 13.4: Email Directory Search

The CGI program `pagesearch.pl` (line `a`) receives the location of the file to search via a hidden form field (line `b`).

The program `pagesearch.pl` performs these tasks:

- Opens a given email listing page
- Looks for the form supplied pattern in each line of the page
- Remembers all matching lines
- Outputs the count of lines matched followed by all matching lines
- Reports errors when something is wrong

This CGI program also sends the content length conforming to HTTP 1.1. Let's look at the source code of `pagesearch.pl`.

The `CGI.pm` module (line `1`) is used. The variable `$sn_root` (line `2`) is the file path of the document root for SymbolicNet. The string `$reply` is the search result and `$error` is for error messages (line `3`). The actual page to search (`$page`) is now retrieved from the value of the form parameter `page` (line `4`). If `$page` is not set, a message is appended to `$error` (line `5`).

The program proceeds to open the file to be searched (line `6`). If that file fails to open, a message is added to `$error` (line `7`). Note the use of the Perl operator `or` which can also

be written as `||`. If `open` fails, its value is `false` which causes the right operand of `or` to be evaluated. If `open` succeeds, its value is `true` and the right operand of `or` will not be evaluated.

```
#!/usr/bin/perl
##  search email listing
use CGI qw(:standard);                          ## (1)

my $sn_root="/home/httpd/htdocs";               ## (2)
my $reply="", $error = "", $file;               ## (3)
my $page = param('page');                       ## (4)

if ( $page eq "" )
{  $error .=
   "<p>Page to search not specified!</p>";      ## (5)
}
else
{  $file = $sn_root . $page;
   open(listing, "$file") or                    ## (6)
     $error .= "<p>Can not open $file!</p>";    ## (7)
}

$pt = param('pattern');                         ## (8)

if ( $pt eq "" )
{  $error .= "<p>You didn\'t " .                ## (9)
       "submit any text to find.</p>";
}

if ( $error )
{ errorReply($error); exit(1); }                ## (10)

### construct reply
outputFile("frontfile");                        ## (11)

### find matching entries
my $count = 0, $match="";
while ( defined($line=<listing>) )              ## (12)
{   if ( $line =~ /$pt/i )                      ## (13)
    {   $count++;                               ## (14)
        $match .= $line;                        ## (15)
    }
```

```
}
close(listing);  ## done with listing

$reply .= "<h3 style=\"margin-top: 16px\">
           Found $count entries matching
           <code>$pt</code>:</h3>";            ## (16)

if ( $count > 0 )
{   $reply .= "<ol> $match </ol>"; }           ## (17)

outputFile("backfile");                        ## (18)
sendPage($reply);                              ## (19)
exit(0);                                       ## (20)
```

With `listing` opened (line 6), we now obtain the pattern to match (line 8), handling the missing-pattern error in a similar way. If any error has been detected (line 10), the `errorReply` function is called to display the `$error` and the program terminates.

Detecting no error, the program begins to construct the `$reply` string. The `frontfile` is added to `$reply` first (line 11).

The `while` loop (line 12) reads each line (in the from of an `<li>` element) from `listing` and, if the line contains the pattern (line 13), increments the *match count* (line 14) and concatenates the line to the variable `$match` (line 15).

The number of matching entries found is appended to `$reply` (line 16) as a level-3 heading. And, if there are any matching entries, they are displayed in a numbered list (line 17).

The program terminates (line 20) after adding the `backfile` and sending the response page (line 19).

The functions used are at the end of the program The `front()` function puts out the `Content-type` header and two line terminators (line 20). Then calls `outputFile` (line 22) to send out the front part of the HTML code.

```
sub outputFile                                 ## (21)
{  my($ln, $f);
   $f = $_[0];                                 ## (22)
   open(FF, $f) ||
```

Figure 13.5: Matching Email Entries


```
      errMail("failed to open the file $f.");  ## (23)
   while ( $line=<FF> ) {  $reply .= "$line"; }
   close(FF);
}
```

The `outputFile` function (line 21) takes a filename argument (line 22). The notation `$_[0]` refers to the first argument passed to a function.

The `frontfile` and `backfile` are fixed HTML codes that put the CGI-generated content in the presentation style of the SymbolicNet site. Thus, the CGI response is integrated with the SymbolicNet site. A unified look and feel is important for both static and dynamic pages.

Code in `frontfile` and `backfile` usually contain hyperlinks relative to the location of the HTML form rather than the location of the CGI program. We can make those links work for the CGI reply page by simply adding to the `base` element

`<base href="`*URL-of-correct-location*`" />`

The `sendPage` function takes the page content argument and sents it out under HTTP 1.1 protocol. By sending the content length (line 24), the server and client can keep their connection alive for better networking efficiency.

Figure 13.6: Error Display


```
sub sendPage
{   my $content=$_[0];
    my $length=length($content);
    print "Content-type: text/html\r\n";
    print "Content-length: $length\r\n\r\n";              ## (24)
    print $content;
}

sub errorReply
{   my $msg=$_[0];
    outputFile("front");
    $reply .= "<h3 style=\"margin-top: 16px\">
               Error encountered:</h3> $msg";
    $reply .= "<p>Please go back and submit your
                  request again.</p>";
    outputFile("backfile");
    sendPage($reply);
}
```

The `errorReply` function (called on line `10`) displays `$error` and then terminates the program. Figure 13.6 shows a sample error page.

The `errMail` function is called (line `23`) when `frontfile` or `backfile` cannot be opened. This internal error won't happen if these two files have been placed in the `cgi-bin` together with `pagesearch.pl` with read access opened.

chmod `o+r frontfile backfile`

When called, the `errMail` function sends the internal error information to the webmaster of SymbolicNet.

```
sub errMail
{  my $mailer;
   $mailer = "/lib/sendmail -t";                          ## (25)
   $msg = $_[0];
   if ( open(MAIL, "| $mailer " ) )
   {  print MAIL "Subject: $ENV{'SCRIPT_NAME'} Error\n";
      print MAIL "To: $ENV{'SERVER_ADMIN'}\n";            ## (26)
      print MAIL "From: SymbolicNet\n";
      print MAIL "Script error: $msg";
      close (MAIL);
   }
   else
   {  print STDERR "mail failed\n"; }
   exit(1);
}
```

The **sendmail** command (line 25) is the most basic email program located at `/lib/sendmail` on most UNIX systems. The email address of the server administrator is obtained from the environment varialbe `SERVER_ADMIN` (line 26).

13.16 Perl Built-in Functions

Perl Documentation

In both Chapter 8 and this chapter, We have seen many built-in functions in action. Many more built-in functions are available in Perl, making it a powerful tool. You can find complete coverage of all functions in the Perl documentation. On UNIX systems the shell command

man perl

shows the Perl overview and the list of topics you can access with the **man** command. In particular

man `perlfunc` (Built-in functions)

man `perlsub` (Defining your own functions)

man `perlre` (Regular expressions)

an so on. On the Web, Perl documentation can be found easily at `www.perldoc.com`.

Generally, a build-in function takes either one scalar argument or an argument list. In the latter case, a function call looks like

functionName`(` *arg1*`,` *arg2*`, ... ,` *argn* `)`

Each argument can be scalar, array or hash. And the arguments are passed as a flat list as described in Section 13.11.

Let's describe a few more useful functions.

The function `join(`$*str*`,` $*list*`)` forms a single string by joining all the elements on *list* separated by the given string *str*. The function `substr(`$*str*`,` $*offset*`,` $*length*`)` obtains a substring of *str*. The last argument is optional. The function `index(`$*str*`,` $*substr*`)` returns a zero-based index of the position of *substr* in *str*.

The `tr` function performs character-by-character translation.

`$var =~ tr/`*chars*`/`*replacements*`/`

For example,

```
$var = "name=John+J+Doe";
$var =~ tr/+/ /;                (becomes name=John J Doe)
```

translates each `+` to a SPACE in `$var`. Whereas

```
$sentence =~ tr/abc/ABC/
```

replaces `a` by `A`, `b` by `B`, etc. The following replaces all lowercase characters by uppercase ones.

`$var =~ tr/a-z/A-Z/;` (becomes `NAME=JOHN+J+DOE`)

The `split` function is used to split a string up into parts. The parts are returned in an array. For example,

```
$info = "Caine:Michael:Actor:14, Leafy Drive";
@personal = split(/:/, $info);
```

then `@personal` becomes

```
("Caine", "Michael", "Actor", "14, Leafy Drive")
```

If `split` is given only one argument (a string), then it is split at white space characters.

The function `eval` can be given a string and it will execute that string as a Perl statement.

Numerical functions in Perl include `abs` (absolute value), `cos` (Cosine), `exp` (exponentiation), `hex` (hex value), `int`, `log` (natural logarithm), `oct` (octal value), `rand` (random number generation), `sin` (Sine), `sqrt` (square-root), and `srand` (sets seed for `rand()`).

13.17 Handling Passwords

Putting some of the functions to use, let's look at a Perl program `unixpw.pl`. This example (Ex: **UnixPW**) is a complete Perl program that produces an encrypted password using the UNIX `crypt` method. The shell-level command

unixpw.pl `catandmouse 9i`

produces the output

```
catandmouse --> 9iYuNF9nj2vm.
```

displaying the password in clear and encrypted text.

The `9i` is a *salt*, a random two-character string from the character set `[./0-9A-Za-z]`, to make the encrypted password more secure.

```
#!/usr/bin/perl
## unixpw.pl generates a Unix encrypted Password

use strict;

my $salt='';
my $encrypted='';
my $password='';
my $use = 'usage: unixpw password [salt]';
my @saltchars=('.', '/', 0..9, 'A'..'Z', 'a'..'z');
my $args=length(@ARGV);

if ( $args < 1 || $args > 2 )           ## (1)
{   print "$use\n";                     ## (2)
    exit;
}

$password=$ARGV[0];

if ( $args == 1 )                       ## (3)
{   $salt = join('',@saltchars[rand(64),
                rand(64)]);
}
else                                    ## (4)
{   $salt=$ARGV[1];
}

$encrypted=crypt($password,$salt);      ## (5)
print "$password --> $encrypted\n";
```

The number of command-line arguments are checked (line 1) and, if incorrect, the program usage is displayed (line 2). If only one argument is supplied (line 3), a salt value is generated randomly by calling the `rand` and `join` functions.

The password and salt is then passed to the `crypt` function to produce the encrypted password (line 5).

A password checking program can use

```
if ( crypt($password, $encrypted) eq $encrypted )
```

to check if a user-supplied `$password` is correct by using an `$encrypted` string from a password file.

13.18 Taint Mode

CGI programs sometimes use form data in shell-level or system-level commands. This presents a security problem if the user data is not well-formed or even intentionally set to break security of the server.

One good way to lock out security bugs in Perl code is to turn on *taint mode*. Taint mode causes a Perl script to automatically treat all user supplied input as tainted and dangerous unless the programmer explicitly OKs the data.

It is simple to turn on the taint mode. You simply add the `-T` option to the Perl interpreter.

```
#!/usr/bin/perl -T
```

In taint mode, Perl does not allow the use of tainted data in potentially unsafe operations (guarded operations). These include shell commands with BACK-QUOTEs (`` ` ``), any system or piped `open()` calls, opening files for output, deleting or renaming files. The first things Perl makes sure are that the command search `PATH` and command related values on the `ENV` are correct. In taint mode, the Perl program must explicitly set `$ENV{'PATH'}` and delete unneeded values on `%ENV`:

```
$ENV{'PATH'}='/usr/sbin:/sbin:/bin:/usr/bin';
delete @ENV{qw(IFS CDPATH ENV BASH_ENV)};
```

The `formMail` program will run under taint mode if you add the preceding two lines to the program.

A program must also check the correctness of user supplied data. Correct user data can be *untainted* before being used in guarded operations. For example,

```
$email = param('email');       ## email address from user
if ( checkEmail($email) )      ## check correctness
{   $email = untaint($email);  ## untainting
    sendMail($email);          ## function uses guarded operation
}
```

The function `checkEmail` is a Perl version of the email checking program described in Section 9.15).

The `untaint` function is simple:

```
sub untaint
{   $_[0] =~ /(.+)/;           ## pattern matching
    return $1;
}
```

Any result from pattern matching, `$1` here, becomes untainted and usable in guarded operations.

Let's look at a simple but complete example (Ex: **UnTaint**) of a taint-mode Perl script. It can serve as a guide for writing your own taint-mode scripts.

```
#!/usr/bin/perl -T
use CGI qw(:standard);

$ENV{'PATH'}='/usr/sbin:/sbin:/bin:/usr/bin';  ## (A)
delete @ENV{qw(IFS CDPATH ENV BASH_ENV)};      ## (B)

$file = noSpecial(param('filename'));          ## (C)
open($FH, ">$file");                           ## (D)
print $FH "a text line\n";
close($FH);

sub noSpecial                                  ## (E)
{  $_[0] =~ s/([^a-zA-Z0-9_@,.-])/\\$1/g;
   untaint($_[0]);
}

sub untaint
{   $_[0] =~ /(.+)/;
    return $1;
}
```

The program first takes care of `PATH` and `@ENV` (lines `A-B`). Then, it checks and untaints the file name supplied by the user before using it to create a new file (line `D`). If the file name is not untainted the guarded operation `open` will fail.

The function `noSpecial` adds the escape character `\` in front of any UNIX shell special characters and then untaints the argument string (line `E`).

See the security section of the Perl manual (`perlsec`) for more inforation.

13.19 The Perl CGI Module

`CGI.pm` is a *Perl module* that we have been using informally since Chapter 8. It is time to provide a description of Perl modules and an overview of the CGI module.

What is a Perl Module

Perl modules are packages of Perl functions to perform particular tasks. Module files are named *module_name*`.pm`. For example, `CGI.pm` is for writing CGI programs for the Web and `LWP.pm` is for making HTTP requests from Perl programs.

A module is a package that provides a separate name space where all its identifiers have the *module_name*`::` prefix when used from outside the module.

Standard modules, such as `CGI.pm`, come with the Perl distribution. Additional modules can be downloaded freely and new ones are being developed all the time. See

```
www.cpan.org/modules
```

for a listing and how to install/update Perl modules. The command

```
perl -MCPAN -e shell
```

loads Perl modules with a high-degree of automation.

Using Perl Modules

To use the functions and variables defined in a *module* you put one of the lines

`use` *`module`* (e.g. `use CGI`)

`use` *`module feature_list`* (e.g. `use CGI qw(:standard)`)

at the beginning of your Perl program. The first line imports all features of *module*. The 2nd line imports only the listed features. Usually, this means a number of functions and variables defined in *module* are now directly usable in your program. For example, `use CGI qw(:standard)` makes the `param` function available.

The `use` operator locates the required module on directories listed in the special variable `@INC` where Perl looks for headers and modules to include.

To load a module without importing its symbols, use

`use` *`module`* `()` (e.g. `use CGI ()`)

In this case, you can access features with the '`CGI::`' prefix (e.g. `CGI::param('email')`).

The CGI Module

The `CGI.pm` makes Web CGI programming easy. It can

- Receive and parse incoming form data
- Create forms with initial values derived from user input
- Generate simple HTML
- Support file upload, cookies, cascading style sheets, server push, and frames.

Some of these features are explained here. On UNIX systems, do **man** `CGI` to see the manual page for `CGI.pm`. You can find the latest version, as well as introduction and complete documentation, at

`stein.cshl.org/WWW/software/CGI/cgi_docs.html`

With CGI.pm, form data can be obtained easily with the param function.

```
$value = param('email');          ## value for email
@value = param('sports');         ## array of values

@names = param();                 ## all param names
foreach $name ( @names)
{   $value = param( $name );
    ...
}
```

A CGI program using CGI.pm can generate its own forms on-the-fly and easily make "submit-again" forms based on user-supplied values and flagged errors.

Such a CGI program follows this structure:

1. If there is no input data (if (! param())), a blank form to be submitted to the program itself is generated.

2. If there is form input, then the correctness of input is checked. A "submit-again" form is generated if errors are detected.

3. For correct input, the input data is processed and success is confirmed. The program may also invite a new request with a blank form.

HTML Code Generation

CGI.pm has a set of functions for HTML code generation. They produce strings that can be sent as output. These functions have obvious names and work as you might expect. Table 13.2 shows the general code generation call syntax that makes producing tags, attributes, and enclosed content easy.

For example (Ex: **CodeGen**), the code

```
print( h3('Survey'),                                  ## (1)
       start_form(-action=>"/cgi-bin/gen.pl"),        ## (2)
       span({-style=>'font-weight: bold'},
            "Your name: "),                           ## (3)
```

Table 13.2: Code Generation Call Syntax

Call	HTML code
p()	
p("text")	text
p({'-class'=>'fine'})	
p({'-class'=>'fine'}, "text")	text

```
    textfield('name'),                              ## (4)
    p("What's your favorite color? ",               ## (5)
       popup_menu(-name=>'color',
          -values=>['red','green','blue']),
     ),
    submit(-name=>"send", -value="Send"),           ## (6)
    end_form;                                       ## (7)
)
```

is a `print` call where the arguments are code generation functions. Lines in the generated code shown below are labeled to indicate how they are generated.

```
<h3>Survey</h3>                                          (from 1)
<form method="post" action="/cgi-bin/gen.pl"             (from 2)
      enctype="application/x-www-form-urlencoded">
  <span style="font-weight: bold">Your name: </span>     (from 3)
  <input type="text" name="name"  />                     (from 4)
<p>What's your favorite color?                           (from 5)
<select name="color">
  <option  value="red">red</option>
  <option  value="green">green</option>
  <option  value="blue">blue</option>
</select>
</p>                                                     (from 5)
<input type="submit" name="send" value="Send" />         (from 6)
</form>                                                  (from 7)
```

HTTP Header Generation

HTTP header generation is also done simply with `CGI.pm` using the `header` function. If it is called with no arguments

```
header()
```

it will return a `Date` header and the default content type header.

```
"Content-Type: text/html; charset=ISO-8859-1\n\n"
```

Use something like `header(-type=>img/gif)` to produce a different content type and something like `-charset=>'UTF-8'` if the page contains characters outside of Latin-1 (ISO 8859-1).

If `header` is called with one or more *-Name=>str* arguments, then the specified headers will be returned together with the appropriate default headers.

```
header(-Name=>str,          -Name2=>str2,
       ...)
```

Any underscore in the name part will become a hyphen (-) in the generated header name. For example,

```
print header(-Content_length=>"990", -charset=>'UTF-8');
```

outputs these lines

```
Date: Tue, 17 Sep 2004 02:22:28 GMT
Content-length: 2990
Content-Type: text/html; charset=UTF-8
<empty line>
```

13.20 Handling File Uploading

In Section 8.7 we have described the HTML form to upload a file. But we still need a server-side program that can handle the uploaded file. With `CGI.pm` the following simple code can retrieve the uploaded file and store it in a pre-arranged directory.

```
$FH = upload('file_name');}        (A)
binmode(FH); binmode(OUT);         (B)
while ( <$FH> ) {     print OUT;    }
```

The `CGI.pm` function `upload` (line `A`) when given the `name` attribute of the file input field (*file_name*) returns a file handle for reading the uploaded file. To store the file, we use binary mode I/O (line `B`).

The content type and length of the uploaded file can be retrieved by the calls

```
$type = uploadInfo($FH)->{'Content-Type'};
$len = uploadInfo($FH)->{'Content-Length'};
```

A file-upload hander will usually check the userid and password supplied to limit file uploading to authorized users.

13.21 Testing and Debugging `CGI.pm` Scripts

`CGI.pm` makes it easy to test and debug Perl scripts. You can employ the `-debug` feature

```
use CGI qw/:standard -debug/;
```

for testing and debugging. In this mode you can test your CGI scripts from the command line. You can supply, from the keyboard or a redirected file, name-value pairs in one line or on separate lines. The code

CGI_Script_Name `<` *data_in*

tests a CGI script with the prepared form data in the file *data_in*.

Also, the `Dump()` function call produces an HTML list of all the input form data.

For general Perl debugging you can run a perl program interactively with the debugging option:

perl `-d` *program_file arguments*

You can also run Perl interactively and experiment with Perl constructs using

perl `-de 7` (Runs Perl interactively)

The `e 7` simply gives something for Perl to execute at the beginning. This allows you to type in Perl statements and have them executed immediately. When you are done, type `q` to quit.

13.22 Session Control

HTTP is a protocol that treats each request-response transaction between a client and a server independently. Consequently, a server normally cannot tell if two requests are from the same client. Hence, a servlet usually handles each incoming request as *new business not related to anything before*. Thus, a CGI program does not remember anything from a previous request because there is no need. This model is fine for simple data retrieval such as getting weather reports, obtaining stock quotes, and listing movies from a local cinema.

But more complicated operations like airline reservation, shopping, credit card purchase, and membership services are difficult because they require a sequence of steps where latter steps depend on former ones. Using HTTP, multi-step operations require a sequence of transactions (*a session*), to complete. To perform multi-step operations we must find a way to track requests from the same end-user that belong to the same session. This is not quite possible for a server-side program to do under HTTP. But it is possible to do something close–*tracking all requests from the same client browser*. Such *session tracking* ability allows server-side agents such as CGI programs to give acceptable support to multi-step operations.

13.23 Sessions under HTTP

With HTTP, a server-side program does not have an easy way to tell the difference between a follow-up request and a new request. Thus, if you send in your billing information in a prior request then send in your shipping information via a subsequent request, the server-side program has no easy way to associate the shipping information with the earlier billing information. The lack of memory of prior requests is known as the *stateless* property of the HTTP protocol.

To maintain sessions, a server-side program must *maintain session state* by remembering the effects of prior requests, and be able to tell which new incoming request belongs to which session.

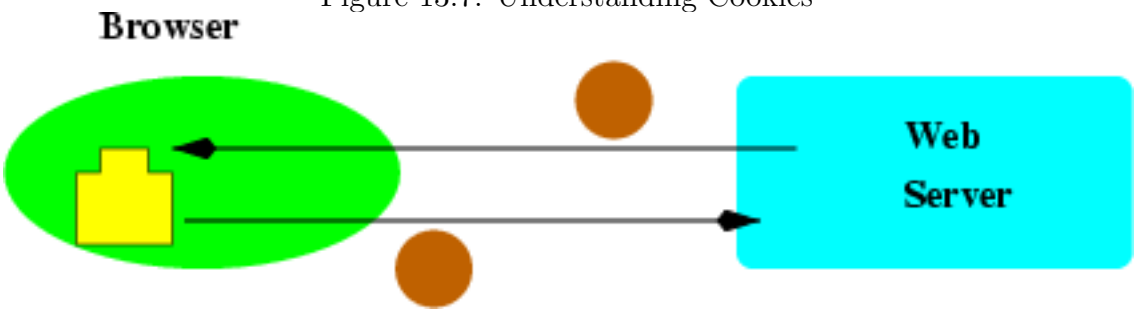
There are several ways to get around the stateless HTTP and achieve session control:

- Hidden form fields—A CGI program generates forms for subsequent steps of a session. Each generated form contains hidden form fields with values from all prior requests. This method essentially makes a multi-step process into a single-step process behind the scenes. Because forms are subject to alteration by end-users, this method is not at all secure or efficient.

- URL rewriting—Instead of hidden fields, the information can be stored in the action URL as extra path-info or command-line parameters. This method has the same drawback as hidden form fields and may conflict with programming or actual form fields.

- Cookies—A way for servers to store information in browsers to be sent to target server-side programs together with future requests. This is perhaps the best way for session tracking under HTTP.

Figure 13.7: Understanding Cookies



13.24 What is a Cookie

A *cookie* is a *name*=*value* pair. that a web server sends to a browser. The browser stores the cookie received and will return it in well-defined future requests to the same server.

A browser may accept or reject a cookie with the approval of the Web surfer. A browser will record cookies accepted and send the appropriate cookies in future requests to servers. The cookie technique was first devised by Netscape and has now become part of HTTP.

HTTP Cookie Headers

A CGI program can send a cookie to a browser with the `Set-Cookie` header in an HTTP response. The `Set-Cookie` header takes the following general form:

Set-Cookie: *name*=*value*; expires=*date*;
path=*path*; domain=*domain_name*; secure

In addition to the required *name*=*value* pair, a cookie has the following optional attributes:

***Expiration time*:** A time/date string, in a special format (*Wdy, DD-Mon-YYYY HH:MM:SS GMT*), to indicate the expiration time of the cookie. A browser will return the cookie until the expiration time is reached even if the user exits the browser and restarts it. Omit this attribute and the cookie will remain active until the browser is closed.

Domain: A domain name or prefix for which the cookie is valid. The browser will return the cookie to any host that matches the specified domain. Thus, if the cookie domain

is "`.webtong.com`", then the browser will return the cookie to Web servers on any of the hosts "`www.webtong.com`", "`control.webtong.com`", and so on. A cookie domain must contain at least two periods to avoid matching on top level domains like "`.com`". The default domain is the host where the cookie originates.

***Path*:** If the `path` attribute is set, the browser will return the cookie only with requests with a URL matching the path. For example, if path is `/cgi-bin`, then the cookie will only be returned to every script in the `cgi-bin` directory. The default path is '`/`'.

The `secure` *flag*: If the word `secure` is included, the cookie will only be sent with requests via a secure channel, such as SSL[1] (a URL that starts with `https`).

Browsers returns a cookie with the `Cookie` header:

`Cookie:` *name1*`=`*value1*`;` *name2*`=`*value2*`; ...`

Consider the following scenario:

1 A Web client requests a document, and receives in the response:

```
Set-Cookie: Customer=Joe_Smith; path=/cgi-bin/;
  expires=Monday, 09-Nov-2000 23:12:40 GMT
```

2 Later when the client sends to this server a request whose URL starts with `/`, the request will include the `Cookie` header:

```
Cookie: Customer=Joe_Smith
```

3 Client now receives from the same server a response that contains another cookie:

```
Set-Cookie: SessionID=AbX0017e; path=/cgi-bin/;
  expires=Monday, 09-Nov-2000 23:12:40 GMT
```

[1] Secure Socket Layer

4 Future requests from the client (to this server whose URL starts with "/") will include the `Cookie` header:

```
Cookie: Customer=Joe_Smith SessionID=AbX0017e;
```

The following points are worth noting when using cookies.

- Cookies for different domains or paths are distinct, even if they have the same name.
- A cookie with a current or past expiration time will be discarded by the browser and will no longer be included in requests.
- Cookies can help server-side programs maintain session state through multiple HTTP transactions with the same client (browser).
- Cookies coming back from user agents are available in the CGI environment variable: `HTTP_COOKIE`

13.25 Cookies in CGI Programs

Let's see how Cookies are set and received in a CGI program. To set a cookie a CGI program can send a `Set-Cookie` header:

```
print "Set-Cookie: Customer=Joe_Smith; ";
print "path=/cgi-bin/; "
print "expires=Monday, 16-Oct-2000 09:00:40 GMT\n";
print "Content-type: text/html\n\n
```

To receive cookies, a CGI program uses the value of the `HTTP_COOKIE` environment variable:

```
$cookies = $ENV{'HTTP_COOKIE'};
if ( $cookies ne "" )
{   print "HTTP_COOKIE: $cookies"; }
else
{   print "No HTTP_COOKIE"; }
```

Figure 13.8: Domain Name Availability Form

`CGI.pm` offers several functions to make cookie handling easy. The `cookie` function creates a cookie

```
$myCookie =
   cookie(-name=>'SessionID',          ## cookie name
          -value=>'AbX0017e',          ## cookie value
          -expires=>'+1h',             ## cookie expiration time (A)
          -path=>'/cgi-bin/member/',   ## cookie path
          -domain=>'.webtong.com',     ## cookie domain
          -secure=>1);                 ## cookie secure flag
```

and the `header` function can set any HTTP response header including `Set-Cookie`:

```
print header(-cookie=>$myCookie);
```

Be sure to set all headers before any output to `STDOUT`.

With `CGI.pm` you receive incoming cookies (from browsers) with a simple call:

$*value* `= cookie('`*name*`');`

If the cookie stored a Perl hash array value, the you can recover that hash array simply with

%*arr* `= cookie('`*name*`');`

The expiration time (line `A`) is set with strings in the following forms:

```
"now"   -- expire immediately     "+180s" -- in 180 seconds
"+2m"   -- in 2 minutes           "+12h"  -- in 12 hours
"+1d"   -- in 1 day               "+3M"   -- in 3 months
"+2y"   -- in 2 years             "-3m"   -- 3 minutes ago(!)
```

13.26 `CGI.pm` Cookie Example

For an application of cookies for session control, let's look at a Perl script to check the availability of domain names for registration (Ex: **DomainCheck**).

Figure 13.9: Domain Check Results

The HTML form (Figure 13.8) is:

```
<p><b>Check Domain Name Availability</b></p>
<form method="post" action="/cgi-bin/wb/domaincheck.pl">
<p>Check <input name="dn" size="16" />
   .com, .net, and .org  
<input type="submit" value=" go " /></p>
</form>
```

The CGI program remembers the domain names checked and lists the ones that are available and unavailable as the user continues to check new possibilities (Figure 13.9).

The **whois** command is used to check if a domain name is taken already (lines 1 and 6). The hash `%recover` receives (line 2) a Perl hash saved as a cookie(lines 7-8). Values on the hash are separated into the available `$namelist` and the unavailable `$takenlist` (line 3).

```
#!/usr/bin/perl -T
use CGI qw(:standard);
$ENV{'PATH'}='/usr/sbin:/sbin:/bin:/usr/bin';
delete @ENV{qw(IFS CDPATH ENV BASH_ENV)};

## script values
$cookie_path = '/cgi-bin/wb/domaincheck.pl';
$front_file="domain.front";
$back_file="domain.back";
@tld_list=(".com", ".net", ".org");
$found="", $reply="";
$found_not="";
$whois="/usr/bin/whois";                                    ## (1)
##### in-script configuration values END

# Get previously found domain names from the cookie:
%recover=cookie('tlds');                                    ## (2)

foreach $key (sort keys %recover)                           ## (3)
{  if ( $recover{$key} == 1 )
   {  push(@namelist, $key);  }
   else
   {  push(@takenlist, $key);  }
}

## check domain names
$name=param('dn');    ## fresh domain name to check     ## (4)

if ( $name )
{  foreach $top ( @tld_list )                               ## (5)
   { $try= "$name$top";
     if ( $recover{$try} < 1 )
     {  $try = noSpecial($try);

        $result=`$whois $try | /bin/grep "No match"`;  ## (6)

        if ( $result ne "" )     ## available
        {    push(@namelist, $try);
```

```
            $found .= "  $try";
            $recover{$try}=1;
        }
        else                    ## unavailable
        {   push(@takenlist, $try);
            $found_not .= "  $try";
            $recover{$try}=2;
        }
     }
   }
}

## set cookie
$the_cookie = cookie(-name=>'tlds',                    ## (7)
                -value=>\%recover,
                -expires=>'+1h',
                -path=>$cookie_path);

# Print the cookie header with expiration date
print header(-cookie=>$the_cookie, -charset=>'UTF-8');   ## (8)
```

The new domain name possibility is obtained from input form data (line 4) and checked in a list of top-level domains (line 5). The UNIX `grep` command looks for the string `No Match` in the **whois** result which indicates an available domain name. A checked domain name joins either the `$namelist` or the `takenlist` and is placed in the hash `%recover` that is stored via the cookie for the next step.

The program is now ready to construct and send the response page to the client. The reply page lists available and unavailable domains checked so far and invites the user to try another possibility.

```
##### send HTML page
outputFile("$front_file");

$reply .= '<table><tr><td style="width: 10px">
            </td><td>';

if ($found ne "")    ## available
{  $reply .= '<p style="font-size: larger">Congratulations:
         we found<code style="font-weight: bold">'
```

```
         . "$found</code></p><p>available.</p>";
}
elsif ( $found_not  ne "" )
{  $reply .= '<p style="font-size: larger">Unfortunately:
         </p><p><code style="font-weight: bold">' .
         "$found_not</code></p><p>are not available.</p>";
}

$reply .= '<p>So far, the following domain names have
      been checked:</p><table><tr><th>Available</th>
      <th style="width:30px"> </th><th>Unavailable
      </th></tr><tr><td align="left" valign="top">';

if (@namelist)                                          ## (9)
{  foreach $name(@namelist)
   {  $reply .= "<code>$name</code><br />\n"; }
}

$reply .= '</td><td style="width:30px"> </td>
           <td align="left" valign="top">';

if (@takenlist)                                         ## (10)
{  foreach $name(@takenlist)
   {  $reply .= "<code>$name</code><br />\n"; }
}
$reply .= "</td></tr></table></td></tr></table>";
outputFile("$back_file");
sendPage($reply);
exit;
```

The `domaincheck.pl` program makes use of several functions we have defined before in this chapter. The complete source code for this practical application can be found in the example package.

13.27 Summary

Perl derives its syntax from C and the UNIX shell. Perl scalar, array, and hash variables use the `$`, `@`, and `%` prefix respectively. Arrays use zero-based indexing and hashes access

values by keys. Strings can be concatenated with the . operator. Perl uses values in a context-dependent way. One result of this is that numbers and their string representations can be used interchangeably.

Perl functions are defined with the notation

`sub` *functionName*`{ . . . }`

Scalar, array, and hash arguments are passed in a flat list which is accessed in the called function by the array `@_` whose elements `$_[0]`, `$_[1]`, ..., are references to the arguments. You can avoid the "flattening" by passing references.

The construct `` `shell command` `` executes the shell command and returns its output. Perl and Javascript uses almost identical regular expressions. The `=~` operator is used for string matching and substitution. A pattern is specified as */pattern string/* or `m|`*pattern string*`|` which allows you to pick the pattern delimiter (`|`). . Use `STDIN` and `STDOUT` for standard I/O and the `open` function to create file handles for I/O to/from files and processes.

It is good to turn on the taint mode to add security to CGI programs. Tainted user input causes guarded operations to fail. This forces the program to check and untaint user input before using them in guarded operations.

A Perl module provides a separate name space. You import a Perl module with `use` *module* `qw(` *features* `)` to access its functions and data directly. The `CGI.pm` module provides good facilities for writing CGI scripts including HTML generation, HTTP header generation, form data access and modification, cookie and debugging support.

The HTTP cookie is a mechanism to get around the statelessness of the HTTP protocol. Server-side programs can set cookies for browsers (client-side agents) to return them in future requests. `CGI.pm` support cookie usage with simple functions to set and retrieve cookies.

Exercises

Review Questions

1. What is the form of the very first line of a Perl program? What does this line do? How does this line change if you move the program to a different computer?

2. Where can you find Perl documentation? On UNIX systems? On the Web?

3. How do you make a Perl program executable as a regular program? As a CGI script? How do you invoke a Perl program from the command line? Describe the URL used to access a CGI program.

4. What are the three types of variables in Perl? Please explain. What is the difference between using double quotes and single quotes to form strings in Perl? What is the difference between these two constructs:

```
$var = "$filename.$suffix";
$var = "$filename" . "$suffix";
```

5. Consider Perl arrays. How does one form an array? Access/set array elements? Obtain array length? Add/delete elements at the beginning or end of the array? Display an array?

6. Consider Perl hashes. How does one form a hash? Access/set hash elements? Obtain keys/values? Add/delete pairs? Display a hash?

7. In Perl how do you find the length of a string? Join two strings? Obtain substrings? Find if a character is on a string?

8. How does one declare variables in Javascript? in Perl?

9. In Perl what is true? What is false?

10. What is a file handle? How do you open a file handle? Name the handles for standard I/O.

11. How do you define a function in Perl? How does such a function receive arguments?

12. Compare pattern matching in Javascript with that in Perl and state their similarities and differences.

13. List the special Pattern characters in Perl.

14. What is taint mode for Perl? Why is it important for CGI programs?

15. How does CGI.pm help CGI script testing and debugging?

16. What is session control under HTTP? Why is there a need for it?

17. What is a cookie? How are cookies used to maintain session state?

18. What HTTP headers are for setting and returning cookies? Explain.

Assignments

1. Deploy the Perl program Ex: **PerlCmdLine** in Section 13.3 on your system and make it run.

2. Write a Perl script that takes what you type on standard input into a file whose name is given as a command-line argument.

3. Write a Perl script that displays a file whose name is given on the command line.

4. Take the `htmlBegin` function (Section 13.11) and the file it uses and deploy them in your `cgi-bin` and experiment with it from the Web.

5. Take the `formMail.cgi` program (Ex: **FormToMail** Section 13.12) and deploy it in your Web space. Make it work.

6. Continue from the previous assignment and make `formMail.cgi` run in taint mode.

7. Write a Perl script to look for the pattern `/<br>/` in a file and create a transformed file replacing `<br>` by `<br />`.

8. Take the password handling code in Section 13.17 and implement a password checking CGI program.

9. Make the `join.cgi` program in Section 8.17 run in taint mode.

10. Make the Ex: **PageSearch** (Section 13.15) program run in taint mode.

11. Take the email address checking program in Javascript (Section 9.15) and translate it into Perl.

12. Connect the club joining example (Ex: **JoinClub** Section 8.17) to the `formMail.cgi` code in Chapter 13. Make sure the email checking is deployed with Javascript in the form and in Perl in the CGI program.

13. Take the example in Section 13.26 (Ex: **DomainCheck**) and make it into a complete program accessible from the Web. Use it to check domain name availability.

14. Implement an on-Web *Imagemap Code Generation* facility:

    (a) The user uploads any Web-supported image for the image map (Section 2.19).

    (b) A page is displayed where the user can enter a rectangle, a circle, or a polygon with mouse operations. The user can also supply a URL as the target for the selected area.

    (c) After each completed area, the program displays the additional `area` code for the `map` element being constructed.

    (d) When the user is done, the completed `map` code is displayed.

15. Follow the model given in Section 10.20 to construct a time clock service consisting of

(a) A server-side CGI program that reports the time in a standard time zone.

(b) A client-side DHTML to access the CGI service and displays the time by ticking the seconds forward for a reasonable length of time, say 15 seconds.