ZDU Student Manual

CGI Programming with Perl

CGI Programming with Perl

ISBN: **0-73725-354-1**
Part number: **ZDU56708**

ACKNOWLEDGMENTS

Content Development

The content of this self-study guide is based on the training course "CGI Programming with PerlCGI Programming with Perl," developed by Instruction Set, Inc. for its curriculum of instructor-led technical training. This guide was designed and developed by an Instruction Set team of instructional designers, course developers, and editors.

Administration

Vice President and General Manager of ZD University: Ed Passarella

Marketing Director: Risa Edelstein

Director, ZD University: George Kane

Senior Editor, Curriculum: Jennifer Golden

Project Director, Instruction Set: Laurie Poklop

Project Manager, Instruction Set: Sandy Tranfaglia

DISCLAIMER

CONTENTS

CGI Programming with Perl

LESSON 1: GETTING STARTED WITH PERL

LESSON 2: GETTING STARTED WITH CGI

LESSON 3: OPERATORS

LESSON 4: FLOW CONTROL

LESSON 5: WORKING WITH SCALARS

LESSON 6: WORKING WITH LISTS

LESSON 7: WORKING WITH HASHES

LESSON 8: READING, WRITING AND MANIPULATING FILES

LESSON 9: PATTERN MATCHING

LESSON 10: CREATING AND USING SUBROUTINES

LESSON 11: REFERENCES

LESSON 12: PACKAGING AND MODULES

LESSON 13: USING THE `CGI.pm` MODULE

LESSON 14: RETRIEVING DATA WITH CGI

LESSON 15: CGI SCRIPT DEBUGGING AND SECURITY

ANSWERS: TO REVIEW QUESTIONS

LESSON 1

Getting Started with Perl

OVERVIEW

Perl stands for Practical Extension and Reporting Language. It was created as a general tool for creating reports. As its user base grew, Perl was modified and more thoroughly developed. Today, Perl has become a popular language that can be used in many different ways.

LESSON TOPICS

- What is Perl?
- Perl Options
- Locating Perl on the System
- Running Perl Programs as Interpreter Files
- Basic Statement Syntax
- Variables
- Basic Operators
- Simple Control Constructs
- Using Functions
- Standard Input and Output

OBJECTIVES

By the end of this chapter, you should be able to:

- Understand Perl execution options.
- Understand basic statement syntax.
- Use the keyboard for program input.
- Write output to the display screen.
- Write simple Perl programs.

WHAT IS PERL?

Perl is an acronym for Practical Extraction and Reporting Language. It was created several years ago by Larry Wall as a general-purpose tool for generating reports. Among other things, it extended the capabilities of another scripting language called *awk*. Larry enhanced the basic language and put it out on the Internet for free use and comments.

The initial users of the language provided feedback, asking for additional capabilities. As a result, the language was improved, with both functionality and portability being added. Programmers who needed to write complex scripts for Unix administration found that Perl was a "one-stop" programming language for tasks that typically required a handful of tools and other utilities. As the language grew in capability, so did the number of users. Estimates have exceeded 300,000 programmers world-wide, with thousands more being added every month.

The language is still under the stewardship of Larry Wall, who is solely responsible for its content. Fortunately, Larry has kept improving the language with major releases coming out about every year.

So what is Perl, exactly? It is a language which in some sense is hard to describe, because it can be used in many different ways. However, the following can be said without restricting its definition too narrowly:

➤ *It is a scripting language.*

Perl programs can easily be written to replace Unix shell scripts which rely heavily on Unix utilities and other tools, such as sed and awk.

➤ *It is an interpreted language.*

This is not strictly true in the sense of a Unix shell, but on the other hand, it is not a compiled language either. The source code is interpreted into an intermediate form which is then executed.

➤ *It is C-like.*

New users quickly discover that Perl borrows heavily from the C Programming Language in terms of statements, operators and control constructs. This makes it easier for programmers familiar with C to become productive with the language.

➤ *It has strong pattern-matching capabilities.*

In this role, Perl borrows heavily from the Unix-based Stream Editor (sed) and awk. It understands extended regular expressions and supports a number of pattern matching extensions.

➤ *It is a list processor.*

Special variable types are provided which make it easy to work with both indexed and associative arrays.

➤ *It is a systems language.*

Perl has many built-in functions which allow it to call directly for services from the operating system.

Perl is a language which is simple to begin programming with, but difficult to master. It has rich functionality and users are continually finding new ways to apply it. This course is designed to give students a strong start with the language, but ultimately, their level of skill will be determined by their willingness to use the language and to explore its capabilities.

Obtaining and Installing Perl

Perl is freely available from the Internet. It is distributed under two licenses: the standard *GNU copyleft license* and the somewhat less restrictive *Artistic License*. The copyleft license means that if you can execute Perl on your machine, you should have access to the full source code as well.

Consequently, Perl can be downloaded without cost and used without royalties or fees of any kind. This is one aspect of the language which has spurred its growth, especially in the Unix community.

The primary distribution site is the *Comprehensive Perl Archive Network* (CPAN). This site has numerous mirror sites, which make it simple to download the most recent version from a nearby server.

Users with WWW browsers can go to the Perl home page at: *http://www.perl.com/CPAN*.

This brings up the CPAN Multiplex Dispatcher with an interface for selecting both a site and files for downloading. Visitors to the site will notice that there are many Perl related files available beyond the basic Perl software itself.

Other sites are available which can be used to download files using the File Transfer Protocol (ftp). Users interested in downloading files using anonymous ftp can try one of the following sites:

- *ftp.perl.com*
- *ftp.cs.colorado.edu*
- *ftp.cise.ufl.edu*
- *ftp.funet.fi*
- *ftp.cs.ruu.nl*

Once at the ftp site, look around for a directory with a name like `/pub/perl/CPAN`.

Once the file has been downloaded, it will have to be uncompressed and then the individual files will have to be extracted. The `README` files are detailed enough to guide a System Administrator through the installation process.

Perl Resources

There are numerous resources available for both novice and experienced Perl programmers.

The first place to look is the Perl home page at `http://www.perl.com/perl`. Browsing through the various links will reveal a variety of articles, tutorials, and example code, plus links to other Perl-related sites.

Under *Other Resources*, look for the following:

- Newsgroups:
  - *comp.lang.perl.announce*
  - *comp.lang.perl.misc*
  - *comp.lang.perl.modules*
  - *comp.lang.perl.tk*
- Mailing Lists
- Related Sites
- *The Perl Journal:* A quarterly publication devoted entirely to Perl. Back issues are available and users can subscribe online.
- *The Perl Institute:* In their own words, "A nonprofit organization dedicated to making Perl more useful for everyone."

Articles on Perl-related topics can be found occasionally in industry trade periodicals, especially in those focusing on Unix. There are numerous books available from various publishers.

The definitive text for the language is still *Programming Perl*, by Larry Wall, Tom Christiansen, and Randal L. Schwartz, published by O'Reilly and Associates, Inc. This book is often referred to as the "Camel Book" due to the picture of a camel on the front cover.

A companion text called *Learning Perl,* by Randal L. Schwartz, published by O'Reilly and Associates, Inc. is useful for new programmers just getting started with the language.

As with any topic, books on Perl vary in quality. Tom Christiansen has a number of book reviews on Perl books which can be found through the Perl home page. These may be useful as a means of narrowing down the list.

PERL OPTIONS

The basic syntax for Perl is:

```
perl [options] [source_file|cmd]
```

Several options useful when developing Perl programs are shown in Table A:

TABLE A. *Options in Developing Perl*

Options	Action/Meaning
`-v`	*Prints version number (does not require use of actual Perl source file)*
`-c`	*Syntax checking without running the Perl program*
`-w`	*Print warnings*
`-e`	*Single Perl command*

For example, to get the version number of the Perl application, run the following from the command line prompt:

```
$ perl -v
```

To run a Perl program with the warning option from a file called `myprog`, type in the following from the command line prompt:

```
$ perl -w myprog
```

The `-e` option is used to execute a single Perl command from the command line. Multiple commands require multiple `-e` options. This particular option is useful for embedding a simple Perl command in a shell script. To run a simple one-line program from the command line, type in the following:

```
$ perl -e 'print("Hello World\n")'
```

Other options will be discussed later in the course.

LOCATING PERL ON THE SYSTEM

Perl is normally installed on Unix systems in either `/usr/bin` or `/bin`, but may be placed elsewhere on a particular machine. For Windows NT installations, it is common to place Perl in a folder called `\Perl` or `\Programs\Perl`.

Users can check for Perl on Unix systems by running one of the commands:

➤ in the Korn or Bourne Shell:

```
$ whence -v perl
```

➤ in the C Shell:

```
$ which perl
```

If that fails, search the filesystems with the following command:

```
$ find / -name perl -print 2>/dev/null
```

Use the `find` utility on Windows NT systems.

The location of the Perl executable should be placed in the `PATH` environment variable so that Perl can be executed directly from the command line.

RUNNING PERL PROGRAMS AS INTERPRETER FILES

Perl source files are text files and can be created with any text editor. By convention, Perl source files are named with a `.pl` extension.

Most Perl programs are written and executed as interpreter files. Assuming the Perl executable is located in the `/usr/bin` directory, to create an interpreter file the following line is always used as the *first* line of the source file:

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

The Perl source code is placed on the following lines in the same file. To execute interpreter files on a Unix system, add execute permission to the file with a command like:

```
$ chmod +x source_file
```

The interpreter file can then be executed directly from the command line *without* prepending the application name `perl`:

```
$ my_perl_prog.pl
```

BASIC STATEMENT SYNTAX

As stated previously, Perl statement syntax mimics the C language syntax. The following are some examples:

- Variables and keywords are case-sensitive.
- Statements are closed with a terminating semicolon.
- {Curly braces} are used to bound statement blocks. How they are placed around the statements is left to the discretion of the programmer.
- Function parameter lists are (optionally) enclosed in parentheses.

Since the semicolon is used as a statement termination character, statements can extend over multiple lines.

Example:

```
#! /usr/bin/perl
$input = <STDIN>;
if ($input ne “”) {
  print($input);
}
```

Comments

Comments are started with a hash character (#) and continue to the end of the line. For this reason, it is not possible to place comments in the middle of a statement.

The first line of an interpreter file begins with a hash character as well, but is a special line recognized by the shell. It is not a comment. If the line appears on a line other than the first one in the file, it is interpreted as a comment.

As with any language, it is considered good programming style to make liberal use of comments in the source code.

Example:

```
#! /usr/bin/perl
          # The first line is not a comment, but this one is!
$x = 5;   # Comment OK after a statement
```

VARIABLES

Perl supports several variable types, not to be confused with data types. The variable type is indicated by a special leading character assigned to each type. Thus, there is no confusion if two different variable types differ only by their first character.

The variable types with examples are:

- Scalar (`$name`)—a variable which holds a single value.
- Array (`@items`)—an indexed list.
- Hash (`%city`)—an associative array, keyed by strings.
- Subroutine (`&calculate`)—a callable section of a Perl program.
- Typeglob (`*light`)—everything named *light*.

Beyond the first special character, variable names may start with either an underscore character (_) or an alphabetic character. After that, the names may continue with an underscore character or an alphanumeric character. Names for all variable types are case sensitive.

Example:

```
$x = 2;
@list = (1, 2, 3, 4, 5);
%mfr(“747”) = “Boeing”;
&display_phone_number(“Instruction Set”);
*filehandle;
```

We will study each of these variable types in more detail as we proceed through the course.

BASIC OPERATORS

Perl uses the same operators that are found in the C Programming Language. In addition to the usual assignment and arithmetic characters, Perl supports two sets of relational operators, one set for numbers and another set for strings:

- Arithmetic

```
+, -, *, /, %
```

- Assignment

```
=, +=, -=, *=, /=
```

- String concatenation

```
.
```

- Relational (numbers)

```
==, !=, >, >=, <, <=
```

➤ Relational (strings)

```
eq, ne, gt, ge, lt, le
```

This is not a complete list, but should serve as a starting point to write basic programs.

SIMPLE CONTROL CONSTRUCTS

Perl has the usual C language control constructs for branching and looping. In addition, there are a few other constructs which are unique to Perl which will be discussed in a later section.

Programmers should note that curly braces are *not* optional around statement blocks associated with control constructs, even if there is only one statement.

➤ branching:

```
# Example if-else statement
if ($i < 5) {
  print("$i\n");
}
else {
  print("Maximum value exceeded!\n");
}
```

➤ looping:

```
# Example while loop
$input = <STDIN>;
while ($input ne "") {
  print($input);
}
```

```
# Example for loop
for ($i = 0; $i < @list; $i++) {
  print("$list[$i]\n");
}
```

There is no multi-branch control construct in Perl.

USING FUNCTIONS

Functions are callable pieces of code that are made available to the program by Perl. For the time being, we will limit the discussion to *built-in functions*.

Functions are called by function name and may accept a parameter list, optionally enclosed in parentheses. The tendency is to omit the parentheses, but initially, it is recommended that the student include them for reasons of readability.

Example:

```
#! /usr/bin/perl
@input = <STDIN>; # Read the input into a list variable
chomp(@input);           # Remove all trailing newlines
@sorted = sort(@input);  # Sort
print(@sorted);          # Print the sorted list
print("\n");             # Print a newline character
```

The chomp() and chop() Functions

In the previous example, the `chomp()` function was applied to the variable holding the input values. Although it is not apparent, whenever Perl reads a line of input, it includes the newline character at the end of the string. The `chomp()` function is used to trim any and all newline characters from the end of the string. If it is applied to a string which has no trailing newline character, it has no effect. The function returns the number of newline characters which were removed.

The `chop()` function is similar to `chomp()`, except it trims the last character in a string regardless of its value. For that reason, it is considered a little safer to use the `chomp()` function if the intent is to remove a trailing newline character. This function returns the character which was removed.

Example:

```
$number = <STDIN>;     # Read a line
chomp($number); # Remove the newline character while
($number ne “”) {      # Test for null string
    print(“$number\n”);# Print remaining string
    chop($number);     # Remove the last number
}
```

Output:

```
12345
1234
123
12
1
```


STANDARD INPUT AND OUTPUT

Reading from Standard Input

Open files in Perl are referenced by means of file handles. These are labels associated with the files when they are opened. By convention, file handles are created using upper case letters.

In Unix, when a child process is created, it inherits the open files of the parent process. Unless previously closed, the open file list includes standard input, standard output, and standard error. Perl establishes file handles for these files of `STDIN`, `STDOUT`, and `STDERR`. By default, standard input refers to the keyboard, but redirection can be used from the command line to obtain input from another file.

Lines of input are read from a file by placing the file handle in brackets (<>). If the file is read using a scalar variable, a single line is assigned to the variable. If the file is read using an array variable, the entire file is read into the array with one line stored in each array element in sequence.

In a later section, we will discuss how other files are opened and assigned file handles by the programmer.

Examples:

```
$line = <STDIN>;# Read a single line
```

```
@file = <STDIN>;# Read the entire file
```

Writing to Standard Output and Standard Error

Standard output and standard error both send output to the screen by default. As with standard input, they can be redirected to other files from the command line.

The `print()` function is used for unformatted output. By default, it sends output to standard output. Therefore, the file handle is not needed with the `print()` function if the output is to be sent to standard output.

If output is to be sent to other files, the file handle associated with the file is used with the `print()` function.

Note that a newline character is not automatically appended to a string by the `print()` function. If a newline character is needed, it must be included as part of the output string, represented by a `\n`.

Examples:

```
print (“Hello World!\n”);# Print to standard output
```

```
print STDERR (“Oops!\n”);# Print to standard error
```

A Simple Example

Here is a simple example to tie some of the basics together in a single program:

```
#! /usr/bin/perl
print (“What is your name? “);# Print user prompt
$name = <STDIN>;          # Get a name from the user
chomp($name);             # Remove newline character
if ($name eq “”) {        # See if input is null
  $name = “whoever you are”;# Assign default string
}
print (“Hello, $name\n”); # Say hello to the user
```

LESSON SUMMARY

In this lesson, you have learned the following:

- Perl is freely available from the Internet.
- Many resources are readily available.
- Basic statement syntax is similar to that of C Programming Language in terms of statements, operators and control constructs.
- Several variable types are supported by Perl.
- Perl has numerous built-in functions.
- The `chomp()` and `chop()` command is used to remove last character of strings.

REVIEW QUESTIONS

1. Name one principal site on the World Wide Web to get more information about Perl.

2. Who wrote Perl?

3. What Perl flag should you use to get warnings of possible errors when invoking Perl.

4. Identify the type of variable or object associated with each of these:

   ➤ `$var`

   ➤ `@var`

   ➤ `%var`

   ➤ `&var`

5. Does assigning to `@var` affect a value in `$var`? What about `%var`?

Answers on page 200

EXERCISE

1. Standard input, standard output, and standard error are available to Perl programs. Run a Perl program from the command line using the `-e` option which will print out your name and address in a mailing label format to standard output.

2. Write a Perl program which asks the user for their name and address using three separate prompts. Test each line for null values. Print out the lines to standard output in a mailing label format.

3. Create a data file which lists last name, first name, address, city, state, and zip, each on separate lines. Write a Perl program which reads each line (using redirection and standard input) and then prints the data to standard output.

4. Write a Perl program which reads a file from standard input and prints it to standard output. Combine the program execution with redirection to copy a file.

LESSON 2

Getting Started with CGI

OVERVIEW

CGI or Common Gateway Interface, is a set of protocols that allow communication between HTTP (HyperText Transfer Protocol) servers, and HTTP clients (typically Web browsers). This lesson presents the basic concepts regarding how client and server communicate under CGI.

LESSON TOPICS

- Web Server and Client Communication
- Setting Up a CGI Program
- Common Mistakes

OBJECTIVES

By the end of this chapter, you should be able to:

- Learn the basic operation of request/response procedure for a CGI script.
- Install a script on your server.
- Identify common mistakes made when using CGI scripts.

WEB SERVER AND CLIENT COMMUNICATION

The Common Gateway Interface (CGI) is a server-side API (Advanced Programmer's Interface) that provides methods that HTTP servers (Web servers) use to pass client requests to an external program associated with the server. These programs, often referred to as *CGI scripts*, enable Web servers to process certain requests they would be otherwise unable to manage. In this way, CGI scripts can be considered helper applications that the Web browser uses to process documents it could not ordinarily process. For example, most (if not all) Web servers do not have a built-in guest book facility. To provide a guest book at a Web site you must instead write a helper application to handle guest book facility requests. The server knows whether or not it can use the resource of a CGI script based on the URL (Unified Resource Locator) of the request.

The following two example URLs could be used to request the use of a CGI script to provide a guest book facility, by a site that had control over the domain name *web.guest.com*:

- *http://web.guest.com/cgi-bin/guest_book*
- *http://web.guest.com/some_dir/guest_book.cgi*

The examples above illustrate the two most common ways a server recognizes a request to use a CGI script resource.

The first example uses a CGI directory, `cgi-bin`. The server has been configured to consider that a request for any resource in that directory is a request to invoke a CGI script. `guest_book` is a resource found in the

NOTE

A server may use any number of methods to identify a resource as one to be treated as a CGI script, but the above methods are the two most commonly used (and the ones we will cover in this class).

`cgi-bin` directory. The server, based on its configuration, knows to execute this resource as a CGI program, since all resources in that directory are earmarked for use as CGI scripts.

The second example shows a CGI script identified by the file suffix `.cgi`. In this case, the server configuration indicates that any resource ending in `.cgi` is a CGI script, and knows to execute it as a program.

Once the server has actually identified the requested resource as a CGI script, it puts the information about the request in a place where the CGI script can easily find it. This information will allows the CGI script to obtain any data being submitted from the client. The server then actually starts the CGI script. It should be noted that the protocol used by CGI scripts is independent of the programming language used. Perl is used because it is considered one of the best programming languages for quickly and easily deploying such scripts.

Once the server has started execution of the CGI script, it waits for a response from the script, and then passes the response to the client that made the original request.

The CGI script will first process any data that it has received from the client (this will be discussed in detail later). The CGI script responds by sending its desired output to standard out (`STDOUT` file handle). When the server starts the CGI script, it first made sure that standard out for the CGI script was redirected to send its data to the server. All the response then entails is for the CGI script to use a Perl print statement to standard out. The server then ensures if there are no errors that this output is appropriately sent to the client, the final destination. Once the CGI script has sent its complete response, it exits, thus completing the transaction.

The response must be properly formulated, however. The first thing the CGI script has to output to the server is an HTTP document header that specifies what type of document is being returned. After that, it can output the HTML it wants to the display.

The program shown in the exercise illustrates a simple CGI program. Throughout this course, the focus will be on the use of the `CGI.pm` module as a way of facilitating the development of better CGI scripts in less time.

SETTING UP A CGI PROGRAM

There are several things you need to know about your Web server before you can run the first example program. These include:

- Does the server recognize CGI scripts by extension or by directory placement?
- If it uses extensions to recognize CGI scripts, is it `.cgi` (typical case)?
- Where must the CGI scripts go so that the server is able to recognize them?
- In what directory should static documents be placed (not CGI scripts), and what file extensions should be used with them? (For the most part, this is obvious, since HTML files end in `.html` and text files end in either `.text` or `.txt`. But, the extensions that go with certain kinds of files, such as audio and other multimedia files may be unknown.)
- What URL is needed to access your HTML files? What URL is needed to access your CGI directory (if your site uses one)?
- Where are the server logs kept (especially the error log)?

All of the above information should be in the documentation for your account from your ISP. If they are not there, or if you were not in fact given any documentation with your account, then you should ask the web site administrator for the information listed above. You will need this information to do the exercise.

COMMON MISTAKES

The following is a list of common mistakes which can occur when setting up a CGI script. These guidelines will help the student with difficulties in get-

ting the example script to run the first time, as well as help avoid similar mistakes in the future:

➤ Is the URL correct?

In the case of a mistaken URL, there should be a message saying that the file requested was not found. To fix this problem, check the URL and try again.

➤ Is the server able to execute the program?

If it is unable to do so, there will probably be a message that reads something like this: "You do not have permission to access `/cgi-bin/hello.cgi`." If this is the case, go to the directory containing the program and type:

```
ls -l hello.cgi
```

The line it returns should look something like this

```
-rwxrwxr-x  1 usernamegroupname 0 Aug 15 04:40
hello.cgi
```

It is important that the first text block has an `r-x` as the last three characters (`-rwxr-xr-x` instead of `-rwxr-xr--`). The last `r` and `x` means that anyone can read and execute this program, which therefore also allows the Web server to run it.

Correct this problem by running the command:

```
chmod a+rx hello.cgi
```

➤ Does the server recognize the program as a CGI script?

If this is the problem, the text of the program should appear in the browser instead of in its output. To fix this problem, make sure the program is either in a CGI directory or that it is named in such a way

that the server recognizes it as a CGI script. After fixing this problem, try again.

➤ Is there a problem with the syntax of the program?

If this is the problem, there should be a message which reads that there was a server error. To fix it, go to the directory where the CGI script is and test it out by checking the Perl program's syntax with the command:

```
perl -c hello.cgi
```

If no errors exist, it will return the message `syntax OK`. Fix any syntax errors Perl finds and then try the program on the Web browser again.

Note that if none of these solutions work (or none of the above seem to be the problem), ask the instructor for help, or refer to the lesson on debugging.

You should now be prepared to go to the first exercise and run your first CGI script. The remainder of this course will consist of teaching enough Perl to prepare you to write good quality Perl programs. The last set of lessons will cover the remaining details needed to write more complex CGI scripts.

LESSON SUMMARY

In this lesson, you have learned:

- How the CGI resources act as an extension to the HTTP server.
- How an HTTP client initiates a CGI resource request.
- How the server manages the life cycle of a CGI request.
- How the CGI script responds to the HTTP client request
- What information is needed in order to install and run your first CGI script.
- The common mistakes that are made when running a CGI script

REVIEW QUESTIONS

1. What does CGI stand for?

2. List two common mistakes made when using CGI scripts and their systems.

3. What is the server's role in running a CGI script?

4. What is the first thing a CGI script must return?

Answers on page 200

EXERCISE

For practice, modify the script given below to print out correct information about you, and then follow the instructions to have it run on your server. Note that you will typically upload the CGI script to an Internet-ready server. The script used in this example is written in Perl and uses the Perl module `CGI.pm`, which is included in recent Perl distributions.

```
#!/usr/bin/perl

use CGI;

my $cgi = new CGI {name => 'bill', job =>
'programmer'};

my $name = $cgi->param('name');
my $job  = $cgi->param('job' );

print$cgi->header,
    "Hello world, my name is <B>$name</B>",
    " and i am a <I>$job</I>.";
```

1. Copy and modify this program to a file named `hello.cgi`.
2. Upload it to the CGI directory available for you to run CGI scripts on your server.
3. Log into the server and issue the following command to give permissions that allow the script to execute:

```
chmod a+x hello.cgi
```

4. Enter the URL *http://www.the_server_name.com/~me/cgi-bin/hello.cgi* on your browser, to cause that script to execute.

LESSON 3

Operators

OVERVIEW

Operators allow the programmer to write expressions that can transform data. There are many different types of operators. This lesson covers the use of arithmetic, logical, relational, and string operators which form the basis for writing more complex programs.

LESSON TOPICS

- Arithmetic Operators
- Relational Operators
- Logical Operators
- String Operators
- Assignment Operators
- Precedence

OBJECTIVES

By the end of this lesson, you should be able to:

- Apply arithmetic and assignment operators to mathematical expressions.
- Apply relational operators to both arithmetic and string expressions.
- Use logical operators for conditional execution of expressions.
- Concatenate strings and create repeating string patterns.
- Understand principles of precedence as it applies to operators.

ARITHMETIC OPERATORS

Perl has a number of arithmetic operators which make it capable of performing mathematical operations. However, programmers should remain aware that Perl is interpreted language. Very time-consuming calculations should be done in a compiled language if performance is an issue.

In general, Perl does calculations in double-precision floating point, but it also supports integer operations. The arithmetic operators supported by the language are shown in Table B:

TABLE B. *Arithmetic Operators*

Unary	-	negartive
Binary	*	Multiplication
	/	Division
	+	Addition
	-	Subtraction
	%	Modulo
	**	Exponentiation

Parentheses can be used to change precedence and improve readability. The modulo operator (%) produces an integer result, even when its operands are floating point numbers.

Examples:

```
$x = $y + $z + 2;
```

```
$hyp = ($a * $a + $b * $b)**0.5;
```

```
$remainder = $s % $t;
```

Autoincrement and Autodecrement Operators

The autoincrement and autodecrement operators will be familiar to C programmers. In general, they are used to increment or decrement the value of a variable by 1. Autoincrement (++) causes a variable to increment. Autodecrement (--) causes a variable to decrement.

There are two forms of autoincrement and decrement operators: prefix (`--$a`), and postfix (`$a--`). When the prefix form is used in an expression, the value used is the incremented or decremented value. When postfix is used in an expression, the value used in the expression is that before the incrementation or decrementation took place.

In both cases, the final value of an autoincremented variable is one greater than its original value. The final value of an autodecremented variable is one less than the original value.

Example:

```
$x = 5;
$x++;      # (Postfix) $x = 6
++$x;      # (Prefix)  $x = 7
$y = ++$x + 2; # $x = 8, $y = 10
$z = $x++ + 2; # $x = 9, $z = 10
--$x;      # (Prefix)  $x = 8
$x--;      # (Postfix) $x = 7
```

RELATIONAL OPERATORS

Relational operators are used in expressions that evaluate to a true or false condition. For this reason, relational expressions are often used as a condition for a control statements, such as an `if` or a `while` statements.

Relational Operators for Numbers

Values that look like numbers to Perl are used as strings or numbers based on the context of use. For this reason, Perl has a separate set of relational operators for numbers and strings. The principal relational operators that are used with numerical expression are shown in Table C.

TABLE C. *Relational Operators*

<	less than
>	greater than
>=	less than or equal to
>=	greater than or equal to
==	equivalent
!=	not equivalent

Examples:

```
if ($x > 5) {
  $y++;
}
```

```
while ($i < $max) {
  print("$i\n");
}
```

It is important to remember that the expressions `$x > 5` and `$i < $max` return a value of `true` if the relationship holds between the operands.

Relational Operators for Strings

As stated earlier, Perl has separate relational operators for strings. The most important ones you should remember are:

- `eq` Equivalent
- `ne` Not equivalent

Example: :

```
$in = <STDIN>;      # Read line from standard input
while ($in ne "") { # Check for EOF
  print($in);       # Print the line to standard output
  $in = <STDIN>;    # Read line from standard input
}
```

In the code example below, the string looks like a number, but is being compared in a string context. Therefore, it is treated as a string, and the existence of the leading `0`'s will matter.

```
if ($license eq "007") {
  $gun = "Beretta";
}
```

LOGICAL OPERATORS

Logical operators establish logical conditions between its operands. `&&` and `||` take two operands, such as:

```
$a && $b
```

Just like in the English language, `&&` or `AND` results in a `TRUE` value if and only if both its operands are `TRUE`, whereas `OR` returns a `TRUE` only if either of its operands are a `TRUE`.

! or NOT takes a single operand, and it returns the opposite of its operand (a FALSE if its operand is TRUE, a TRUE if its operand is FALSE). The logical operators are shown in Table D

TABLE D. *Logical Operators*

&&	AND
||	OR
!	NOT

In addition, there are equivalent mnemonic forms having a lower precedence as shown in Table E . (Precedence will be discussed later in this lesson.)

TABLE E. *Mnemonic Logical Operators*

and	AND
or	OR
not	NOT

These operators are often referred to as *short-circuit operators*, since the right hand operand is not always evaluated (it is bypassed or short-circuited around).

Suppose two expressions are combined with a logical operator. In the case of the AND operator:

```
expr1 && expr2
```

If expr1 evaluates to TRUE, then expr2 is also evaluated. If expre1 evaluates to FALSE, then the result is guaranteed to be FALSE, and expr2 is not evaluated.

In the case of the OR operator:

```
expr1 || expr2
```

If expr1 evaluates to TRUE, then expr2 is not evaluated, otherwise, expr2 is evaluated.

Examples:

```
$x > $max && print("Max exceeded.\n");
```

```
$name ne "" || $name = $default;
```

STRING OPERATORS

Two useful Perl operators, designed for use with strings are shown in Table F:

TABLE F. *String Operators*

.	Concatenation
x	Repetition

The concatenation operator is used to combine strings. It can be applied with string literals or with variables that hold a string value. Recall that Perl data types are context oriented, so any variable contents which are used in a string context will be treated as strings, even if they look like numbers.

Example:

```
$city = "Boston";
$state = "MA";
print($city . ", " . $state . "\n");
```

The repetition operator is used to create repeating text patterns. The operator is the ASCII character x. The right-hand operand is the number of times the text on the left-hand side is to be repeated. The right-hand operand can be a variable, but it must hold an integer value.

Example:

```
$pattern = "abc";
$repeat = $pattern x 3;# $repeat = "abcabcabc"
```

ASSIGNMENT OPERATORS

Programmers should be familiar with the basic assignment operator (=) used in previous examples. However, there are a number of other assignment

operators which are supported by Perl which have been borrowed from the C programming language.

These other assignment operators are used as a form of shorthand when the result of a simple expression is assigned back to the variable used in the expression.

The assignment operators supported by Perl are shown in Table G:

TABLE G. *Assignment Operators*

=	Simple assignment
*=	Multiplication and assignment
/=	Division and assignment
+=	Addition and assignment
-=	Subtraction and assignment
%=	Remainder and assignment
**=	Exponentiation and assignment
&=	Bitwise AND and assignment
|=	Bitwise OR and assignment
^=	Bitwise XOR and assignment
.=	String concatenation and assignment
x=	String repeat and assignment

Note that the syntax below is also supported by Perl. Assignments are made from right to left:

```
$x = $y = $z;
```

Example:

```
$x += 2;   # $ x = $x + 2
$x *= 2;   # $ x = $x * 2
$x %= 2;   # $ x = %x % 2
$str .= "\n";# $ str = $str . "\n"
```

PRECEDENCE

Precedence refers to the order in which operators are applied when different operators are used, given that the order is not modified by parentheses. Normally operators are assumed to perform from left to right. Some operators, however, violate this rule and operate in a different order because they have higher precedence. Parentheses force the order of operation. So, when in doubt about the order, parenthesize the expression to force the order.

Below is a list of operators in the order of highest precedence to lowest. The operators on the same line have the same precedence.

- `++, --`
- `-, !`
- `**`
- `*, /, %, x`
- `+, -, .`
- `<, <=, >, >=, lt, le, gt, ge`
- `==, !=, <=>, eq, ne, cmp`
- `!`
- `&&`
- `||`
- `=, +=, -=, *=, etc.`
- `not`
- `and`
- `or`

LESSON SUMMARY

The main Perl operators are:

- Arithmetic operators

  `*, /, +, -, %, **`

- Autoincrement and autodecrement operators

  `++, --`

- Relational operators (numbers)

  `<, <=, >, >=, ==, !=`

- Relational operators (strings)

  `lt, le, gt, ge, eq, ne`

- Logical operators

  `&&, ||, !, and, or, not`

- String operators

  `., x`

- Assignment operators

  `=, *=, /=, +=, -=, %=, **=, &=, |=, ^=`

REVIEW QUESTIONS

1. What result does the following code give?

```
$a = 3 + 5 * 2;
```

2. How can you change the order in which the operations are performed without changing the order of the operands? (Clue how can you override the precedence of the operators.

3. In the following program, what is the printed output? What is the value of `$a` when the program is done?

```
$a = 5;  print($a++ . "\n");
```

4. In the following program, what gets printed out? What is the value of `$a` when done.

```
$a = 5;  print(++$a . "\n");
```

5. Why would you use the following expression?

```
open(IN, "a_file") || die("Could not open file\n");
```

6. In the following program, what is the value of `$c` and the value of `$d`?

```
$a = 5; $b = 2; $c = $b < $a;  $d = $b > $a;
```

7. In the following program, what values do `$a` and `$b` get?

```
$a = "00" eq 0;  $b = "00" == 0;
```

8. What will this program print out?

```
$a = '';  $a .= "dog:"; $a .= "cat:";  print($a x 2);
```

Answers on page 201

EXERCISE

1. Write a program which will ask the user for a temperature and then ask whether it is to be converted to degrees Celsius or Fahrenheit. Perform the conversion and display the answer.

   The equations for temperature conversion are:

   - Celsius to Fahrenheit: C = (F - 32) x 5/9
   - Fahrenheit to Celsius: F = 9C/5 + 32

2. Write a program which gets two strings from the user and then prints them out in sorted order.

LESSON 4

Flow Control

OVERVIEW

In this lesson flow control constructs such as the `if-else` *and* `elseif` *are discussed along with functions such as* `warn()`, `die()`, *and* `exit()` *that usually are used in conjunction with them. The significance of the ability to determine the true or false status of constructs is shown. Loops and the different methods of their use are also included as vital to the discussion of flow control.*

LESSON TOPICS

- True and False in Perl
- The warn(), die(), and exit() Functions
- The while Loop
- The for Loop
- The foreach Loop
- The last Statement
- The Many Ways to Do Something

OBJECTIVES

By the end of this lesson, you should be able to:

- Apply the `if-else` and `elsif` constructs to situations requiring two and multi-way branching.
- Use both exit-condition and entry-condition `while` and `until` constructs for program loops.
- Use both forms of the `for` construct.
- Apply the `last`, `next`, and `redo` statements to loop constructs.

TRUE AND FALSE IN PERL

The use of control constructs like the `if-else` and `while` statements introduced earlier, hinge on determining the boolean true or false value of an expression.

Where relational expressions are used, a boolean condition is automatically determined. However, Perl also supports the use of numerical expressions and string expressions or values as the boolean or conditional expressions that predicate the path taken by a control constructs (e.g., `if` or `while`). The equivalent boolean value of a numerical or string expression is a bit more ambiguous.

In general, Perl decides if a numerical or string expression is true or false based on its value. A numerical expression is true if it evaluates to a nonzero value and false if it evaluates to a zero value. A string expression evaluates to false if it evaluates to an empty string, otherwise it evaluates to true. This covers 99% of the cases.

There are subtle problems however. Since Perl evaluates a scalar as either a string or a numeric value based on context, we have the situation where 0 in a string context `0` is a non-empty string. The rule here is that `0` is treated as false, even though it is a non-empty string, but `00` is treated as true, even though it is numerically equivalent to `0`. There are other odd cases related to strings that begin with a number.

Empty strings are clear false values. For numeric values, or strings that may evaluate to such things as `0` or `00`, it is safest to use a relational expression such as one of the following:

```
$a != 0
```

```
$a eq 00
```

The `if-else` statement

The basic control construct for two-way branching is the `if-else` statement. This construct was introduced in the first section, but will be reviewed here in more detail. The syntax was borrowed from the C language with a minor restriction:

```
if (expr) {
  statement 1;
  statement 2;
     . . .
}
else {
  statement 1;
  statement 2;
     . . .
}
```

The use of the `else` statement is optional. The restriction is that the `{ }` must enclose the statements following the `if` and `else` statements, even if there is only one statement. Programmers familiar with C will be aware that the `{}` are optional if there is only one statement. This is not the case with Perl.

The exact placement of the `{}` is left to the discretion of the programmer. The `if-else` statement works as follows:

- If the expression evaluates to a true condition, the statements in the statement block following the `if` statement are executed.
- If the expression evaluates to a false condition, the statements in the statement block following the `else` statement are executed, assuming the `else` statement exists.

The `elseif` Statement

The `if-else` statement can be nested within other `if-else` statements. The `elsif` statement is available as a shorthand replacement for an `if` statement nested within an `else` statement block. This allows the programmer to create multi-way branching constructs without deep nesting of `{ }` delimited blocks. In fact, this is the only way that multi-way branching constructs can be created without nesting, because Perl does not support the use of the case or switch statement that can be found in other languages.

Examples:

➤ Version 1: nested `if` statements:

```
if ($x < 0) {
  print("$x is negative\n");
}
else {
  if ($x == 0) {
    print("$x is zero\n");
  }
  else {
    print("$x is positive\n");
  }
}
```

➤ Version 2: use of `elsif` statement:

```
if ($x < 0) {
  print("$x is negative\n");
}
elsif ($x == 0) {
  print("$x is zero\n");
}
else {
  print("$x is positive\n");
}
```

THE warn(), die(), AND exit() FUNCTIONS

The `warn()`, `die()`, and `exit()` functions are not control constructs, but are often used in conjunction with them.

The `warn()` function is used to send a message to the standard error file without terminating the program. The function parameter is a string containing the message that is to be written to the file. After printing the message, the program continues with the next statement.

Syntax:

```
warn(message)
```

The format of the message is determined partially by the presence or absence of a newline character at the end of the message. If a newline character is appended to the message string, only the string is printed. If a newline is not appended, then the message is printed, followed by the file name and the line number at which the `warn()` function was executed.

Example:

```
warn("No newline character");
warn("Newline character\n");
```

Output:

```
No newline character at warn.pl line 1.
Newline character
```

The `die()` function behaves identically to the `warn()` function, except that the program terminates after printing the message to the standard error file. This includes the behavior corresponding to the presence or absence of a trailing newline character.

Syntax:

```
die(message)
```

The `exit()` function causes the program to terminate immediately with no message. Instead, an optional integer argument can be provided which is passed back to the calling process (often times a Unix shell) as a return code.

Syntax:

```
exit(return_code)
```

THE `while` LOOP

The `while` loop are entry-condition loops. That is, a condition must be met before the loop can be entered. Otherwise, it is bypassed.

Syntax:

```
while (expr) {
  statement1;
  statement2;
     . . .
}
```

With the `while` loop, if `expr` evaluates to true as the loop is reached, the loop is entered. The program stays in the loop as long as `expr` continues to evaluate to true at the end of each iteration. The loop terminates at the close of the loop where `expr` becomes false. You may reverse the sense of the test by putting an `!` in front of the boolean expression.

Be careful to make sure that the accompanying statement block contains code which can drive the loop into a termination condition. Otherwise, the program will stay in the loop forever, or until the whole program is aborted by the user.

THE for LOOP

The `for` loop integrates the three elements needed for most loops into a single construct:

1. An initialization statement which is executed one time prior to entering the loop.
2. A pre-iteration expression which is executed at the beginning of each iteration. If the expression evaluates to true, another iteration is started.
3. A post-iteration expression which is executed at the end of each iteration.

All three of the elements are optional. However, note that if the pre-iteration expression is omitted, the loop will continue until terminated externally or from within the statement block.

Syntax:

```
for (init; pre_expr; post_expr) {
  statement1;
  statement2;
     . . .
}
```

Example:

```
for ($i = 1; $i <= 5; $i++) {
  print("Iteration $i\n");
}
```

Output:

```
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
```

THE `foreach` LOOP

The `foreach` loop is a special form of the `for` loop which is designed to process lists. At this point in the course, a list can be considered to be a collection of items separated by commas. Later, list variables which can also be used with the `foreach` loop will be discussed. In fact, any function which generates a list as a return item can be used.

The `foreach` keyword is actually an alias for the `for` keyword, so either can be used depending on whether the emphasis is on readability or brevity.

Syntax:

```
foreach variable (list) {
  statement1;
  statement2;
     . . .
}
```

The loop is entered as it is encountered without initialization or entry-condition testing. At the beginning of the first iteration, the first item in the list is assigned to the variable and the iteration is processed. At the beginning of the second iteration, the second item in the list is assigned to the variable and the iteration is processed. This procedure continues until the last item in the list has been processed by an iteration.

Note that the variable defined in the construct is considered a local variable for the loop statement block. Any values assigned during the loop are defined only for the duration of each iteration. If a variable of the same name is assigned a value prior to the loop, that value will be restored after the loop is completed.

Example:

```
$var = "original";
print("$var\n");
foreach $var (1, 2, 3) {
  print("$var\n");
}
print("$var\n");
```

Output:

```
original
1
2
3
original
```

THE last STATEMENT

Often times it is more convenient to control your loop from a condition that occurs from within a loop than a condition available as we we enter the loop.

The `last`, `next`, and `redo` statements generalize the control of the loop by allowing the placement of a test condition to control the flow of the loop, anywhere inside the loop. This can be done at the begining, at the end, or from the middle. These provide a very general way to control loops addressing potentially difficult coding problems.

The `last` statement is used to terminate a loop (not the program). When the `last` statement is executed, processing within the loop ceases and control is transfered to the next statement after the close of the loop statement block. `last` provides a convenient way to terminate a loop from within the statement block.

Example:

```
while (1) {
  $in = <STDIN>; # Read a line
  if ($in eq "") { last; }; # Quit if EOF
  print($in); # Otherwise, print the line
}
```

The `next` statement is used to terminate the current iteration of the loop by forcing the program flow to the bottom of the statement block. The test expression at the begining of the loop is then evaluated and if the conditions for staying in the loop are met, the loop continues with the next iteration.

Example:

```
while ($num++ < 6) {
  if  ($num % 2) { next; }
  print("$num");
}
```

The `redo` statement forces the current iteration to restart, without revaluating the test expression of a `while` or a `for` loop. The use of `redo` is sometimes very convenient for handling some very specific difficult programming problems, which require peeking ahead at upcoming data that a loop test may provide, without consuming it. This, however, occurs rarely. Use of the `redo` statement should be avoided for most programming problems since it makes it more difficult to follow the flow of the program

Example:

```
$in = <STDIN>;
chomp($in);
while ($in ne "") {
  $last = chop($in);
  print($last);
  redo unless ($in eq "");
  print("\n");
  $in = <STDIN>;
  chomp($in);
}
Input:
abcde
fghij
```

Output:

```
edcba
jihgf
```

THE MANY WAYS TO DO SOMETHING

In general, Perl provides many ways to do anything. This chapter has covered a basic set of control constructs that should be sufficient to write any

program. However, there are other control constructs to learn about to increase knowledge of Perl programming. For instance the statement:

```
if (! boolean_expression) { ... }
```

can be replaced by:

```
unless ( boolean_expression ) { ... }
```

This statement:

```
while (! boolean_expression) { ... }
```

can be replaced by:

```
until (! boolean_expression) { ... }
```

Also, there are shortcuts to express various control constructs, such as the form:

```
statement if boolean_expression
```

or

```
statement while boolean_expression
```

The && and || operators are sometimes used as a brief kind of `if` statement, as shown by the following example:

```
open(IN, $a_file) || die("Cannot open file $a_file
     for reading\n");
```

In the above example, if the `open file` function fails to return a `false`, the `die` function gets invoked. If not, the `die` function is not invoked. Thus, the short-circuit evaluation nature of the && and || allow it to function as a type of `if` statement.

A basic set of very general flow control commands sufficeint to address all programming needs have been reviewed thus far. As proficiency in programming increases, additional Perl flow control commands can be learned to provide other possible ways to write code.

LESSON SUMMARY

- Numerical values equate to boolean true if nonzero and boolean false if zero.
- The `if-else` and `unless-else` statements are used to implement two-way branching.
- The `elsif` statement can be used to implement multi-way branching without excessive indentation.
- The `while` and `for` loops are entry-condition loops.
- The `if`, `unless`, `while`, and `until` statements have single-line forms.
- The `foreach` loop is another form of the `for` loop which is used for processing lists.
- The `warn()` and `die()` functions can be used to send messages to the user through standard error.
- The `exit()` function causes the program to terminate immediately.
- The `last`, `next`, and `redo` statements control program flow within a loop.

REVIEW QUESTIONS

1. What is the value of `$i` in the following program?

```
$i = 5; ($i < 3) && ($i = 5)
```

2. What is the purpose and what are the differences between the `last`, `next`, and `redo` statements?

3. How do `print`, `warn` and `die` differ?

4. Which of the following statements produce the message `hello`?

```
$i = "1"; if ($i) { print("hello\n"); }
$i = "0"; if ($i) { print("hello\n"); }
$i = 0; if ($i) { print("hello\n"); }
$i = ""; if ($i) { print("hello\n"); }
$i = " "; if ($i) { print("hello\n"); }
$i = "00"; if ($i) { print("hello\n"); }
$i = 1; print "hello\n" if $i;
$i = "0"; if ($i) print("hello\n");
```

Answers on page 201

EXERCISE

1. Write a program which asks the user what language they would like to be greeted in. Give them a choice of English (Hello), Australian (G'day), Spanish (Buenos dias), or French (Bon jour). After making a selection, print out the corresponding greeting to standard output. Use the `die()` function to terminate the program with a message if the user selects an unknown language code.

2. Modify the program in exercise 1 to greet a list of people (John, Duncan, Hector, and Rene) hardcoded into a `foreach` list with a language code following each name. Greet each person on a separate line of `standard output` with the appropriate greeting. Use the `warn()` function to notify the user if an unknown language code is specified.

3. Write a program which generates a Fibonacci series of a length specified by the user. The Fibonacci series starts with 1 and 1 and always creates the next number in the series by summing the previous two. The series begins with:

```
1, 1, 2, 3, 5, 8, ...
```

LESSON 5

Working with Scalars

OVERVIEW

Scalar variables in Perl provide the programmer with a lot of flexibility. These variables automatically convert between integer, double precision floating point, and string form based on the context of usage. This lesson, covers scalar variables and how to use them, as well as the special default, special scalar variable ***$_****.*

LESSON TOPICS

- Scalar Variables
- Integer Scalar Values
- Floating Point Scalar Values
- Character Strings
- Escape Sequences
- Conversions Between Numbers and Strings
- Initial Values
- Explicit Conversion Between Numbers and Strings
- The $_ Variable

OBJECTIVES

By the end of this lesson, you should be able to:

- Create and identify scalar variables.
- Work with integer constants using either decimal, octal,or hexadecimal notation.
- Work with floating point values.
- Manipulate strings.
- Understand `single-`, `double-`, and `escape` quoting mechanisms.
- Understand the use of the $_ variable as used with several common functions.

SCALAR VARIABLES

Perl uses a leading $ character for variables which hold a single unit of data. This can be an integer value, a floating point value, or a string.

In addition, data items held in variables are converted between the three types automatically, depending on the context in which a scalar variable is used.

INTEGER SCALAR VALUES

An integer scalar value consists of one or more digits, optionally preceded by a + or - character and optionally containing underscores.

Examples:

```
25
```

```
-19
```

```
+457
```

```
1_234_567_890
```

Integers are actually stored as floating point numbers and displayed automatically as integers. This means that on most machines (32-bit registers), Perl can hold approximately 16 digits of precision.

This is consistent with most programming languages which have an upper integer limit of 4294967295 or (232 - 1).

By default, numbers are specified using decimal notation. Integers can also be specified in terms of octal (base 8) or hexadecimal (base 16) notation.

If an integer is specified with a leading zero, Perl assumes the number is being specified as an octal number.

If an integer is specified with a leading `0x`, the number is assumed to be in hexadecimal notation. Either upper-case or lower-case characters `a-f` can be used to represent the numbers 10-15.

Examples:

```
0325 Octal
```

```
0x94ae7 Hexadecimal
```

FLOATING POINT SCALAR VALUES

In Perl, a floating point value consists of:

- a minus sign (optional)
- a sequence of digits with an optional decimal point (can also include `_` characters)
- an optional exponent

Examples:

```
22.5
```

```
-46.1
```

```
0.275
```

```
12.6592+e03
```

```
0.3429-E02
```

The exponent has the effect of multiplying the value by the specified power of 10. For example, the notation `e02` means that the number will be multiplied by 102 or 100. The notation, `e00` is legal, but has the effect of multiplying the number by 1.

Programmers should be aware that just as with other languages, Perl is susceptible to round-off errors through combinations of operations involving very large and very small numbers.

CHARACTER STRINGS

Data values can also be stored as text, or character strings. In previous examples, strings have been bounded by double quotes, but quotes are only necessary for strings if they contain whitespace. However, it is never a bad idea to use quotes to bound strings, even when not technically necessary.

Perl supports scalar variable substitution for character strings enclosed in double quotes. That is, the value of a scalar variable is substituted at the same position in the string where the variable itself is placed.

Example:

```
$name = "John Smith";
print("Hello, $name!\n");
```

Output:

```
Hello, John Smith!
```

Strings that are bounded by single quotes do not support scalar variable substitution. Instead, the literal text is used, even if it would normally be recognized as a special escape character, such as the newline character.

Example:

```
$name = `John Smith';
print(`Hello, $name!\n');
```

Output:

```
Hello, $name!\n
```

Single quoted strings can also be specified over multiple lines. In this case, newline characters are inserted in the string at the line breaks.

Example:

```
$string = `Line 1
Line 2';
```

This is equivalent to:

```
$string = "Line 1\nLine2";
```

ESCAPE SEQUENCES

Strings that are bounded by double quotes can use escape sequences for special characters. An escape sequence is a backslash \ followed by one or more characters. The \n escape sequence representing the newline character had been used in previous examples.

Table H lists escape sequences recognized by Perl:

TABLE H. *Escape Sequences Recognized by Perl*

`\a`	Bell
`\b`	Backspace
`\cn`	The Ctrl+n character
`\e`	Escape
`\E`	Terminates \L, \U, or \Q
`\f`	Form feed
`\l`	Next letter lower case
`\L`	Characters up to next \E are lower case
`\n`	Newline
`\r`	Carriage return
`\Q`	Ignore special characters
`\t`	Tab
`\u`	Next letter upper case
`\U`	Characters up to next \E are upper case
`\v`	Vertical tab

A character can also be specified in terms of its equivalent octal notation using the `\nnn` syntax. Also, hexadecimal notation can be used with a `\xnn` syntax.

Example:

```
M77(decimal\115 (octal)x4D (hexadecimal)
```

CONVERSIONS BETWEEN NUMBERS AND STRINGS

As stated earlier, values in Perl are context oriented. That is, a string that looks like a number can be used as a number. Perl will automatically do whatever conversion is needed.

Example:

```
$str = "52";
$num = 18;
print($str + $num, "\n");
print($str . $num, "\n");
```

Output:

```
70
5218
```

If a string consists entirely of characters that are not digits, the string is converted to a zero when used in a numerical context. This is not considered an error, so Perl does not notify the user when this occurs.

If the string begins from the left-hand side with numbers, Perl will convert the string from left-to-right until it reaches the end of the string or a non-numeric character.

Examples:

```
$str1 = "17";# Will convert to integer 17
```

```
$str2 = "abc";# Will convert to integer 0
```

```
$str3 = "12ab";# Will convert to integer 12
```

INITIAL VALUES

Variables in Perl do not have to be initialized. By default, variables are initialized to a `null` string. When used in a numerical context, the value is converted to an integer zero.

Examples:

```
print("Hello" . $var . "\n"); # $var is null
```

```
print(25 + $var, "\n"); # $var is zero
```

Output:

```
Hello
```

```
25
```

EXPLICIT CONVERSION BETWEEN NUMBERS AND STRINGS

The following functions can be used to make some explicit conversions between numbers and strings, and vice-versa.

- `chr(NUM_EXPR)` is a function that return the ASCII character that is represented by the number passed as a parameter. For example, `chr(0x2b)` will return a + sign. Most Unix systems include an ASCII table in their online documentation.
- `ord(STRING_EXPR)` is a function that return a numeric value for the 1st ASCII character given by the parameter.
- `STRING_EXPR`. `hex(STRING_EXPR)` is a function that returns a number corresponding to the hexadecimal string passed.

When a constant in a program of the form `0xa123` is used, the Perl interpreter automatically converts that to the corresponding value. But when the string `0xa123` is used in a numeric context, Perl does not make this conversion automatically. Use the `hex` function to make this conversion.

Examples:

```
print hex '0xAe'; # prints '174'
```

```
print hex 'aE';   # prints the same.
```

THE $_ VARIABLE

Of all the system variables available to Perl programmers, the $_ variable is used by more functions than any other. It is the default input buffer for many commonly used functions and operators.

- the `chomp()` function

  If no parameter is given, any trailing newline characters are removed from the contents of the $_ variable.

- the `chop()` function

  If no parameter is given, the last character is removed from the contents of the $_ variable.

- the <> operator in `while` or `for` loops

  If the input line is not assigned to a scalar variable, it is assigned to the $_ variable.

- the `print()` function

  If no parameter is given, the contents of the $_ variable are printed.

- the substitution operator

  If no string is supplied, the contents of the $_ variable are used.

- the translation operator

  If no string is supplied, the contents of the $_ variable are used.

Example:

```
while (<>) {        # Read a line into $_
  chomp;            # Remove any newlines
  s/these/those/g;  # Substitute "those" for "these"
  tr/a-z/A-Z/;      # Convert to upper case
  print;            # Print the line
}
```

LESSON SUMMARY

In this lesson, you have learned:

- Scalar variables begin with a leading $ character.
- Integer constants can be expressed in either decimal, octal, or hexadecimal notation.
- All numbers are actually stored as floating point numbers.
- Floating point numbers can also be expressed using exponential notation.
- Double quoted strings allow scalar variable substitution.
- Single quoted strings specify literal characters.
- Escape sequences can be used to specify special characters.
- Conversion between strings and numbers occurs automatically, depending on context.
- Variables are automatically initialized to a `null` string which is converted to an integer zero when used in a numerical context.
- $_ is a default variables for many functions that can often go unmentioned.

REVIEW QUESTIONS

1. What will be printed by the program below?

```
$b = "Betty";
$a = "Anna";
print("1. $a and $b are friends\n");
print('2. $a and $b are friends\n');
```

2. What is the meaning of these characters:

   `\n`

   `\r`

   `\a`

3. For the program:

```
$a = 3.5;  $b = "6";  $c = $a + $b;
```

a. What is the result?

b. What type do the values get converted to, prior to executing the operation?

4. What does this program do?

```
while (<>) {
chomp; chop;
print($_);
};
```

Answers on page 202

EXERCISE

1. Write a program that will accept a floating point number with a decimal point and truncate it to an integer. Hint: use the `chop()` function.
2. Write a program that will accept an octal integer and display it as a decimal number.
3. Write a program which accepts a string and displays it both as all lower case and then as all upper-case.

LESSON 6

Working with Lists

OVERVIEW

Lists facilitate the manipulation and collection of data. This lesson covers the variety of ways in which Perl lists are created, accessed, and modified.

LESSON TOPICS

- What is a List?
- Storing Lists as Constants
- Defining and Using List Variables
- Working with Lists
- Manipulating Lists
- Lists and Strings
- Reading a List from Standard Input
- Using chomp() and chop() with Lists

OBJECTIVES

By the end of this lesson, you should be able to:

- Define and create lists.
- Store lists in variables.
- Access list elements.
- Use list range and slice operators.
- Sort a list.
- Reverse the order of a list.
- Convert between a list and a string

WHAT IS A LIST?

In Perl, a list is a set of scalar values. The items in the set can be any legal scalar entity: strings, integers, or floating point numbers. Elements of the list can be referenced and acted upon. Also, lists can be used as a single entity, referring to the collection of all the items in the list. This makes it convenient to pass groups of scalar values between functions.

Also recall that the `foreach` (`for`) control construct can be used to process a list element by element.

Example:

```
foreach $num (1, 2, 3, 4, 5) {
  print("$num\n");
}
```

Output:

```
1
2
3
4
5
```

STORING LISTS AS CONSTANTS

Lists can be used to store groups of elements of any data type. Lists do not have to be associated with a variable name, but can be established as a list constant.

To create a list constant, place a series of data items in parentheses, separated by commas.

Example:

```
(abc, def, ghi)# A list of strings
```

```
(1, 2, 3, 4, 5)# A list of integers
```

```
(12.2, 5.7, 8.04)# A list of floating point numbers
```

```
("Hi", 12, 0.24)# A mixed list
```

List elements can also include variables when they are being created. For now, think solely in terms of scalar variables, even though other types of variables can be used as well, including other list variables.

Example:

```
($var1, 5, $var2, xyz)
```

Note that the use of such variables establishes a constant value for the list elements it is initializing at the time at which the statement containing this

list is executed. If the value of the variable changes later, the corresponding list element is not changed.

DEFINING AND USING LIST VARIABLES

As discussed earlier, a list can also be stored in a special variable type. A list variable is any legal Perl symbol name, preceded by a @ character.

Recall that symbol names are allowed to begin with an underscore character _ or an alphabetic character, optionally followed by a combination of underscore and alphanumeric characters. Because of the leading @ character, there is no conflict between scalar variables and list variables having the same symbol name. Often times, a list is given the same name as a scalar, such as `$list1` and `@list1`. It is important that you be able to make the distinction between these two completely different entities.

Example:

```
$list# Scalar variable
```

```
@list# List variable
```

Lists can be created several different ways. The easiest way is to assign a constant list to a list variable. Note that both scalar variables and list variables can be used to create a list. We will discuss other methods later in this section.

Example:

```
$x = 1;
$y = 2;
@num1 = ($x, $y);
@num2 = (3, 4);
@num3 = (@num1, @num2); # (1, 2, 3, 4)
@rhyme = (@num1, "buckle", "my", "shoe");
```

WORKING WITH LISTS

Accessing Individual List Elements

Individual list elements are accessed by using the `[]` or indexint operator. Since list elements are scalars, a leading `$` is used with the list symbol instead of a leading `@` when an individual element is indexed.

By default, lists use zero-based indexing. That is, the first element of a list is designated with index `0`, the second with index `1`, and so on.

Example:

```
@notes = ("do", "re", "mi");
print("$notes[0] $notes[1] $notes[2]\n");
```

Output:

```
do re mi
```

Scalar variables containing integer values can also be used instead of literal index numbers. In fact, any expression that will resolve to an integer value can be used.

Example:

```
$index = 3;
@list = ("a", "b", "c", "d");
print("$list[$index - 2]\n");
```

Output:

```
b
```

Note also that it is not an error to access a *non-existent* list element. Perl simply returns a `null` value, or zero value, depending on the context in which it is used. Remeber that you must use a `$` when referring to a scalar element of a list. However, keep in mind that `$list1[1]` is an entity referring to one scalar element of `list @list1`. That bears no connection to a scalar hav-

ing the same name, such as `$list1`. Scalar values cannot be indexed. Therefore, these two values are unambiguously distinguihed by the Perl language.

Assigning List Elements to Other Variables

List elements can also be assigned collectively to both scalar and list variables. Entire lists can be copied from one list variable to another by using the assignment operator.

Example:

```
@list1 = (1, 2, 3);
@list2 = @list1;
print("@list2\n");
```

Output:

```
1 2 3
```

If a list consists solely of variables, it can be used as a value that can be assigned to. For instance:

```
($a, $b, $c) = (1, 2, $d);
```

The *left-hand side* (*lhs*) of an assignment statement is often referred to as an *lvalue*, and contains the values being assigned to. When assigning a list to a list as in the above example, the elements of the lvalue are assigned the elements of the list on the *right-hand side* (*rhs*) of the equal sign on a one-to-one basis. If there are more lhs elements than rhs elements, then any unmatched elements on the lvalue list are given a null value.

If there are more rhs elements than lhs elements, the right-most element on the rhs will not be assigned to an lhs element.

The lvalue list can also contain a list variable, such as:

```
@list1 = ($a, $b, $c)
```

Assignment is still from left-to-right. Upon execution of such an assignment statement, all elements on the rhs will be the values contained in the list variable of the rhs.

If the lvalue list consists of two list variables, such as the following, all of the rhs elements will be assigned to the leftmost list variable and nothing will be assigned to the rightmost list variable.

```
(@l1, @l2) = ($a, $b, $c)
```

Right-hand side list elements can be constants, variables (including list variables), or expressions.

Examples:

The comments in the following exampes indicate the results of the statements.

```
($var1, $var2) = (1, 2); # $var1 = 1, $var2 = 2
```

```
($var1, $var2) = (1, 2, 3); # $var1 = 1, $var2 = 2
```

```
($var1, $var2, $var3) = (1, 2);
# $var1 = 1, $var2 = 2, $var3 = null
```

```
($num, @list) = (1, 2, 3);
# $num = 1 and @list = (2, 3)
```

```
(@list1, @list2) = (1, 2, 3);
# @list1 = (1, 2, 3) and  @list2 = (null)
```

```
@list1 = @list2;
# elements of list1 are same as elements of list2
```

Retrieving the Length of a List

We have stated earlier that Perl is context oriented with regard to its data types. This is true with lists as well.

If a list or list variable is used in a scalar context, the length of the list is used instead of its elements. This makes it easy to obtain the length of any list.

Example:

```
@list = (1, 2, 3, 4, 5);
$len1 = @list;
$len2 = (a, b, c);
print("$len1\n");
print("$len2\n");
```

Output:

```
5
3
```

Note from the previous page that simply printing out the list variable itself would not give the same result. However, the scalar() function can be used with a list to force a scalar context, meaning the length of the list.

Example:

```
@list = (1, 2, 3, 4, 5);
print(@list, "\n");
print(scalar(@list), "\n");
```

Output:

```
12345
5
```

Command Line Arguments

Command line arguments are brought into a Perl program through a special system list variable called @ARGV. This list contains the command line argu-

ments in sequence as they were given from the command line. The first element does not contain the command name. That is stored in another system scalar variable called `$0`.

Command line arguments can be assigned to individual scalar variables or processed using a `foreach` loop.

Examples:

```
($arg1, $arg2) = @ARGV;
print("The 1st argument is: $arg1\n");
print("The 2nd argument is: $arg2\n");
```

```
foreach $arg (@ARGV) {
  print("$arg\n");
}
```

Input:

```
$ myprog one two three
```

Output:

```
The 1st argument is: one
The 2nd argument is: two
one
two
three
```

Adding and Removing Elements at Either End

There are four functions which can be used for adding and removing list elements at either end of a list.

The `shift()` function removes an element from the left end of a list. The argument is the list variable and the return value is the element which was removed.

Syntax:

```
element = shift(@list)
```

The `unshift()` function adds an element(s) to the left end of a list. The function parameters are the list variable and one or more elements to be added. The return value is the number of elements in the updated list.

Syntax:

```
count = unshift(@list, element, ...)
```

The `pop()` function removes an element from the right end of a list. The argument is the list variable and the return value is the element which was removed.

Syntax:

```
element = pop(@list)
```

The `push()` function adds an element(s) to the right end of a list. The function parameters are the list variable and one or more elements to be added. The return value is the number of elements in the updated list.

Syntax:

```
count = pop(@list, element, ...)
```

Example:

```
@list = (1, 2, 3);
while ($i < @list) {
  print(push(@list, unshift(@list)), "\n");
}
```

Output:

```
231
312
123
```

MANIPULATING LISTS

Sorting a List

There are several built-in functions which are used to manipulate lists. One of the more useful is the `sort()` function. This function is used to sort the elements of a list in ascending order. It can be applied to lists of numbers or strings. If the list has a mixed set of elements (numbers and strings), all elements will be treated as strings and the list will be sorted in ascending ASCII order.

The list to be sorted is buffered during the sort process. This means that the same variable can be used on both sides of an assignment operator if the intent is to sort the contents of a list variable and then use the same variable to hold the sorted list.

Example:

```
@list1 = (4, 7, 3, 2, 5);
@slist1 = sort(@list1);
@list2 = ("xyz", "abc", "qrs", "ijk");
@list2 = sort(@list2);
print("slist1\n");
print("list2\n");
```

Output:

```
2 3 4 5 7
abc ijk qrs xyz
```

Reversing the Order of a List

The `reverse()` function is used to reverse the order of the elements in a list. It can also be used with the `sort()` function to provide a reverse sorted list.

Examples:

```
@list = (3, 5, 4, 6, 2);
@rlist = reverse(@list);
@rslist = reverse(sort(@list));
print("@list\n");
print("@rlist\n");
print("@rslist\n");
```

Output:

```
3 5 4 6 2
2 6 4 5 3
6 5 4 3 2
```

LISTS AND STRINGS

Converting a List to a String

The `join()` function is used to convert a list to a string. It takes two arguments:

- a string separator
- a list

If a null string separator is used, the resulting string is formed from the list elements without separators between them.

The list argument can be any list constant, list variable, or any function or expression which produces a list as a return or resulting value.

Example:

```
@list = ("How", "now", "brown", "cow");
$string = join(" ", @list);
print("$string\n");
```

Output:

```
How now brown cow
```

Converting a String to a List

A string can be converted to a list with the `split()` function. This function also takes two arguments:

- a pattern (actually, a regular expression) enclosed in `//` characters
- a string

The pattern specifies the separation points where the string is to be divided into individual list elements. The pattern is not used as part of the list elements.

If a null pattern is specified, the list consists of a sequential list of characters that comprised the string.

Examples:

```
$string = "root:x:0:0:Admin:/:/usr/bin/ksh"
@acct = split(/:/, $string);
# use of /:/ splits the items between the : characters
($logid, $UID, $GID) = @acct[0, 2, 3];
@letters = split("", "abcde");
# use of null pattern splits every character
print("$logid\n$UID\n$GID\n");
print("\n@letters\n")
```

Output:

```
adm
0
0

a b c d e
```

READING A LIST FROM STANDARD INPUT

The syntax for reading a line from standard input was presented in an earlier section. If desired, the entire file can be read into a list variable using the same syntax. Each line in the file becomes a single item in the list variable.

Example:

```
@input = <STDIN>;
while ($i < @input) {
  print("$input[$itt]");
}
```

Input:

```
Line 1
Line 2
Line 3
```

Output:

```
Line 1
Line 2
Line 3
```

USING chomp() AND chop() WITH LISTS

Recall that when the `chomp()` function is applied to a scalar variable, any newline character at the end of the value is removed. Likewise, when the `chop()` function is used, the last character of the value is removed.

When the same functions are applied to a list variable, the same actions are taken against every element in the list. That is, the `chomp()` function removes any newline characters from the end of every element. The `chop()` function removes the last character from every element.

Example:

```
@input = <STDIN>;# Read the file into a list
chomp(@input); # Remove newline characters
chop(@input);# Remove last character or each element
print(@input,"\n"); # Print the list on one line
```

Input:

```
1a
2b
3c
```

Output:

```
123
```

LESSON SUMMARY

In this lesson, you have learned:

- A list is a set of values treated as a single entity.
- Lists can be created from constants, variables, and expressions.
- The `[]` operator is used to access individual list elements as a scalar variable.
- Command line arguments are available through the `@ARGV` system variable.
- The `sort()` function is used to sort a list.
- The `reverse()` function is used to reverse the order of a list.
- The `join()` function is used to convert a list to a string.
- The `split()` function is used to convert a string to a list.
- The `shift()`, `unshift()`, `push()`, and `pop()` functions are used to add and remove elements from the end of a list.
- The `splice()` function is used to add, replace, and delete elements of a list.
- Reading a file into a list variable places the entire file in the list, one line per element.
- The `chomp()` and `chop()` functions are applied to every element in a list.

REVIEW QUESTIONS

1. Write two programs that will take the list below and concatenate "is an animal," to each string in the list.

   For the first program, use `foreach`. For the second program, use a `for` loop.

```
@an = ("dog," "cat," "rabbit");
```

2. Sort the list and store the result in an array `@sorted_animals;`

3. Write a program to merge all the strings in the sorted list, while putting a `;` between the elements.

4. Take the resulting strings and write a program to split them up again, removing the `;`.

Answers on page 202

EXERCISE

1. Write a program that accepts a `-l` or `-u` option and reads the standard input file. If the `-l` option is given, print the input to standard output in all lower case. If the `-u` option is given, print the input to standard output in all upper case. If no option is given, print out the file as is.
2. Write a program which reads a file containing a multi-digit integer number on each line. Determine how many times each digit appears in the file.
3. Write a program which reads a text file (no numbers or punctuation) and lists the words used in alphabetical order. Omit any duplicates.
4. Implement a simple four-function Reverse Polish Notation calculator using the `push()` and `pop()` functions. Include the use of `x` to exit and `c` to clear the previous entry. Provide a simple prompt for the user.

Example:

```
> 10
> 20
> +
> =
30
>
```

LESSON 7

Working with Hashes

OVERVIEW

A very powerful and unique feature of the Perl language is the use of built-in hash tables. These are also often referred to as symbol tables or associative arrays. This language facility allows for the creation of arrays that can be indexed into strings instead of numbers. This lesson covers how to use these very powerful constructs. It also discusses %ENV, a very important table which is frequently used in writing CGI scripts.

LESSON TOPICS

- What are Hashes?
- Creating Hashes from Lists
- Adding and Deleting Hash Elements
- Testing for the Existence of a Hash Element
- Getting a List of Hash Keys
- Getting a List of Hash Values
- Looping Through Hash Elements
- The %ENV Hash

OBJECTIVES

By the end of this chapter, you should be able to:

- Define and create hashes.
- Reference hash elements.
- Add hash elements.
- Delete hash elements.
- List hash keys and values.
- Understand the uses of the `%ENV` environment hash table.

WHAT ARE HASHES?

Indexed arrays, or lists, are useful in that sets of values can be stored in a common variable in a manner which makes it easy to retrieve individual elements. The problem with indexed arrays is that the index used to reference the value has no meaning with regard to the value itself. In order to reference a particular value, the user has to remember, or be able to look up, the particular index which identifies the list element where the value is stored.

*Hashe*s, or *associative arrays*, are special list types which use a string as an index instead of a number. This string index is called *a key.* The key is used to reference an individual value.

The downside of hashes is that any concept of sequence is lost. A `foreach` loop can be used to process every element in a hash. Elements of a hash cannot be retrieved in any particular order, even the order in which the hash was created.

Associative arrays are called hashes because the values are stored in a data structure called a *hash table*. Details as to how hash algorithms work are beyond the scope of this course and are not needed to use hashes within the context of a Perl program.

Hash Variable Names

Hash variable names begin with the % character followed by any legal Perl symbol name. Recall that symbol names may begin with an underscore or alphabetic character optionally followed by one or more underscore or alphanumeric characters.

Examples:

```
%city
```

```
%Q1_1997_Dept_Total_Expenses
```

Referencing Hash Elements

Hash elements are referenced by using the {} operator instead of the [] operator used with indexed arrays. As with indexed arrays, individual hash elements are scalar values, so a leading $ character is used with the variable name instead of the % character.

The index is a string instead of a number. If numbers are used, they are still used in a string context.

Note that if an element is referenced with a key that is not associated with a value, a null value will be returned.

Examples:

```
$string3 = $City{"Bob Smith"};
```

```
$Total_Expenses += $Dept_Expenses{"Training");
```

```
$name = $First{"32865"} . " " . $Last{"32865"};
```

CREATING HASHES FROM LISTS

Hashes can be created using a list syntax. The odd elements in the list become the keys, or string indexes, and the even elements become the values. The hash name still begins with a % character even though the assignment is from an indexed list.

The right-hand side list can be a list constant or list variable, or any function which produces a list as a return value.

Examples:

```
%First = ( "32512", "Bob",
           "33761", "Karen",
           "32177", "George",
           "32956", "Jean");
```

```
%Inventory = @item_list;
```

```
%count = split(/ /, join(" ", @file));
```

Alternate Notation for Hash Elements

The normal list syntax can be difficult to read, especially if many elements of a hash are defined on each line. For that reason, an alternate notation is available which uses a => operator instead of a comma operator to separate each key and value pair.

Examples:

```
%First = ( "32512" => "Bob",
           "33761"=> "Karen",
           "32177"=> "George",
           "32956"=> "Jean");
```

Adding and Deleting Hash Elements

Hash elements are easily added to a hash. Just use the hash element notation with an assignment operator. If an element already exists with the same key, it's value will be overwritten with the new value. Just remember that a hash element is a scalar value, so the variable requires a leading $ character.

If an element is added to a hash that did not previously exist, it will be created.

Example:

```
%State = ("Bob Smith" => "MA"; # List notation
$State{"Betty Davies"} = "TX"; # Element notation
```

The `delete()` function is used to remove elements from a hash table. In fact, this is the only way that elements can be removed from a hash. Remember that the concept of sequence is lost with an associative array, so trying to use functions like `shift()` and `pop()` will not work.

The function takes a single argument; the hash element in element notation referencing the element to be removed.

Example:

```
delete($City{"Bob Smith"});
```

Testing for the Existence of a Hash Element

The `exists()` function can be used to test for the existence of a particular hash element. The function takes a hash element as its only parameter and returns a boolean true if the element exists. Otherwise, it returns a boolean false.

Example:

```
if (exists($City{"$empno"})) {
  print("Element for $empno exists\n");
}
else {
  print("Element for $empno does not exist\n");
}
```

GETTING A LIST OF HASH KEYS

Even though the concept of sequence is lost with a hash, the `keys()` function can be used to create an indexed list of the hash keys. Being a list, the elements can then be sorted to create a list of keys in alphabetical order.

Once the keys have been obtained, the values for each key can be obtained by using the `{}` operator.

The `keys()` function takes a hash as its only parameter.

Example:

```
%First = ( "32512" => "Bob",
           "33761"=> "Karen",
           "32177"=> "George",
           "32956"=> "Jean");
@list = sort(keys(%First));
foreach $empno (@list) {
  print("$empno\t$First{"$empno"}\n");
}
```

Output:

```
32177       George
32512       Bob
32956       Jean
33761       Karen
```

GETTING A LIST OF HASH VALUES

If a list of hash values is needed without having to have the keys associated with them, the `values()` function can be used to create such a list. The `values()` function takes a hash as its single parameter and returns an indexed list of values.

Again, the concept of sequence is lost with a hash, so there is no way to predict the order in which the values will be extracted from the hash.

Examples:

```
%First = ( "32512" => "Bob",
           "33761"=> "Karen",
           "32177"=> "George",
           "32956"=> "Jean");
@values = values(%First);
print("@values\n");
```

Output:

```
George Bob Jean Karen
```

LOOPING THROUGH HASH ELEMENTS

Once a list of keys has been obtained, it is easy to loop through the list and find the values associated with each key. Another way of looping through a hash is to use the `each()` function with a while loop to obtain sets of key-value pairs as a two-item list. The `each()` function takes a hash as its only parameter and returns a two-item list consisting of a hash key and its value.

When used in a loop, every separate call to the `each()` function will produce another key-value pair until every pair has been extracted. The pairs will be extracted in an unpredictable order.

When in a loop which uses the `each()` function, do not add or delete elements to the hash. This would cause unpredictable results if it were done.

Example:

```
%First = ( "32512" => "Bob",
           "33761"=> "Karen",
           "32177"=> "George",
           "32956"=> "Jean");
while (($empno, $name) = each(%First)) {
  print("$empno\t$name\n");
}
```

Output:

```
33761      Karen
32965      Jean
32177      George
32512      Bob
```

THE %ENV HASH

The %ENV hash lists all of the environment variables that are available to the program. The environment variable name is used as the key. Individual values can be obtained by specifying the variable name as the index into the hash.

Note that while values can be changed in the hash, not all changes are honored by the shell or the underlying operating system. For example, LOGNAME or USER reflects the actual user identity and is kept separately by the system. Changing the value in the %ENV hash will not cause the system to update it's table with the new value.

Examples:

```
$ENV{'EDITOR'} = 'vi';
print("$ENV{TERM}\n");
```

Output:

```
vt100
```

LESSON SUMMARY

In this lesson, you have learned:

- Hashes, or associative arrays, are lists which use strings as indices rather than numbers.
- Hashes can be created directly from indexed lists.
- The `=>` operator can be used to improve readability when creating a hash from a list.
- The `{}` operator is used to address individual hash elements.
- The `delete()` function is used to remove elements from a hash.
- The `exists()` function is used to test for the existence of a hash element.
- The `keys()` function is used to create an indexed list of hash keys.
- The `values()` function is used to create an indexed list of hash values.
- The `each()` function is used to retrieve key-value pairs from a hash, one pair per call.

REVIEW QUESTIONS

Answer the questions below with reference to the following table:

```
%a = (rabbit => "hops", dog => runs, lion => runs,
   kangooroos => 'run');
```

1. Write a statement that deletes the `rabbit` entry.

2. Write a statement that changes the `kangoroo` entry's value from `run` to `hops`.

3. Write a one line statement that collects all the animal names that hop into a list, `@hoppers`.

4. Is it possible to write a program that will retrieve the keys in the order given from just this data structure. Justify your answer.

Answers on page 203

EXERCISE

1. Write a program that will read a series of last names and phone numbers from standard input. The names and numbers should be separated by a comma. Then print the names and numbers alphabetically according to last name.

2. Write a program which will read text from standard input (no punctuation or numbers) and print out a sorted list of words and the number of times each occurs in the text.

3. Write a program that will read a series of employee numbers and daily work hours from standard input, one set per line. The employee number and the hours worked should be separated by a space. Use hashes and calculate the total number of hours worked and the average number of hours per work period. Print out a report by sorted employee number, the number of work periods, the total hours worked, and the average number of hours per work period. Assume that some of the employees are on a part-time schedule and do not work the same number of days or hours as regular employees.

LESSON 8

Reading, Writing and Manipulating Files

OVERVIEW

When a program ends, all traces of data calculated disappear; what remains behind are the files created or modified. This lesson covers how to use functions for the creation, modification, and deletion of files, as well as other file related subjects.

LESSON TOPICS

- Using Redirection from the Command Line
- Opening a File
- Reading Lines from a File
- Opening a File for Write and Append
- Closing a File
- The printf()and sprint() functions
- Controlling Output Buffers
- The rename() and unlink functions
- Using File Test Operators

OBJECTIVES

By the end of this lesson, you should be able to:

- Open a file for an input or output operation.
- Determine file status.
- Read from or write to a file.
- Use the `printf()` function for formatted output.
- Use the `sprintf()` function to do formatted output to files or into strings.
- Control automatic buffer flushing.
- Use unlink and rename to remove and rename files.
- Use operators to get information about files.

USING REDIRECTION FROM THE COMMAND LINE

When a Perl program begins executing, there are normally three files that are already opened for use:

- Standard Input
- Standard Output
- Standard Error

These files are opened with the file handles `STDIN`, `STDOUT`, and `STDERR`, respectively. Both the Unix and Windows NT operating systems support command line redirection. This means that input and output can be

obtained from files other than the default files or devices associated with the standard I/O files by using the redirection operators. These operators are:

- < for read
- > for write
- >> for append

Example:

When executing a command line:

```
$ myprog < infile >outfile
```

Source code:

```
$line = <STDIN>;# Reads a line from infile
print("$line\n");# Writes to outfile
```

OPENING A FILE

Before a Perl program can make use of a file, it must first open that file. In the process of opening a file, a file handle is specified which is used to reference the open file. Files can be opened in one of three modes: *read, write,* and *append.*

- Opening a file for a *read* operation positions the file pointer at the beginning of the file. The file can be read from, but not written to.
- Opening a file for a *write* operation causes the file contents to be destroyed. The file pointer is positioned at the beginning of the file. The file can be written to, but not read from.
- Opening a file for an *append* operation positions the file pointer at the end of the file. The file contents are not destroyed. Additional data can be written to the end of the file, but the file cannot be read from.

A file can also be opened for both reading and writing. This allows the file pointer to be positioned at a specified place in the file. The programmer can specify either a read or write operation.

The `open()` function is used to open a file. It takes two arguments: a file handle and the pathname of the file to be opened. The file handle can be any legal Perl symbol and is not preceded by a special variable symbol. By convention, file handles are created with all upper case characters.

The pathname can be preceded by a character(s) which indicates the mode for which the file is to be opened.

The function returns a positive integer if the file was opened successfully and a zero otherwise. This will be interpreted as boolean true or false, respectively.

Syntax:

```
open(filehandle, [mode]pathname)
```

Example:

```
if (! open(INPUT, "$file")) { die("Can't open
$file\n"); }
```

READING LINES FROM A FILE

By default, a file is opened for a read operation if no other mode is specified. The file handle can then be used with the `<>` operator to read from the file either one line at a time into a scalar variable or collectively into a list variable. The syntax is the same as that which has been used to read from standard input using the `STDIN` file handle.

Note that the mnemonic form of `or` should be used in combination with the `open()` function and other functions such as `die()` due to precedence.

Examples:

```
open(FILE1, $ARGV[0]) or die("Can't open $ARGV[0]\n");
```

```
open(FILE2, $ARGV[1]) or die("Can't open $ARGV[1]\n");
```

```
$file1 = <FILE1>;# Read a single line
chomp($line);
```

```
@file2 = <FILE2>;# Read the entire file
chomp(@file2);
```

OPENING A FILE FOR WRITE AND APPEND

Before a file can be written to, it must be opened in either write or append mode. This is done by prepending the filename in the `open()` function parameter list with a mode character. The mode character is the same ones used for redirection: `>`, and `>>`.

If a file is to be opened for both read and write, then use either `+<` or `+>` or `+>>`, depending on where the file pointer should be positioned when the file is opened.

Example:

```
open(OUTFILE, ">out.file");
open(ADDFILE, ">>add.file");
```

Writing to a File

Output can be directed to a particular open file by using the file handle with the `print()` function. The file handle is placed between the function name and the parameter list.

Output can be directed to *Standard Error* by using the `STDERR` file handle.

Another option is to use the `select()` function to select an open file as the default file for the `print()` function. This allows the `print()` function to be used as if output was being written directly to standard output.

Example:

```
open(OUTFILE, ">outfile")
print OUTFILE ("$line\n");
print STDERR ("Error detected\n");
select(OUTFILE);# Default is outfile
print("$line\n");# Output to outfile
```

Checking for End of File

The `eof()` function can be used to determine if the next read from an open file will return an "End of File." It can be used in conjunction with a file handle or with the <> operator.

Syntax:

```
eof(handle)
eof
```

The first form specifies a particular open file. If no file handle is given, then all files in the command list are used. That is, `eof()` will return a true value only after the last file in the sequence has been read. Either `eof()` or `eof(ARGV)` can be used.

The second form is used to test each file individually in the command line sequence when using the <> operator.

Example:

```
$file = $ARGV[0];
while ($input = <>) {
  print($input);
  if (eof) {
    print("End of $file\n");
    $file = $ARGV[0];
  }
}
```

CLOSING A FILE

When a process ends normally, the operating system closes all open file automatically. This is true for Perl programs as well. However, it is always good programming practice to close a file when it is no longer needed.

The `close()` function is used to close an open file. It takes a single parameter—a file handle of an open file.

Syntax:

```
close(handle)
```

The `close()` function is not needed if the same file handle is going to be used to immediately open another file. If an open command is issued using the file handle of a file that is already open, Perl will close the file first before using the file handle to open another file.

Example:

```
open(INPUT, "file1.data");
#Process file1.data
       . . .
open(INPUT, "file2.data"); # file1.data closed first
#Process file2.data
       . . .
close(INPUT);              # file2.data closed
```

THE `printf()` AND `sprint()` FUNCTIONS

The `print()` function can be used to print output to files, including standard output. This function is useful, but is limited in that it only provides unformatted output capability.

The `printf()` function, while more complex, provides a means of printing formatted output. The parameter list consists of a format specifier string

followed by an optional list of variables or literals which are to be printed in the positions indicated by the format specifier.

As with the `print()` function, the `printf()` function does not automatically append a newline character on the end of the output string.

Syntax:

```
printf(format_specifier, item, ...)
```

Output can be sent to files other than standard output by placing the file handle of an open file between the `printf()` function name and the parameter list.

Syntax:

```
printf(Optional_file_handle format_specifier,
item, ...)
}
```

The `format_specifier` parameter of the `printf()` function is a string consisting of (optional) text combined with one or more format specifiers which position the following list of variables and literals in the string.

TABLE I. *Format Specifiers*

%c	Single character
%d	Integer (decimal format)
%e	Floating point (scientific notation)
%f	Floating point (fixed-point notation)
%g	Floating point (compact of %e or %f)
%ld	Long integer (decimal format)
%lo	Long integer (octal format)
%lu	Long unsigned integer
%lx	Long integer (hexadecimal format)
%o	Integer (octal format)
%s	String
%u	Unsigned integer (decimal format)
%x	Integer (hexadecimal format - lower case)
%X	Integer (hexadecimal format - upper case)

Examples:

```
$i = 21;
$x = 456.78439;
$str = "Hello";
printf("%s %d %e\n", $str, $i, $x);
```

Output:

```
Hello
21  25  15
4.567844e+02  456.784390  456.784
```

The `sprintf()` function is not used for output per se, but is presented here due to it's similarity with the `printf()` function. This function is used to place formatted output into a string rather than writing it into a file. The function returns a formatted string which can be assigned to a scalar variable. The format specifiers are identical to those used with the `printf()` function.

This function is useful when converting integers to equivalent octal or hexadecimal strings.

Syntax:

```
$string = sprintf(format_specifier, item, ...)
```

Example:

```
$dec = 28;
$oct = sprintf("%o", $dec);
$hex = sprintf("%x", $dec);
print("$dec $oct $hex\n");
```

Output:

```
28 34 1c
```

CONTROLLING OUTPUT BUFFERS

Printed output is always buffered. *Buffering* implies that the output is kept inside an internal data structure for a while, until it is actually required to force the output of that data. For this reason, output may not immediately be displayed.

Perl does not provide a separate function to flush buffers. The `$|` variable can be used to force buffers to be flushed after each write operation. To invoke buffer flushing for a particular file, use the `select()` function with the file handle of the open output file. Then set the `$|` variable to 1. `$|` is normally set to 0, so setting it back to 0 will turn off the buffer flushing. It is important to invoke `select` to its previous value. By default the selected file handle is `STDOUT`. If you do not reselect `STDOUT`, a command such as `print("hello\n")`; will no longer print to `STDOUT` by default.

Example:

```
open(OUT, ">outfile");
select("OUTFILE");
$| = 1;
select(STDOUT);
print("hello\n");
```

THE `rename()` AND `unlink` FUNCTIONS

Two important functions to manipulate files are `rename` and `unlink`. The `rename()` function is used to move or rename files. The move operation is restricted to filesystem boundaries.

Syntax:

```
rename(oldname, newname)
```

Note that the function does not check to see if a file having newname already exists, but will simply overwrite it without warning. The function

will return a value of `true` upon success, and `false` if there is an error, such as having insufficient permission to rename the file.

The `unlink()` function is used to remove a file. It will accept a list of files and returns an integer representing the number of files actually removed. Technically, `unlink` removes a link to an internal data structure of Unix called an inode and decrements the link count.

If the link count is reduced to zero, the inode entry is "nulled" and the data blocks associated with the file are the free disk space data block pool. If the count has not reached zero, the inode entry and data blocks are retained for the other remaining link(s).

For further details consult a Unix manual.

The `unlink` function returns an integer with the number of files actually removed. If the number returned is zero, it is an indication that an error occurred and no file was removed. Such an error will occur, for instance, if the user attempts to remove a file while having insufficient permission to do so.

Syntax:

```
unlink(filelist)
```

In addition to these functions, Perl has many other functions available to create, remove, and list directories, create and follow symbolic links, and many more such functions. Although these are not typically needed to write simple CGI scripts, they will be needed when writing larger programs. Reading the Perl documentation is recommended for learning how to use these functions as the need arises.

USING FILE TEST OPERATORS

File test operators are used to test the status of files, whether they are open or not. The operator is followed by an expression which represents a file name. If the test is valid, a true value is returned. This makes the operators useful for testing file conditions with an `if` statement. Some of them return other values as well, such as the size of a file, and so on.

Example:

```
if (-r $file) {   # -r $file returns a true if the file
name in $file exists and is readable.
    open (IN, "$file") || die ("can\'t open $file\n");
}
```

Table J lists the various operators used to determine file types.

TABLE J. *File Test Operators*

b	block special file
-c	character special file
-d	directory
-l	symbolic link
-f	regular file
-p	named pipe
-t	terminal device file
-B	binary file
-S	socket
-T	text file

The operators used to determine access permissions are shown in Table K.

TABLE K. *Access Permissions Operators*

g	setGID bit set
-k	saved text bit set
-r	readable
-u	setUID bit set
-w	writable
-x	executable
-R	readable by real user only
-W	writable by real user only
-X	executable by real user only

Table L includes the operators for testing ownership.

TABLE L. *Ownership Testing Operators*

o	owned by effective user
-O	owned by real user

The operators for testing existence and contents are shown in Table M.

TABLE M. *Existence and Contents Operators*

-e	existence
-s	not empty
-z	empty

Table N includes the operators for access, modification, and change times.

TABLE N. *Access, Modification, and Change Times Operators*

-A	returns days since last access
-C	returns days since inode entry changed
-M	returns days since modified

LESSON SUMMARY

In this lesson, you have learned:

- Perl supports redirection from the command line.
- Files can be opened for read, write, or append operations.
- A file is opened for a read operation if no other operation is specified.
- File handles are used to reference open files.
- The <> operator is used to read from a list of files specified on the command line. The `close()` function is used to close a file.
- File test operators can be used to determine a variety of file conditions.
- Pipes can be opened to move data between a Perl process and another shell process.

REVIEW QUESTIONS

1. What does it mean to append to a file?

2. Show an example of opening a file for appending.

3. Show an example function call to convert an integer to a hex string, and save it in a variable.

4. What function can you use to change the name of a file?

5. What functions can you use to remove a file?

6. What operator can be used to determine if a file exists and has read permission?

7. Write a one line program to get the number of days since file `a_file` was last modified.

Answers on page 204

EXERCISE

1. Write a program which takes a list of files from the command line and prints them out double spaced.

2. Write a program which takes a directory name as a command line argument. For each file in the directory, list the access permissions for the user.

Example:

```
file1   read write execute
file2   read execute
file3   read write
```

3. Create a data file with 10 characters per record, but no record separator.

Example:

```
Record 001Record 002Record 003
```

- Write a program which will accept a record number as a command line parameter and print out the record.
- Write a program which will accept a record number as a command line parameter and a replacement record string. Replace the specified record with the command line string.

LESSON 9

Pattern Matching

OVERVIEW

Pattern matching is another very powerful facility in Perl. It allows the program to detect complex patterns in a string and trigger actions or modification based on these patterns. This lesson will cover the basics of the use of the pattern matching facilities in Perl.

LESSON TOPICS

- What is a Pattern in Perl?
- Operators and Characters
- Variable Substitution in Patterns
- Specifying Choices
- Reusing Previously Found Patterns
- The Translation Operator
- Extended Pattern Matching
- Pattern Matching Options

OBJECTIVES

By the end of this lesson, you should be able to:

- Use the match operator to identify a pattern in a string.
- Use extended regular expressions to specify patterns in strings.
- Understand the concept of pattern anchors.
- Reuse previously found patterns.
- Use the substitution operator to replace and delete text.
- Use the translation operator to modify text.

WHAT IS A PATTERN IN PERL?

A pattern is a character string that is used to search for all or part of another character string. Often, a pattern is nothing more than text, such as a word or group of words. At other times, it is important to limit the text to a specific location, such as the beginning or end of a line. Note that the use of patterns was introduced as the first parameter of the `split()` function. This first parameter can be specified in terms of an extended regular expression, not just limited to a character delimiter.

Perl uses extended regular expressions to specify patterns. Extended regular expressions take advantage of metacharacters to specify patterns in the text. The metacharacters used by Perl are:

```
\ | ( ) [ ] { } ^ $ * + ? .
```

These metacharacters must be escaped with a \ character if they are being used as a regular characters. Note that if the \ character itself is being used in a pattern, it must be preceded by a \ character as well. In addition, if the / character is used as a delimiter and as part of the pattern, it must be escaped with the \ character as well.

Programmers who are familiar with the `egrep` utility found with the Unix operating system, will already be familiar with the extended regular expression metacharacters that Perl understands. Perl provides an extensive regular

expression facility that is a superset of practically every other Unix utility that uses them.

Here are some examples of regular expressions:

```
/perl/
```

```
/How are you\?/
```

```
/\/usr\/bin\/perl/
```

OPERATORS AND CHARACTERS

The Match Operators

The match operator =~ is used to determine if a specified pattern occurs in a string. Note that it does not demand that the pattern match the entire string. The complement match operator !~ is used to determine if a specified pattern does not occur in a string.

The result of the comparison is either true (nonzero value) if there is a match, or false (zero value) if there is not a match. The opposite is true for the complement match operator. This makes it convenient to use match operators in the context of a conditional construct. These return values can be assigned to a separate variable if desired.

Note that Perl uses a "greedy" algorithm for matching. That is, when there is a choice, by default the match operator will take the largest pattern. A minimal algorithm can be invoked by appending a ? operator after any metacharacter which may invoke a greedy algorithm.

Examples:

```
$ret = $fruit =~ /apple/;
```

```
if ($cityStateZip =~ /AZ/) {
  $state = "Arizona";
}
```

Extended Regular Expressions in Perl

The metacharacters used in Perl are basically the same as those used with the `egrep` utility and the `awk` programming language. They are defined in Table O:

TABLE O. *Metacharacters Used in Perl*

`()`	used to create a pattern group (and store it in a numbered buffer)
`|`	used to divide alternate matches
`*`	matches zero or more of the previous character or pattern group
`+`	matches one or more of the previous character or pattern group
`?`	matches zero or one of the previous character or pattern group
`.`	matches any single character
`[]`	matches one of a set or range of characters (a leading ^ character is used to complement the set)
`{}`	specifies a specific number or range of matches of the previous character or pattern groups

Table P shows examples of metacharacters:

TABLE P. *Examples of Metacharacter Use*

`/ab*c/`	matches `ac`, `abc`, `abbc`, `abbbc`, ...
`/ab+c/`	matches `abc`, `abbc`, `abbbc`, ...
`/ab?c/`	matches `ac`, `abc only`
`/a.c/`	matches `a`, followed by any single character, followed by `c`
`/a[0-9]c/`	matches `a`, followed by a number, followed by `c`
`/a[^0-9]c/`	matches `a`, followed by any character except a number, followed by `c`
`/a{2,3}/`	matches `aa` or `aaa`
`/a{2,}/`	matches `aa`, `aaa`, `aaaa`, ...
`/a{2}/`	matches `aa`
`/a(b|c)/`	matches `a` followed by `b` or `c` (and stores `b` or `c` in buffer 1)

Character Range Escape Sequences

Some character sequences are used often enough to warrant their own escape designation. This is a single character preceded by a \ character which represents a range of characters usually designated with the [] operator. Escape characters are shown in Table Q.

TABLE Q. *Escape Characters*

\d	any digit [0-9]
\d	any character except a digit [^0-9]
\w	any word character [_0-9a-za-z]
\w	any character except a word character [^_0-9a-za-z]
\s	any white space character [\r\t\n\f]
\s	any character except a white space character [^ \r\t\n\f]

Pattern Anchors

Certain patterns are only valid if they occur in specific places in a file. The special characters and escape characters that are used for these conditions are called anchors. They are shown in Table R. Utilities which recognize regular expressions have used the ^ character to represent the beginning of the line and the $ character to represent the end of the line. Perl recognizes those and several other escape characters as well.

TABLE R. *Pattern Anchors*

^ or \A	beginning of line
$ or \Z	end of line
\b	word boundary
\B	not on word boundary

Examples:

```
/^In the beginning/
```

```
/\AOnce upon a time/
```

```
/They lived happily ever after\.$/
```

```
/the End\Z/
```

```
/\bthe\b/
```

```
/\Bthe\B/
```

VARIABLE SUBSTITUTION IN PATTERNS

Patterns can also be stored as strings in variables and then used within the pattern delimiters. This allows patterns to be manipulated as strings and then to be used in pattern matching operations.

Example:

```
$pattern = "\AOnce upon a time";
if ($line =~ /$pattern/) {
  $beginning = $line;
  }
```

SPECIFYING CHOICES

As mentioned before, the | metacharacter is used to separate pattern choices. It can be used within ( ) but is not required.

The pattern choices can include regular text, as well as any of the other metacharacters. This applies to the anchor metacharacters and escape characters as well.

Examples:

```
/abc|xyz/
```

```
/[Yy]($|es|ea|eah|ep)/
```

REUSING PREVIOUSLY FOUND PATTERNS

Patterns or portions of patterns enclosed in `()` are placed in temporary numbered buffers. The contents of the buffers are also stored in corresponding system variables.

The pattern in the first set of `()` is stored in buffer `1` and variable `$1`. The second in buffer `2` and variable `$2`, and so forth.

The buffer contents are referred to as `\1`, `\2`, `\3`, and so on. This syntax can be used in the overall pattern to designate a subpattern that has been repeated.

The contents of `$1`, `$2`, etc., are held until another match is performed. Thus, the contents of the pattern buffers persist in the variables beyond the point in the program where the match is attempted.

In addition to the system variables `$1`, `$2`, etc., the `$&` variable is used to hold the entire pattern which was matched.

Examples:

```
$string = "abcabcabc";
if ($string =~ /(abc)\1\1/) {
    print("$1\n");
}
print("$&\n");
```

Output:

```
abc
abcabcabc
```

Specifying a Different Pattern Delimiter

Patterns are usually delimited by `//` characters. However, the `m` operator allows the programmer to specify the character which directly follows the operator as the pattern delimiter. This is useful when the `/` character is part of the pattern.

Example:

```
/\/usr\/bin\/perl/
m!/usr/bin/perl!
```

The Substitution Operator

The substitution operator is used to replace all or part of a string with another string. It can also be used to delete portions of strings by using a null replacement string.

The syntax is similar to that used for the Unix Stream Editor (sed), except that it is used in conjunction with the match operator.

Syntax:

```
string =~ s/pattern/replace/options
```

Where:

- `string` = string to be searched (and modified)
- `pattern` = search pattern in the string
- `replace` = replacement pattern (may be null)
- `options` = options to modify the search pattern match

Most of the options for the substitution operator, shown in Table S, are similar to those used by the match operator:

TABLE S. *Substitution Operator Options*

`g`	change all occurrences of the pattern in the string
`i`	ignore case when evaluating replacements
`e`	evaluate replacement string as an expression before replacement
`m`	treat string as multiple lines
`o`	evaluate only once
`s`	treat string as a single line
`x`	ignore white space in pattern

Example:

```
$string = "A Man For Almost All Seasoning";
$string =~ s/ing/s/;
$string =~ s/Almost //;
print("$string\n");
```

Output:

```
A Man For All Seasons
```

THE TRANSLATION OPERATOR

The translation operator is similar to the Unix `tr` utility. That is, it performs a character-by-character replacement of the characters in one string for the characters in another string. Either `tr` or `y` (from `sed`) can be used.

The operator returns the number of characters actually translated. This return value can be assigned to another variable, if desired.

Unlike the substitution operator, replacement is on a single character basis and not on a pattern basis.

As with the `tr` utility, options are provided which allow characters to be deleted or "squeezed" down from a sequence of multiple occurrences to a single occurrence.

The match operator is used with the translation operator to designate the string in which the replacement is to occur.

Syntax:

```
string =~ tr/targ_list/repl_list/options
```

```
string =~ y/targ_list/repl_list/options
```

Where:

- `string` = string to be translated
- `targ_list` = list of characters to be replaced
- `repl_list` = list of replacement characters

The `tr` options are shown in Table T:

TABLE T. *`tr` Options*

c	complement (use all characters not specified)
d	delete characters (null replacement string)
s	replace sequence of same character with single characte

The target list and the replacement list are expected to be the same size. If the target list is shorter than the replacement list, the last character in the target list is used to pad the list to the length of the replacement list. This does not matter if the `d` option is used.

If the replacement list is null and the `d` option is not being used, the target list is replicated and used as the replacement list. This is useful when using the return code to obtain the length of a string.

EXTENDED PATTERN MATCHING

Perl also has some extended pattern matching capabilities which allow programmers to do the following:

- Use parentheses to group a subpattern without having them saved in a buffer.
- Embed match operator options in patterns.
- Define additional boundary conditions which must exist before a pattern is matched.
- Add comments to patterns.

PATTERN MATCHING OPTIONS

There are several options which can be used when specifying a pattern which modify the way the pattern is matched. Two of the more common options used are the `g` and the `i` option.

Syntax:

```
/pattern/options
```

The `g` match option is quite typically used in the context of a substitute command. It indicates to match and replace multiple instances of the pattern in the string, not just the first match.

Example:

```
$string1 = $string2 = "Mississippi";
$string1 =~ s/i/o/;
$string2 =~ s/i/o/g;
print "$string1\n";
print "$string2\n";
```

Output:

```
Mossissippi
Mossossoppo
```

The `i` option indicates to ignore upper or lower case distinctions.

Example:

```
$string = "Memorabilia";

$match = $string =~ /MEMO/i;
print("Did it match: $match\n");
```

Output:

```
Did it match: 1
```

The `e` pattern option is a very powerful and useful option. Instead of the substitution pattern providing a string, it provides a piece of code to `eval` or `execute`. That piece of code is often times a function, which can take as parameters one of the items found in the pattern, for example, `$1`, `$2`, etc.

The substituted string is whatever that function returns.

```
$str =~ s/=\s*(\w+)\s*;/name_alternative($1)/e;
```

The above call will call the function `name_alternative` with the alphanumeric found on the right of an `=` inside the string `$str`. If such a name is found the function `name_alternative` is expected to return an alternative name for this alphanumeric string and return it. The result will be substituted in.

LESSON SUMMARY

In this lesson, you have learned:

- Perl uses extended regular expressions to assist in defining patterns.
- The match operators, =~ and !~ are used to test for a pattern contained in a string.
- Escape characters are used to represent commonly used character sets and pattern anchors.
- The substitution operator is used to replace and delete subpatterns in a string.
- The translation operator is used to convert between sets of characters.

REVIEW QUESTIONS

1. Explain in words what this pattern finds:

```
$a =~ /(abc|cda)/;
```

2. What possible values can `$1` have in the above example after the statement is executed?

3. Assume the following statement:

```
$a = "aaeeoouuuuuuaaaaaaa";  $a =~ /(a+)/;
```

   What will be the value of `$1`?

4. Assume the following statement:

```
$a = "aaeeoouuuuuuaaaaaaa";  $a =~ /(a+)$/;
```

   What will be the value of `$1`?

5. What does the combination of `e` and `g` options allow you to do in a substitute invocation?

Answers on page 204

EXERCISE

1. Write a program that mimics a basic Unix `egrep` utility, that is, one which accepts a pattern (string representing an extended regular expression) and a list of files from the command line and prints out all lines which contain the pattern.
2. Write a program which accepts a word and a file name from the command line and lists the line number and position in each line in which the word occurs.
3. Write a program which accepts a list of file names from the command line and strips out all characters except numbers. Count the number of times each individual digit occurs.
4. Write a program which mimics the `javadoc` utility by printing only the text in a file which falls between the `/**` and `*/` comment delimiters.

LESSON 10

Creating and Using Subroutines

OVERVIEW

Computer programs facilitate complex and repetitive tasks. When a good programmer finds code repeating over and over again, he or she right away thinks of using a subroutine to make life easier and write a better program. This lesson shows how to write and use subroutines precisely for this in mind.

LESSON TOPICS

- What are Subroutines?
- Calling Subroutines
- Returning Values from Subroutines
- Defining Local Variables in Subroutines
- Passing Values to Subroutines
- Recursive Subroutines

OBJECTIVES

By the end of this lesson, you should be able to:

- Define a subroutine.
- Define and use local variables within a subroutine.
- Pass values to a subroutine.
- Return values from a subroutine.

WHAT ARE SUBROUTINES?

Up to this point, the course has used various functions provided by the Perl language. It is also possible to create user-defined functions, usually called subroutines. The difference is largely semantic, although in general, subroutine names do require a leading & character when they are called.

Other than that, subroutines are called the same way as the built-in functions. They can be designed to accept a list of parameters, and they always return a value.

Subroutines can be defined in the same file or in different files, or created "on the fly" using the `eval` function. The latter two techniques will be discussed in later sections.

Syntax:

```
sub sub_name {statements}
```

Example:

```
sub greeting {
  print("Hello!\n");
}
```

CALLING SUBROUTINES

Subroutines are called the same way that built-in functions are called. If a parameter list is required, the list is enclosed in parentheses, unless the subroutine has been imported or predefined. The return value can be assigned to another variable or used directly.

Syntax:

```
sub_name;
```

```
sub_name(params);
```

```
sub_name params;
```

A leading & character is required with the subroutine name if the subroutine definition occurs after the point in the program where it is first called. If the subroutine is defined before it is called, then the leading & character may be omitted.

Instead of defining subroutines at the beginning of a program, a forward reference can be used to indicate that the subroutine will be defined in another part of the program. If a forward reference is used, the leading & character may be omitted from the subroutine name.

Syntax:

```
sub subname;
```

Example:

```
sub greeting;# Forward reference. . .
&greeting; # Always correct to use leading &
greeting;  # Can be called without leading &
```

RETURNING VALUES FROM SUBROUTINES

A subroutine always returns a value. The term *value* is used in a generic sense, because a subroutine may return either a scalar or a list.

There are three different ways in which values are returned from a subroutine. If no other mechanism is provided, the output of the last statement executed in the subroutine is returned. This may seem like the most convenient mechanism, but it can be deceiving. For example, if the last set of statements in a subroutine are within a loop construct, the last statement executed will be the loop test expression, not the last statement in the loop statement block.

Example:

```
sub counter {
  while ($i++ < 10) { # 0 (false) returned
    $total += $i;# Not this
  }
}
```

The preferred way to return a value is to use a return statement. With this method, values are returned explicitly, and there can be no doubt as to what is being done or what is being passed. Note that when the return statement is executed, the subroutine terminates.

Example:

```
sub counter {
  while ($i++ < 10) {
    $total += $i;
  }
  return $total;# Value of 45 returned
}
```

DEFINING LOCAL VARIABLES IN SUBROUTINES

Variables which are defined in the main part of a Perl program can be used by subroutines as global variables. This is not recommended, as it is difficult to track bugs if a number of subroutines are assigning to the same variable.

Instead, subroutines should use their own local variables and receive external values through the parameter list. To define variables as local to the subroutine, either the `my` or `local` declarations can be used. Otherwise, variables defined in a subroutine will become global variables and may create a conflict.

The `my` declaration defines variables which only exist inside the subroutine within which they are declared.

The `local` declaration defines variables which remain in effect within the subroutine within which it is declared and any other subroutines which are called by this subroutine.

In either case, variables can be initialized when they are defined.

Examples:

```
my $init;
```

```
my $name = "Jennifer";
```

```
my ($x, $y) = (2, 4);
```

```
my ($name, @grades) = ("Smith", 87, 92, 95);
```

The `local` declaration can be used in the above examples if the variable is to be carried into called subroutines.

PASSING VALUES TO SUBROUTINES

Values are passed into subroutines using a parameter list. Normally, the parameter list is enclosed in parentheses following the subroutine name given to invoke the subroutine. Leaving the parenthesis out at a call has certain obscure effects that can confuse the beginning programmer. For the time being, always include the parenthesis at a call even if no parameters are being passed.

Inside the subroutine definitions, parameters are contained in the @_ list variable. They may be accessed as a list or individually using the [] operator.

Quite often, the individual parameter elements are reassigned to local variables using a list assignment construct. If only a portion of the parameter list is to be assigned to a list variable, make sure that the list variable appears as the last item in the assignment list. Otherwise, the entire parameter list will be assigned to the list variable and only null values will be assigned to any following scalar or list variables.

Example:

```
my @params = @_;
my ($x, $y, $z) = @_;
my ($a, @list) = @_;
my (@items, $i) = @_;# WRONG! $i = null
my (@a, @b) = @_;    # WRONG! @b = (null)

print($_[0]);        # prints the first parameter in @_
```

RECURSIVE SUBROUTINES

Perl supports recursive subroutines. That is, a subroutine may call itself. The programmer must take great care in writing recursive subroutines to make certain that a termination condition is included that guarantees that the routine will not call itself forever. The subject of recursive subroutine calls is fairly complex. This technique will rarely be required for writing CGI scripts. It is mentioned, so that the student is aware of the existence of this

technique. Certain kinds of problems can be solved more easily using this methodology, such as the function in the following example:

```
sub factorial {
  my ($num) = @_;
  if ($num < 0) {
    return 0;
  }
  if ($num == 1 || $num == 0) {
    return 1;
  }
  else {
    return $num * factorial($num - 1);
  }
}
```

LESSON SUMMARY

In this lesson, you have learned:

- Subroutines are functions that are defined by the programmer.
- Subroutines may be designed to accept a parameter list.
- A parameter list is carried into a subroutine through the `@_` list variable.
- The return statement should be used to explicitly return scalar or list values to the calling program.
- Forward referencing can be used to declare a subroutine at the beginning of a program so that it can be defined later.
- The `my` and `local` declarations are used to define variables that are local to a subroutine.

REVIEW QUESTIONS

1. When you enter a subroutine, where are the parameters passed stored?

2. What is the difference between *global* variables, *local* variables and *my* variables?

3. What happens if a function returns a list and gets called as follows:

```
$l = func(); ?
```

4. Can you call a function without it having been yet defined and with no forward declaration?

5. Assume that the parameters of a function are assigned this way:

```
my (@l, $a) = @_;
```

   Is this an acceptable way to collect parameters?

Answers on page 205

EXERCISE

1. Write a program with a subroutine that accepts a list of values and returns the largest, smallest, and average values to the calling program.
2. Write a program that uses a recursive subroutine to print out a string in reverse order.

LESSON 11

References

OVERVIEW

References, also known as pointers in other languages, extend the facilities provided by scalar, lists and hashes to allow the construction of arbitrarily large and complex data structures with greater ease. This lesson shows how to use references for this purpose.

LESSON TOPICS

- What is a Reference?
- Dereferencing
- References to References
- Creating Multidimensional Lists
- Creating Multidimensional Hashes

OBJECTIVES

By the end of this lesson, you should be able to:

- Distinguish between hard and symbolic references.
- Create references for data types supported by the Perl programming language.
- Dereference hard references.
- Use references to create multidimensional arrays.

WHAT IS A REFERENCE?

The subject of references in Perl is not a "must learn" item for CGI programming, nor is it considered a basic topic in Perl programming. It is however, a very useful tool in writing complex programs. If you are totally new to programming, and this section seems quite difficult, you may wish to skip it, and return to it after you have gained some more experience.

A reference is a scalar value which acts as a pointer to another object. There are two types of references in Perl:

- *Symbolic references* point at another item via Perl's symbol table.
- *Hard references* have a more direct link to the object referenced and are more efficient.

This course will only cover hard references.

TIP

To have the Perl interpreter guarantee that you do not use symbolic reference by accident, issue the statement:
`use strict refs;`

References have many uses. Either symbolic or hard references can give the programmer a means to pass a large amount of data to a function without moving a lot of data around. That is to say, instead of copying some large data structure onto a temporary that the function can look at, we copy a small pointer to such data, thus accomplishing the call more efficiently.

TIP

When you become more proficient at Perl you will want to revisit the subject of symbolic references, since there are many valuable uses for these as well.

A hard reference, or reference of short, is a type of scalar that acts as a pointer to any data structure. The data structure pointed to is not required to be a scalar. References allows the creation of arbitrarily complex data structures, such as for instance, a list of scalars where any one of the scalar elements can itself be a reference to another list or a hash. Copying a reference

does not make a copy of whatever item is pointed to. It only copies the reference, which is typically considerably smaller.

The backslash \ operator is used to create a hard reference to any data item. The operator is placed in front of the data item to be referred to. It is then typically assigned to a scalar variable. Hard references can be created for any data item, even literals.

Examples:

```
$scal_ref = \$scalar;
```

```
$cons_ref = \3.141592654;
```

```
$list_ref = \@list;
```

```
$hash_ref = \%hash;
```

```
$subr_ref = \&subroutine;
```

```
$glob_ref = \*HANDLE;
```

DEREFERENCING

The process of using a reference is called *dereferencing*. There are several ways to dereference a reference. The easiest way is to just use the reference as a variable name.

Example:

```
$language = "Perl";
$langref = \$language;
$string = $$langref;          # $string = "Perl"
```

This works the same for all data types

```
$$listref[1] = 2;
```

```
push(@$listref, 5);
```

```
$$hashref{"Bill"} = "Hillary";
```

```
$&subref($a, $b, $c);
```

```
print $handref ("Hello!\n");
```

REFERENCES TO REFERENCES

References can be made more than one level deep, if desired. A reference to a reference can be created and dereferenced using the same techniques described earlier. This can be extended to create references to references which reference references, and so on.

Example:

```
$language = "Perl";
$langref = \$language;
$reflangref = \$langref;
print($language, "\n");
print($$langref,  "\n");
print(${$langref}, "\n");
print($$$reflangref,     "\n");
print($${$reflangref},   "\n");
print(${${$reflangref}}, "\n");
```

All of the `print()` statements above print the string `"Perl"`.

CREATING MULTIDIMENSIONAL LISTS

It was noted earlier that a list could consist of a set of references. Even though the previous example used references to scalars, references to any data type could be used. In fact, any combination of references to any data type would be legal.

To dereference a list reference element, use the same methods as before.

Example:

```
@reflist = (\$a, \$b, \$c);
print("${$reflist[1]}\n");
```

Suppose each element is a reference to a list. Such a structure can be created using anonymous lists. An anonymous list is defined by a list encased in brackets, `[]`'s. It builds a list and returns a reference to that list. There is no symbol table entry for directly accessing the list, without dereferencing the pointer.

Once created, individual elements can be dereferenced as shown in this example:

```
@twoxtwo = ([1,2], [3,4]);
With this structure:
$twoxtwo[0]->[0] == 1
```

This can be shortened to:

```
$twoxtwo[0][0] == 1
```

This makes it look more like the multidimensional array notation of the C programming language.

Note the subtle difference between the previous example and this one:

```
$ref2x2 = [[1, 2], [3, 4]];
```

This is a reference to a list of lists. It must be dereferenced as follows:

```
$ref2x2->[0]->[0] == 1
```

It can also be written as:

```
$ref2x2->[0][0] == 1
```

CREATING MULTIDIMENSIONAL HASHES

Multidimensional structures can also be applied to hashes.

Consider the following structure:

```
%couples = ( Clinton => ["Bill", "Hillary"],
             Bush    => ["George", "Barbara"],
             Reagan  => ["Ronald", "Nancy"  ] );
```

This is a hash in which the values are references to anonymous lists. Dereferencing an individual element is similar to the method used previously:

```
$couples->{"Bush"}[1] == "Barbara"
```

As before, the infix operator may be omitted:

```
$couples{"Bush"}[1] == "Barbara"
```

Now consider this structure:

```
$refcouples = { Clinton => ["Bill",   "Hillary"],
                Bush    => ["George", "Barbara"],
                Reagan  => ["Ronald", "Nancy"  ] };
```

This is a reference to an anonymous hash whose values are references to anonymous lists. It can be dereferenced as follows:

```
$refcouples->{"Bush"}[1] == "Barbara"
```

Since `$refcouples` is a scalar holding a reference, the `infix` operator cannot be omitted when dereferencing.

LESSON SUMMARY

In this lesson, you have learned:

- A reference is a scalar value which holds a variable name (symbolic) or address (hard).
- The backslash \ operator is used to create a hard reference to a data item.
- A reference to a file handle can be created with a reference to a type-glob.
- Anonymous references can be created for lists, hashes, and subroutines.
- The `ref()` function can be used to return a string indicating the data type of the item being referenced.
- References can be dereferenced by using the name directly or within `{}` characters as a statement block.
- Elements of list and hash references can be dereferenced using the infix `->` operator.
- Multidimensional lists and hashes can be created with lists and hashes of references.

REVIEW QUESTIONS

1. Given the following statement, write two different ways to access the third element of the array and assign it `$b`.

```
$a = ["dog", "cat", "rabbit", "lion"]
```

2. Write an example of a statement involving anonymous hash containing anonymous lists.

3. One can write a function that traverses (accesses every element) depth first an arbitrary list of lists starting at `@l` and adds up all the numbers found in the leaf nodes (nodes not containing list references).

   If you have some programming experience, or are extremely bold, try to write such a function.

   If you are human and beginning programming, take a look at the answer and see if you can understand all it is doing. Make sure you identify all the places in the program where references are being used. (hint it uses recursion).

4. Show some code to get a reference to an existing array `@l` and store it in a scalar variable `$r`.

Answers on page 205

EXERCISE

1. Demonstrate the pass-by-reference capability of Perl by swapping two scalars defined in the main part of the program within the context of a subroutine.

2. Define two 2 x 2 arrays using anonymous lists. Pass the arrays to a subroutine, and add them together. Return a reference to sum array, and print the values from the main part of the program.

3. Write a program to read a data file with records containing the following fields:

   - Account number
   - Name
   - Social Security Number
   - Balance

   Use the account number as a key for a hash that has a reference to a list of the other items. Create subroutines which will return a specified item in the list, given an account number.

LESSON 12

Packaging and Modules

OVERVIEW

Many good programmers hate doing the same tasks over and over. The key approach to avoiding this is code reusability. That is what packages and modules are all about. The approach to teaching CGI programming in this course is based on an excellent module, `CGI.pm`*. This lesson covers the basics of how to use and create packages and modules.*

LESSON TOPICS

- What is a Package?
- The main Package
- Defining and Switching between Packages
- Defining Subroutines in Packages
- Symbol Tables
- Referencing Package Contents
- What is a Module?
- Using Object-Oriented Modules
- The Comprehensive Perl Archive Network (CPAN)

OBJECTIVES

By the end of this lesson, you should be able to:

- Describe the concept of a package.
- Switch between packages.
- Create packages with variables and subroutines.
- Use object-oriented modules.
- Access the Comprehensive Perl Archive Network.

WHAT IS A PACKAGE?

Variables and subroutines that are defined within a Perl program are stored in a *namespace*, sometimes called a symbol table. In Perl, a namespace is called a *package*.

Perl allows the programmer to define more than one package for a program. Each package has its own set of variables and subroutines. Variables are treated separately between packages, even if they have the same name. Therefore, programmers can carry two or more sets of values in a program by assigning to the same variable names in different packages.

THE main PACKAGE

By default, the package in which variables and subroutines are stored is called `main`. The following identifiers are forced into the main package even though they may be used in other packages:

- `ARGV`
- `ENV`
- `INC`
- `SIG`
- `STDIN`
- `STDOUT`
- `STDERR`

Only identifiers beginning with a letter or an underscore can be kept in packages other than `main`. All others, including system variables like `$_` and `@_`, are kept in `main`.

DEFINING AND SWITCHING BETWEEN PACKAGES

The package statement is used to define a new package within a Perl program.

Syntax:

```
package packagename
```

Once a new package name has been declared, subsequent variable and subroutine definitions are defined within the context of that package and become part of the namespace.

The same package statement and syntax is used to switch to a different package. Switching packages brings that namespace into focus. Since pack-

ages are independent, there is no problem with using the same variable name in more than one package.

Example:

```
$x = 5;              # $x defined in package main
print("$x\n");
package mypack;      # Define package mypack
$x = 7;              # $x defined in package mypack
print("$x\n");

package main;        # Restore package main
print("$x\n");
```

Output:

```
5
7
5
```

DEFINING SUBROUTINES IN PACKAGES

Subroutines are placed in the package that is current at the time they are defined. Thus, several versions of the same subroutine can be placed in different packages without conflict.

Example:

```
package mypack;      # mypack is current package
sub packid {         # packid defined in mypack
  print("mypack\n");
}
package urpack;      # urpack is current package
sub packid {         # packid defined in urpack
  print("urpack\n");
}
```

SYMBOL TABLES

The symbol table for each package is kept in a hash whose name begins with the package name followed by two colons. If desired, all of the entries in the table can be obtained for a particular package.

Example:

`%main::` Symbol table for the `main` package

`%::` Symbol table for the `main` package (default)

`%mypack::` Symbol table for the `mypack` package

REFERENCING PACKAGE CONTENTS

Variables in one package can be referenced from another package by appending the variable identifier onto the symbol table name. The leading character of the variable becomes the leading character of the symbol table name.

Example:

`$mypack::var` scalar variable `$var` in package `mypack`

`@urpack::list` list variable `@list` in package `urpack`

`%main::hash`hash variable `%hash` in package `main`

`&urpack::mysub` subroutine `&mysub` in package `urpack`

Perl also allows programmers to specify that there is no current package by using the package statement without a package name. When this is done, variables must carry an explicit package name when they are used.

Example:

```
$x = 5;    # Current package is main
package mypack;# Current package is mypack
$y = 9;
package;   # No current package
print("$main::x\n");
print("$mypack::y\n");
```

Output:

```
5
9
```

WHAT IS A MODULE?

A module is a package that is defined in a file whose name is the same as the package name with a `.pm` extension. For example, a module containing the `mypack` package would be stored in a file called `mypack.pm`.

This allows packages to be defined and reused in the same sense that function libraries are defined and reused in other languages.

USING OBJECT-ORIENTED MODULES

Some modules have *object-oriennted* (OO) interfaces. These modules are typically designed with the goal of hiding the implementation details. The external interface is typically much simpler than the internals. The use of object-oriented modules is only slightly different than the use of non-object-oriented modules.

To use a Perl object-oriented module it must first be included into the program, as would any other module. This is done by means of the `use` command, as in the following example, which includes the `CGI.pm` module.

```
use CGI;
```

Unlike what is typically done with the `require` command, the name of the module should be given without quoting it and without including the `.pm` extension.

It is expected that the module can be found in the `@INC` include path.

After including the module, the programmer is typically required to create an instance of the object involved. To do this, one typically invokes a so-called *constructor* method. Methods are simply functions that act upon an object.

The constructor method can have any name in Perl, but a common convention is to use the name `new`. The following example shows the two common ways to invoke a constructor:

```
my $var = new Some_mod 1, 2, 3, 4;
```

or

```
my $var = Some_mod->new(1, 2, 3, 4);
```

The first method is the most commonly used. Despite the syntactic difference, the two ways are identical. They both return a reference to a new `Some_mod` object created and initialized with parameters `1`, `2`, `3`, 4.

When using the CGI module, an object might be instantiated this way:

```
my $cgi = new CGI {name => 'bill', job =>'postmaster'};
```

or alternatively:

```
my $cgi = CGI->new( {name => 'bill', job =>
postmaster'} );
```

The `{name => 'bill', job => 'postmaster'}`, is an anonymous hash reference, which defines an internal state for the object referenced by `$cgi`. The keys `name` and `job` are hashed parameter entries that can initialize the state of this object.

Once you have initialized the object, you can invoke its methods by derefreancing by means of the `->` opreator, as shown in this example:

```
$cgi->param("name");
```

This method is simply a function call to a function named `param`, which acts on the object whose reference is stored in `$cgi`. `"name"` is a

parameter passed to this function. In the example below, a function header acts on an object `$cgi`, which gets passed, in this case, no parameters:

```
$cgi->header;
```

Any global variables associated with the object can be accessed just like they would for any other module that is in the form of:

```
$package::var_name
```

For example, `$CGI::POST_MAX`, as in:

```
$CGI::POST_MAX = 3000;
```

or

```
$maximum_post_size = $CGI::POST_MAX;
```

Each module should have documentation that describes the various methods available for using that module. Modules typically need to get installed into a Perl distribution by a system administrator. When modules get installed, typically their documentation gets installed as well. Some modules come pre-installed in the Perl distribution, such as the popular `CGI` and `Safe` modules. Once the documentation to a module has been properly installed, the documentation can be read by means of the `perldoc` program that comes with the Perl distribution.

With the documentation that ships with a module and this explanation, you should be able to use any object-oriented module that you think can help you create better and more successful programs with less effort. In the CGI programming lessons, the CGI module written by Dr. Lincoln Stein is used. This module, as previously mentioned, comes pre-installed in recent Perl distributions. It makes it much easier to write powerful CGI scripts, with a lot more fun.

To view the documentation on the `CGI.pm,` execute the command:

```
perldoc CGI
```

THE COMPREHENSIVE PERL ARCHIVE NETWORK (CPAN)

A large number of modules are available for downloading from the Comprehensive Perl Archive Network (CPAN). A good place to start is the home page at:

```
http://www.perl.com/perl/CPAN/README.html
```

Note that there are a number of mirror sites. Both documentation and source code are available. In addition, a search facility is available to help find specific modules. The archive is arranged in a series of directories.

Once the correct directory has been found, files can be downloaded through a Web browser by selecting the appropriate file link.

The CPAN module is another useful resource that can be used to simplify downloading and installing new modules from a CPAN site.

LESSON SUMMARY

In this lesson, you have learned:

- A package is a namespace, or symbol table.
- A Perl program can reference more than one package.
- The default package in which variables and subroutines are stored is called `main`.
- The `package` statement is used to define a new package within a Perl program.
- Subroutines and variables are placed in the package that is current at the time they are defined.
- Variables and subroutines in one package can be referenced from another package by using the `::` operator.
- A module is a package that is defined in a file whose name is the same as the package name with a `.pm` extension.
- Modules can be imported into a Perl program with either the `use` statement or the `require` statement.
- Perl provides a number of predefined modules with different capabilities.
- Even more modules are available for downloading from the Comprehensive Perl Archive Network (CPAN).

REVIEW QUESTIONS

1. Suppose you have the following piece of code:

```
{ package angel;  $a = 1; $b = 2; }
            package devil;
            $a = 'dog';
            $b = 'cat';
```

   What would be a statement within package `devil` that would assign to `$c` the sum of variables `$a` and `$b` from package angel?

2. What are packages for?

3. Show an example of the two different types of syntax to create an object with a new constructors for a particular module, say `CGI.pm`.

4. Show a call to a method `param` with a parameter value `foo` for an object called `$cgi`.

5. Give the statement you would use to include package `CGI.pm`.

6. Show a one line statement to dump the name of all the symbols in package `main`.

Answers on page 206

EXERCISE

1. Write a program that has two versions of a subroutine in different packages. One version prints a string in lower case and the other version prints a string in upper case. Print the command line parameter list using both versions.

2. Write a program which uses the built-in integer package to perform arithmetic operations using integer arithmetic. Demonstrate the difference between floating point and integer division, multiplication, addition, subtraction, and modulo operations.

3. Create a module with a package called `string`. Define subroutines in the package which reverse the order of a string, rearrange the characters of a string in sorted order, move the first character to the end of the string, and move the last character to the beginning of the string.

LESSON 13

Using the `CGI.pm` Module

OVERVIEW

There are many useful Perl modules to use when writing CGI scripts. These modules can aid in a number of tasks, including decoding CGI input from the client, sending mail, and maintaining databases and logs. This lesson discusses the `CGI.pm` *Perl module, which is used throughout this course.*

LESSON TOPICS

- The CGI.pm Module
- Creating a CGI.pm Object
- Getting an Input Parameter

OBJECTIVES

By the end of this lesson, you should be able to:

- Create a CGI object with or without default parameters.
- Retrieve parameters using the `param` method of `CGI.pm`.
- Print out an HTTP document headers using the `CGI.pm` header method.

THE `CGI.pm` MODULE

`CGI.pm` is a very useful module which allows you to respond to client requests without having to toil with the low-level functions of the CGI architecture. This lesson explains the most basic features of the `CGI.pm` module. To understand the module in greater detail, read the document that comes with it.

CREATING A `CGI.pm` OBJECT

The example below simply creates a CGI object which can be used throughout the script as needed:

```
$obj = new CGI;
```

The second example is the same as the first, but it also sets up some default input so the program can be tested without a Web server. This is discussed in greater detail in the lesson on debugging:

```
$obj = newCGI {param1 => 'value1', param2 =>
'value2 }
```

GETTING AN INPUT PARAMETER

The code below will set `$value` to the value associated with the CGI parameter `param_name`.

```
$value = $obj->param('param_name');
```

In the event that the CGI parameter `param_name` has more than one value, the param method will return a list of all the values associated with the parameter. Therefore, you should instead assign the return value to a list.

Multiple values for a parameter occur when you have a form that has multiple inputs with the same name, or list boxes that allow multiple selections.

Example:

```
@values = $obj->param('param_name');
```

To print out the HTTP header:

```
print $obj->header;
```

This statement will print the header to standard out (for example, to the client while passing through the server).

To create a document that is not an HTML document, explicitly pass in the document MIME type desired. For example, to print out the header for a plain-text document use this expression:

```
print $obj->header("text/plain");
```

The three methods discussed above should provide most of what is needed from this Perl module, allowing a programmer to quickly and easily deal with input from a client.

An important variable to know about is the `$CGI::POST_MAX` variable, a global variable which sets the amount of data that `CGI.pm` will read in from a client. The importance of the `$CGI::POST_MAX` variable will be further discussed in the later lesson on CGI security.

LESSON SUMMARY

In this lesson, you have learned:

- How to Create a CGI object with or without default parameters.
- How to retrieve parameters using the `param` method of `CGI.pm`.
- How to print out the HTTP document headers using the `CGI.pm` header method.

REVIEW QUESTIONS

1. What is the default document type for the header method of `CGI.pm`?

2. Give a example of creating a `CGI.pm` object with two default parameters.

3. How would you retrieve a parameter named `cat` using `CGI.pm` object you created for question 2?

Answers on page 207

EXERCISE

1. Rewrite the example script in the previous lesson to use the parameters `bread` and `pet` instead of `name` and `job`.

LESSON 14

Retrieving Data with CGI

OVERVIEW

The last lesson covered the basics of how a CGI request works. This lesson will describe in greater detail how data is passed between the HTTP client and the CGI script, as well as what other types of information get passed to the CGI script.

LESSON TOPICS

- GET and POST Methods
- URL Encoding
- Environment Information Available to the CGI Script
- HTML Forms

OBJECTIVES

By the end of this lesson, you should be able to:

- Both describe and contrast the different ways data can be passed to your script.
- Describe some of the other types of data the server passes to your script.
- Describe and understand the methods used to encode requests sent to your CGI script.
- Describe how to use HTML forms and other methods of making requests to a CGI script.

GET AND POST METHODS

The most common methods of sending data to the server are the `GET` and `POST HTTP` request methods. Other methods such as `OPTIONS`, `HEAD`, `PUT`, `DELETE`, and `TRACE` also exist in the HTTP/1.1 protocol, but these will not be discussed in this course.

The `GET` method is the most commonly used method for requesting a document, and is also the method most Web clients (browsers) use to request regular static documents such as HTML and graphic files. With the `GET` method, all data is passed on through the request URL (Unified Resource Locator. Because of the way this request mechanism works, scripts using `GET` can be embedded into the links of HTML documents, as well as used to respond to requests generated by HTML forms.

The ability to embed data into requests is indeed useful, except that, for most servers, the utility of the `GET` method is limited by the amount of data that can be passed via this method. Most servers limit the parameter part of the request to 256 characters, so that it is often too small for returning data involved with larger forms. The `GET` method passes data to the CGI script via the environment variable `QUERY_STRING`. This variable either contains the part of the request URL following the `?` as it was passed from the web client, or it contains the contents received from an HTML form using URL encoding.

POST is the method primarily used to process forms. With the POST method, the data is sent in the request body instead of as a part of the request URL (as it is in the GET method). The advantage of using the POST method is that the amount of data which can be passed is virtually unlimited. The data is made available to the CGI script via standard in (STDIN file handle). The number of bytes to be read is given in the environment variable: CONTENT_LENGTH. When you use this request method. As with the GET method, this input will also be in URL-encoded form.

The good news is that any CGI script you write can automatically handle both request methods, so you need to only consider which method best suits your needs. Usually the POST method is used for processing forms and, the GET method is implicitly used for requests embedded in anchor tags (<A HREF="specific_url">).

URL ENCODING

Using the GET method to embed requests for resources implemented as CGI scripts that involve parameters requires an understanding of URL encoding. The same encoding is used with the POST method when getting data from a form. The encoding scheme for parameters is fairly simple, as illustrated by the following example:

```
http://web.server.com/cgibin/
hello.cgi?name=jon&job=web
```

The first part of this URL is the resource identifier http://web.server.com/hello.cgi. This is followed first by the character ?, which separates the resource identifier from its parameters. This is then followed by the parameters which are in the form of a list of name=value pairs, each separated by the character &. Thus, this URL makes a request to the CGI script hello.cgi and passes it the two parameters name and job with the values jon and web assigned to each, respectively. The only catch is that there are some characters not permitted to be in either the name or the value portion of the parameter list when they are trying to represent themselves and not a special meaning. These characters are:

```
*, !, ?, #, &, ., +, =, %, (space)
```

The rationalization for URL encoding is explained in RFC (Request for Comments) 1630, Universal Resource Identifiers in the WWW. (This is available at *http://www.w3.org*; search for "RFC 1630.") If any of these characters need to be put into the CGI parameters, *escape* them. To do this, turn the characters into the form `%xx`, where `xx` is the hexadecimal ASCII value of the character.

For example, to pass the parameter `flavor` with the value `chocolate&vanilla` to the script `icecream.cgi` say:

```
http://web.server.org/31f/
icecream.cgi?flavor=chocolate%26vanilla
```

A small program which escapes any of these characters can make things simpler:

```
url_escape.pl
#!/usr/bin/perl
$cmd_line_input = $ARGV[0]; # Get the command line
input.
$cmd_line_input =~ s/([^\w\d])/sprintf("%%%2.2X",
ord($1))/ge;
print($cmd_line_input, "\n");
```

This program escapes some characters which do not need to be escaped, for example a non-letter or numeric characters. This is acceptable since the receiving end ultimately decodes all escaped characters correctly. In fact, we could simply escape every single character, but that would make our strings up to three times longer and quite unreadable. There is also a shortcut you may use when escaping space characters. To escape a space you can simply put a `+` where you want a space.

For example "Use the force Linus" could be safely escaped as either of the following:

```
Use+the+force+Linus
```

```
Use%20the%20force%20Linus
```

It is, in fact, this alternate meaning of the character + that requires that + be escaped when the character is trying to represent itself and not a space.

This useful program decodes a URL escaped string into a readable string:

```
url_decode.pl
#!/usr/bin/perl
$cmd_line_input = $ARGV[0]; # Get the command line
input.
$cmd_line_input =~ tr/+/ /; # convert '+' back to
blanks.
$cmd_line_input =~ s/%(..)/chr(hex($1))/ge; # Find all
    appearances of hex numbers of form: %xx
print($cmd_line_input, "\n");
```

With this new knowledge, the first CGI script can be modified to allow input from the client. To do this simply change the line:

```
my $cgi = new CGI {name => 'bill', job =>
     'programmer'};
```

to:

```
my $cgi = new CGI;
```

It is now possible to access the script as before, but also adding the following to the end of the URL:

```
'?name=my+name&job=my+job'
```

For instance, the modified example from the first section would be:

```
http://web.your_server.com/~me/cgi-bin/
hello.cgi?name=my+name&job=my+job
```

Once that is working, try embedding the link into an HTML document. The link in the document would look like:

```
<A HREF="http://web.your_server.com/~me/cgi-bin/
hello.cgi?name=my+name&job=my+job">
          About me</A>
```

ENVIRONMENT INFORMATION AVAILABLE TO THE CGI SCRIPT

Other information is also available to your CGI script, and is passed to your script in environment variables. There are two distinct categories of information.

The first category contains information the server sets up to tell the CGI script more about the environment in which it is running. Unfortunately, because there is no formal standard for the CGI programmer's interface, different servers will set up different information.

The second type of information passed to the CGI script is the contents of the headers for the HTTP request it is handling. The headers are information passed from the HTTP client (Web browser) to the server, as part of the request. This header provides additional information about the request itself. There are a variety of headers fields that can be sent by any given Web browser, but the following is a short list of some of the more common and useful header fields:

- `USER_AGENT`: the name of the HTTP client
- `ACCEPT`: the media types that the client will accept
- `HTTP_HOST`: the domain name used in the request URL

There are many more fields defined by the HTTP/1.1 protocol. To learn more about these other fields; or for a more formal description of the ones mentioned in the above list, refer to RFC 2068. (This is available at *http://www.w3.org*, search for "RFC 2068".)

To see what information is received from the Web server, try the following script which will print out all the information passed into the program through environment variables.

```
env.cgi
#!/usr/bin/perl
use CGI;
my ($key, $val);
my  $cgi = new CGI;
print $cgi->header;
while (($key, $val) = each %ENV) {
        print "<B>$key</B>  = '$val'<BR>\n";
}
```

The sample output below shows a typical result from a request made form the Netscape Navigator™ 4.05 to an Apache 1.1.3 HTTP server.

```
SERVER_SOFTWARE = 'Apache/1.1.3 Debian/GNU'
GATEWAY_INTERFACE = 'CGI/1.1'
DOCUMENT_ROOT = '/var/www'
HTTP_ACCEPT_LANGUAGE = 'en'
SCRIPT_NAME = '/cgi-bin/user_name/env.cgi'
SCRIPT_FILENAME = '/usr/lib/cgi-bin/moore/env.cgi'
REMOTE_ADDR = '209.66.100.145'
SERVER_NAME = 'www.server_host.com'
SERVER_PROTOCOL = 'HTTP/1.0'
HTTP_ACCEPT_CHARSET = 'iso-8859-1,*,utf-8'
REQUEST_METHOD = 'GET'
REMOTE_HOST = 'santacruz-x2-3-145.got.net'
SERVER_PORT = '80'
QUERY_STRING = ''
HTTP_USER_AGENT = 'Mozilla/4.05 [en] (X11; I; Linux
2.1.24 ppc)'
HTTP_HOST = 'machine1.server_host.com'
PATH = '/bin:/usr/bin:/sbin:/usr/sbin'
SERVER_ADMIN = 'root@server_host.com'
HTTP_CONNECTION = 'Keep-Alive'
HTTP_ACCEPT = 'image/gif, image/x-xbitmap, image/jpeg,
image/pjpeg, image/png, */*'
```

The variables which describe the HTTP headers are differentiated from the others by the prefix `HTTP_`. These are the only ones that would change if the request had been made with a different browser.

To learn the exact meanings of the other fields, read the documentation on the CGI protocol for your own HTTP server.

HTML FORMS

Most interaction with CGI scripts is through HTML forms. Since a full explanation of HTML and HTML forms is out of the scope of this particular class, the following example is provided to show a form using the sample program `hello.cgi`:

```
hello.html
<HEADER>
<TITLE>Hello!</TITLE>
</HEADER>

<FORM ACTION="/cgi-bin/moore/hello.cgi" METHOD="POST">
What is your name:<INPUT TYPE="text" NAME="name"></BR>
What is your job :<INPUT TYPE="text" NAME="job"></BR>
<INPUT TYPE="submit">
</FORM>
```

The `FORM` tag has the two parameters `ACTION` and `METHOD`. The `ACTION` parameter specifies the URL of the resource that is meant to handle request, for example, the URL of the CGI script involved, much as an HREF in an anchor tag would.

The `METHOD` parameter specifies which method to use when making the request (`GET` or `POST`).

In between the form tags there can be any HTML except another `FORM` tag. The `INPUT` tag is used to create an interface to the CGI script. Other interface elements include `SELECT` and `TEXTAREA`, which we will not cover in this course. Only two of the parameters to the `INPUT` tag are used, `TYPE` and `NAME`.

TYPE defines how the user will be prompted for the input. The valid values for this parameter are:

- text
- password
- checkbox
- radio
- submit
- reset
- file
- hidden
- image
- button

This lesson describes the types: *text*, *submit*, and *hidden*.

The TYPE text will display a small entry box in which the user can type. It can have a default value by setting an additional parameter VALUE to a string you wish to be the default. Further, its length can be changed by setting the additional parameter SIZE to width in characters.

The HIDDEN type is the same as the TEXT type, except it is not displayed to the user. It is useful for setting default parameters for submission to a CGI script. (The user of the form dose not have to be aware of this. See the cautionary note about this in the security section.)

The SUBMIT type creates a button for the user to depress, causing the form to be submitted to the HTTP server.

For more information on forms or other HTML, see the HTML 4.0 specification at *http://www.w3.org/*.

So far, you should understand how data is passed to your CGI script from the user, as well as how to process data from forms and embedded URLs. Lesson 15, *CGI Script Debugging and Security*, will cover more about Perl before returning to the discussion on programming for the CGI.

LESSON SUMMARY

In this lesson, you have learned:

- How to describe and contrast the `GET` and `POST` methods of passing data from HTTP client to CGI script.
- Additional data available to the CGI script.
- The methods used to encode and decode requests/data sent to the CGI script.
- A basic understanding of how to use HTML forms and other methods for making requests to a CGI script.

REVIEW QUESTIONS

1. When should you use the `GET` method and why?

2. Where should you use the `POST` method and why?

3. Where can you find out what HTTP client a request was made with?

4. Write a URL encoded request to a CGI script at the URL `http://www.people-places.com/cgi-bin/person.cgi` that sends the parameters `full_name` and `age` with your full name and age as the value for the parameters.

5. Where does one put the URL to a CGI script when writing an HTML form to be used to invoke the script?

Answers on page 208

EXERCISE

1. Now you can make a page with several links in it, each passing different information to the CGI script. Try this by making a page with links in it for each member of your family, i.e., mother, father, etc.

2. As a practical exercise, write a CGI script which takes three parameters— *name*, *email*, and *comments*—to be displayed in an output HTML document. Then write a forms interface that exercises this CGI script.

LESSON 15

CGI Script Debugging and Security

OVERVIEW

Debugging a program is the art of fixing the mistakes that cause it to malfunction. This is an important skill that will take time and practice to develop. This lesson provides some tips on how to get started debugging a CGI script as well as a few tricks that can make it much easier.

Three general areas of security problems will also be discussed: denial of service attacks (DOS's), information theft attacks, and break-ins which allow an attacker to gain full or partial access to your computer.

LESSON TOPICS

- CGI Script Debugging
- CGI Script Security

OBJECTIVES

By the end of this lesson, you should be able to:

- Devise an attack plan to debug a CGI script that is malfunctioning
- Obtain information from the servers error log about errors that occurred during the execution of your script.
- Run and debug your CGI scripts from the command line.
- Contrast and describe three different types of security risks associated with CGI scripts.
- Identify and correct dangerous security holes in scripts.

CGI SCRIPT DEBUGGING

Using the `print` Function

If you find that your program does not work, there are several things you can do to aid you in the debugging process. One of the simplest and easiest-to-use debugging tools is Perl's `print` function. To help create bug-free programs, use `print` statements throughout your CGI script while debugging, to check and make sure that the variables possess the expected values. Often, this is all that is required to discover and repair a bug in a program. Using the `print` function is especially useful in debugging CGI scripts, because when the script is running on the server, its output is one of the few clues to know what it is doing during execution.

Using the Error Log

Besides output to the HTTP client, any errors you generate during execution of the script will be recorded in the Web server error log.

If while using `print`, the program appears to execute but produces no output in the HTTP client (browser), try replacing the `print` statement with a `warn` statement. `warn` sends its output the `standard error` which gets collected in the server error log.

A good way to access the information in the error log is to run the command `tail -f` on this file. This command will print output to your screen exactly as it is written to the log file. Once this command is running, any errors that are logged will be seen automatically each time an HTTP request that invokes the script is made. Viewing the errors as they happen is useful, because it is often difficult to tell which errors in the log were caused by the CGI script and which were caused by other programs.

Testing from the Command Line

If the problem with the script cannot be discovered while it is running from the Web server, further testing for bugs may be done from the command line. This can be difficult to accomplish, due to the vastly different environments in which scripts run on the command line and scripts run by the HTTP server run. The `CGI.pm` module provides some features to help simulate the environment of the HTTP server and make the task of debugging a CGI script on the command line a bit easier.

The simplest method `CGI.pm` provides to call a program from the command line is a feature which will directly ask for input in the event that it is unable to detect any input from an HTTP server. To supply the appropriate input, an input string in URL-encoded form must be provided on the command line. After typing this string, type *Control-d* to end input. To test your CGI script several times from the command line with the same input, set some default input when the CGI object is created.

For example:

```
$obj = new CGI { param1 => 'a', param2 => 'b' };
```

This expression will create a CGI object with the parameters `param1` and `param2` set to `a` and `b`, respectively. Scripts which use the CGI module in this way will be able run from the command line without having to give them input each time they are run.

CGI Script Security

When writing CGI scripts, it is imperative to consider the matter of security, because the scripts are essentially an extension to the HTTP server. As

such, any security hole in a script is effectively a security hole in the server. An attacker can leverage even the smallest hole in a CGI script to gain access to the system in which the script resides.

In the event of a successful attack on a server, the result may be not only loss of control over the computer, but also Denial of Service (DOS) attacks in which the server becomes unusable and loss of private of proprietary material can take place. Because of these and other risks, it is critical to be aware of the issues and dangers involved in CGI-scriptwriting.

Denial of Service Attacks

A *Denial of Service* (DOS) attack occurs when an attacker prevents access to one or more of the services the server provides. DOS attacks are usually just malicious and petty attempts to inflict harm, but they can sometimes be part of a more complicated attack to gain control over the server or other computers. This type of attack is usually effected by using up various system resources such as disk space, memory, and/or more abstract resources such as sockets, file descriptors, and processes. Usually one need only worry about those attacks which use up disk space and memory when writing CGI scripts; the web server should protect most other system resources in the event of an attack.

To try and protect memory resources, be cautious of how much data your program reads into memory at once: do not allow your CGI script to read an arbitrarily large amount data from the remote user. When you are using `CGI.pm`, you can limit the amount of data it will read in by setting `$CGI::POST_MAX`. By default `CGI.pm` will read in an infinite amount of data, so setting `$CGI::POST_MAX` is a good idea. The units of `$CGI::POST_MAX` are bytes; therefore setting its value at 1024 bytes would allow a kilobyte (1024 characters) of data to be read in. For most applications, 3000 bytes should be more than enough to process any form, but you may need to set it to a higher amount if your script needs it.

To protect against DOS attacks in which the assailant is trying to starve your system of disk space, you can take similar measures and limit the quantity of data you write to a file for each request. Although this is partially taken care of if you limit the amount of data the remote user can send you (as mentioned above), you must also think about the other types of data your script writes, such as error log and other logs. Make sure that these do not grow so large as to pose a problem. There should be a directory tree where logs are stored in your system.

Due to the way UNIX file systems work, in the event that any of the logs become oversized and fill up the file system, this will only affect any other programs which are logging into that particular file system, and will hopefully not cause any disruptions in the server's other functions.

The effects of a memory-starving DOS attack are usually quite severe. As a result of such an attack, the system is often left with no memory to perform even the most basic and menial of tasks, effectively rendering that system useless. In the worst case scenario, the server may even completely crash.

Disk starvation is, in comparison, usually less spectacular than memory starvation; however, it can make your server equally as useless in other ways. For instance, if the disk in which you log completed orders for online sales is full, you will no longer be able to process orders, effectively shutting down an important part of your Web site. The same is true in the event that the file system which holds your log files fills up: programs will be unable to log anything. An attacker can exploit this situation by committing other illicit acts without any records. Ultimately, DOS attacks are very difficult to completely prevent. Exercising caution regarding this security issue will most likely guarantee survival to all but the most determined attacks.

Information Theft

The second issue of security to be discussed is of greater concern. In this case, the attacker uses your script to gain access to restricted information. There are many reasons why an attacker may be trying to steal data from you. The information itself may be of use to the attacker: it could contain such valuable data as business plans, customers' personal records or credit card information; or the attacker may find the information useful in gaining further access to the server. Such information would likely be password files and/or other system configuration information.

These types of attacks can be completely prevented with careful and judicious programming, as well as thorough examination of all input from clients prior to using it as part of a plan, such as opening a file or running a system command.

The best way to do this is to check the input to make sure it contains only the information you expect from it. For example, imagine you have written the following program:

```
#!/usr/bin/perl

use CGI;
$CGI::POST_MAX = 3000;

my $cgi = new CGI;

my $file = $cgi->param('file');

open(OUT, "/tmp/safe/$file");

    my @lines = <OUT>;

print($cgi->header, @lines);
```

This program is written to read a file from the directory `/tmp/safe`; however, since there are no checks in this program, a devious person could use this program to retrieve any file readable by the web server. This hole in the program would allow an attacker to pass a file name starting with `../../`. This then allows the attacker to obtain any file on the computer rather then just in the directory `/tmp/safe`.

To fix this program, verify that the file name which has been passed in contains only the characters that you expect it to:

- `a-Z`
- `0-9`
- "_"
- "."

The fixed version of the program would look like this:

```
#!/usr/bin/perl

use CGI;
$CGI::POST_MAX = 3000;

my $cgi = new CGI;

my $file = $cgi->param('file'); # only allow local files with
                                #"_", ".", and alphanumeric
                                #characters
$file =~ /^([\.\w\d]+)$/;

unless (defined $1) {
print $cgi->header,
     "<H1>Sorry '$file' is not a valid file name.</H1>";
        die "Attempt to give invade file name '$file'";
}

$file = $1;

open(OUT, "/tmp/safe/$file");

my @lines = <OUT>;

print($cgi->header, @lines);
```

This new version of the program avoids the earlier problem by only allowing file names with alphanumeric characters "_" and dots. If you need to permit files which contain other characters or nested directories, you must change the regular expression; however, the above example will work for the needs of most programs.

Command Execution

As in the above case, the best way to protect yourself from this kind of security hole and its consequences is to carefully scrutinize any data you receive from the client before doing anything with it (this same warning also holds true for the last type of security hole we will discuss). In the case of this security hole, the attacker will try to manipulate your program to execute program code or system commands which you did not intend it to execute.

This type of security hole will occur only if you use one of the Perl functions which compiles code at run time, such as `eval` and `require`, or one of the functions that makes calls to the system such as, back ticks (``), `system`,

`exec` and some uses of pipes. The best policy is to avoid this type of system interaction altogether, thereby eliminating any chance of making a mistake.

If a program does require use of one of these facilities, the following tactics can be applied to help avoid getting into any kind of security trouble.

Data received should be filtered through regular expressions and/or functions which only allow acceptance of data which is expected to be passed in. Yes, this was stated one more time. That is how important this security tactic is. The stress should be on extracting only the expected data, as opposed to trying to remove that data that appears dangerous. The logic behind this is that it is often difficult (impossible, even) to predict all the possible types of dangerous data.

This mistake is very easy to make. Even the people in charge of maintenance of the NCSA HTTP daemon have fallen into this trap. In their distribution, they shipped a CGI program with a security hole which allowed for unsafe handling of system calls. The programmer who wrote the program was careful to check for those shell characters he thought might be dangerous, such as `&`, `;`, etc., before he used the input as part of a system call; but he forgot to check for the newline character `\n`, which allowed for the execution of arbitrary commands as the HTTP server. This problem could have been avoided if, instead of relying on his ability to predict those characters that would be hazardous to system security when passed through to the system, the programmer had simply made sure that input passed through to the system could include only those characters he expected to be used. In essence, the program had a line equivalent to:

```
if ($line =~ /[;&><\(\)]/) { error "danger" }
```

Instead, it should have been something like:

```
if (! ($line =~ s/^([\w\d]+)$/$1/)) { error "bad input" }
```

To aid with this security task, Perl provides a valuable tool to guarantee input has been checked, and ensure it is what is expected, before using it in a manner which could subject the program to security breaches. The tool is called *taint checking*, and it can be activated by running Perl with the `-T` flag. When this tool is activated, Perl does not let data to be used in a risky manner unless it has been passed through a regular expression (as in the example

above). To enable this feature, change the first line of a program to include the `-T`.

Example:

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

If a programmer had used this feature in the first example of this lesson, Perl would not have allowed the program to open the file in an unsafe manner. It would have required that changes be made to the program before allowing the program to execute.

Although this feature is not foolproof, it is a great help towards the end of writing secure CGI scripts.

Using the Safe Module

Another wise idea when writing CGI scripts is to use the `Safe` module instead of `eval` for runtime compilation. The `Safe` module allows code to run in a safe environment, in a manner similar to how Java allows applets to run safely on a Web browser. (See the documentation that comes with Perl for instructions on using this module.)

Dangers with Back Ticks and System Calls

When calling other programs from your script, avoid using back ticks and the single parameter form of the system call. For security, the multi-parameter form of the system call should be used. The reason is that the latter does not invoke a shell to process the command.

Not invoking a shell is a safer method, since it does not attribute special meaning to any particular characters. The following example illustrates the difference:

```
$myfile = "test;rm *";
system("cat $myfile");
```

Or:

```
`cat $myfile`;
```

Versus:

```
system('cat', $myfile);
```

In the above example, the single parameter form of the system and the back ticks interpret the command first as `cat test` and then as the (separate) command `rm*`. The multi-parameter form of the system, however, simply interprets `test;rm*` as an odd-looking file name.

Last but not least, remember that periodic peer review is a great way to catch nasty security bugs in CGI scripts.

LESSON SUMMARY

In this lesson, you have learned:

- How to plan an attack plan to debug a CGI script that is malfunctioning.
- How to use print and warn statements to debug a CGI script.
- How to obtain information from the server's error log to debug a CGI script.
- How to run and debug your CGI scripts from the command line.
- About denial of service attacks and how to protect against them.
- About information theft attacks and how to protect against them.
- About attacks when the attacker attempts to run commands on your server and how to protect against them.
- The safe way of checking your data.
- The safe way of making system calls.

REVIEW QUESTIONS

1. What command can you use to view the change in log files as it happens?

2. Where are two places that you can look for error messages?

3. State one way you can run your CGI script from the command line.

4. What is Taint Checking and how do you use it?

5. What is the safest way to make system calls?

6. What are some of the consequences of not carefully opening files?

7. Why is the following code dangerous and how would you fix it?

```
my $file = $cgi->param('file_name');
system("rm /tmp/$file");
```

Answers on page 208

EXERCISE

1. Fix this program:

```
#!/usr/bin/perl

use CGI;

my $cgi = new CGI;

my (
   $file,   # log file name
   $name,   # user name
   $return, # return header
   @ps,     # result of ps
   );

$file   = $cgi->param('file');
$name   = $cgi->param('name');
$return = $cgi->param('return');

open(LOG, ">>/tmp/$name/$file");

@ps = `ps aux | grep $name`;

print LOG join('\n' @ps);

close LOG;

open HED, "/tmp/-heds/$return";

print $cgi->header,
      join('', <HED>),
      join('<BR>', @ps);
```

ANSWERS

To Review Questions

LESSON 1

Review questions, page 17

1. *http://www.perl.com*
2. Larry Wall wrote the main Perl interpreter. Many others have helped in many aspects of the full distribution and Perl Modules.
3. `-w`.
4. `$var` is a scalar

   `@var` is a list.

   `%var` is a hash table, also often referred to as an associative array.

   `&var` makes a reference to a function, most typically used to invoke the function.
5. No. Names of scalars, lists, hashes, and functions are all in different symbol tables and refer to totally distinct objects.

LESSON 2

Review questions, page 26

1. Common Gateway Interface.
2. The common mistakes are:
   - ➤ Is the URL correct?
   - ➤ Is the server able to execute the program?
   - ➤ Does the server recognize the program as a CGI script?
   - ➤ Is there a problem with the syntax of the program?
3. It acts as a facilitator, accepting the request from the HTTP client (Web browser), passing it on to the CGI script and then accepting the reply from the CGI script and passing it on to the HTTP client.
4. The HTTP document header which specifies what type of document you are returning.

LESSON 3

Review questions, page 40

1. 13, since * has greater precedence than +.
2. $a = (3 + 5) * 2; Which results in 16. The () overrides the default precedence of * and +.
3. 5 gets printed, the value of $a is 6.
4. 6 gets printed, the value of $a is 6.
5. Functions often return a value of false when an error occurs. A TRUE in front of the || operator causes the whole expression to be TRUE without evaluating the second operand. .
6. $c is 1, $d is "" in a string context, 0 in a numeric context; in other words FALSE.
7. $a gets "" for FALSE, $b gets TRUE for true. eq converts the 0 to "0" in string context which is different than "00" == converts "00" to 0 in numeric context which is the same as 0.
8.

```
dog:cat:dog:cat:
```

LESSON 4

Review questions, page 56

1. 5, because the $i < 3 is false enough to make the && be false without evaluating the 2nd operand which would have changed the value to 1.
2. They are all operators to control the loop from within the loop.
   - last - exits the loop.
   - next - goes to the next iteration of the loop without executing the code that follows it.
   - redo - goes around the loop again but does not execute any tests that are at the beginning of the loop.

3. The differences are:
   - `print` outputs a message to `STDOUT` by default.
   - `warn` outputs a message to `STDERR` by default.
   - `die` outputs a message to `STDERR` by default, but also exits the program.
4. `1`, `5`, `6`, `7,` and `8` gives a syntax error because of the absence of the `{ }` braces on the if block

LESSON 5

Review questions, page 70

1. The following gets printed:

```
Anna and Betty are friends
```

```
$a and $b are friends\n
```

2. Line feed, carriage return and a bell sound (alarm) respectively.
3. a) 9.5;

   b) double precision floating point.
4. It repeatedly reads lines from `STDIN`, removes the last character (line feed) and the next to last character. Then it prints the line without these characters.

LESSON 6

Review questions, page 89

1.

```
foreach $str (@an) $str .= " is an animal";
```

```
for (my $i = 0; $i <= $#an; $i++) { $an[$i] .= " is
an animal"; }
```

2.

```
@sorted_animals = sort(@an);
```

3.

```
$result = join("; ", @sorted_animals);
```

4.

```
@newlist = split(/; /, $result);
```

LESSON 7

Review questions, page 100

1.

```
delete $a{'rabbit'};
```

2.

```
$a{'kangoroo'} = 'hop';
```

3.

```
while (($a, $loc) = each %s) { push(@hoppers, $a)
if ($loc eq "hop"); }
```

4. No, the hash table does not keep the order in which the entries are assigned, only the association.

LESSON 8

Review questions, page 117

1. When you open a file to append and write to it, first go to the end of the file, then start adding records.

2.

```
open(FILEHANDLE,  ">>a_file") || die("unable to open
file a_file for appending\n");
```

3.

```
$h = sprintf("%0X", $integer);
```

4. `rename`
5. `unlink`
6. The `-r` operator.
7.

```
$ndays = -M "a_file";
```

LESSON 9

Review questions, page 132

1. It finds the first instance of `abc` or `cda` in the string `$a` from left to right.
2. Because of the parenthesis it captures into `$1` either: `abc` or `cda` or leaves it undefined if there was no match.
3. `aa` since that is the longest run starting from the left of multiple `a`'s.
4. `aaaaaaa`, since this is the longest run of `a`'s from left to right while forcing the fact that the sequence of `a`'s is anchored relative to the end of the string with the $ in the pattern.

5. The second parameter of the substitution is not the pattern being substituted but a function that gets called (effect of the `e`). The return value of the function is what gets substituted by the pattern matched. The substitute would keep marching down the string and doing multiple substitutions based on the fact that the `g` command was given.

LESSON 10

Review questions, page 143

1. In an array by the name of `@_`.

2. `global` variables can be accessed in any scope within a particular package.

   `local` variables can be accessed within the subroutine in which they where defined, and within any subroutine called directly or indirectly. Declaration of another local variable by the same name will overshadow it.

   `my` variables are only defined inside the subroutine within which it is defined.

3. Since the list is being assigned to a scalar, it would evaluate the return list in scalar context, and the result in `$l` would be the size of the return list.

4. Yes, if you put an `&` in front of the name (e.g, `&func();`).

5. No, it is not a good way, because the `@l` would consume all the parameters in `@_`, and the `$a` would get none. When using a list while capturing parameters this way, always make the list the last parameter.

6. Example:

```
. my ($a, $b, @l) = @_;
```

LESSON 11

Review questions, page 153

1.

```
$b = $a->[2];
```

```
$b = ${$a}[2];
```

2.

```
$a = {"dog"=>["hopper", "barker"],
"cow"=>["walker", "mooer"]}
```

3.

```
$a = [1, [2,3], [[1,2], [3,4]]];
        $sum = 0;
        sub traverse { my ($l) = @_;
           my $sum;
           foreach (@{$l}) {
              if ( ref $_ ne "ARRAY" ) { $sum += $_;
              } else {
                  $sum += traverse($_);
              };
           }
           return($sum);
        }
        print traverse($a);
```

An important note to understanding this program, is that in recursion, every call to the function makes a new instance of the `my` local variable. Consequently, each call, when returning, returns the sum of the leaf nodes in that subtree of list.

4.

```
$b = \@l;
```

LESSON 12

Review questions, page 165

1.

```
$c = $angel::a + $angel::b;
```

2. A package hides a symbol table from other programs. This method allows programs to grow and expand without the likelihood that global variables and function names will start colliding with each other. They essentially encapsulate the elements of a piece of code.

3.

```
my $cgi = new CGI {name => 'bill', job =>
'postmaster'};
```

```
my $cgi = CGI->new( {name => 'bill', job =>
'postmaster'} );
```

4.

```
$val = $cgi->param("foo");
```

5. Use `CGI`;

6.

```
foreach (keys %main::) { print("$_\n"); }
```

LESSON 13

Review questions, page 171

1. `text/html`.

2.

```
my $obj = new CGI { param1 => 'hello', param2 =>
'world'};
```

3.

```
$obj->param('cat');
```

LESSON 14

Review questions, page 183

1. You should use the `GET` method when embedding CGI requests into links. You have to use this method because it is the only one you can use to do this.
2. You should use the `POST` method when you are using an HTML form. You should use this method over the `GET` method because the `GET` method often places overly strict limits on the amount of data you can send.
3. This information, if available, would be in the environment variable `HTTP_USER_AGENT`, available throughout the `%ENV` hash.
4.

```
http://www.people-places.com/cgi-bin/
person.cgi?age=40&full_name=Madonna+Ciccone
```

5. In the `ACTION` parameter of a `FORM` tag, for example:

```
<FORM ACTION="/cgi-bin/my.cgi">
```

LESSON 15

Review questions, page 197

1. `tail -f`
2. In the output from the CGI script and in the server error log.
3. When using `CGI.pm`, you can type in a URL-encoded query string on the command line when the program is run, or you can set default parameter values when you crate the CGI object.

4. Taint checking is a Perl feature that forces a script to filter any data that comes from an untrusted source, such as data coming from an HTTP client. Taint requires that the untrusted data be passed through a Perl regular expression. To use taint checking, the programmer adds a `-T` to the `#!/usr/bin/perl` line.

5. The safest way of making system calls is with the multi-parameter form of the system command.

6. Some of the consequences are: loss of private and/or sensitive information and possible further breaches of security.

7. There are two things wrong with the code. First, it does not safely deal with the file name that was passed in. Second, it makes a system call in an unsafe way. The correct code could be:

```
 my $file = $cgi->param('file_name');
     $file =~ /^([\.\w\d]+)$/;

     unless (defined $1) {
        print $cgi->header,
          "<H1>Sorry '$file' is not a valid file
name.</H1>";
        die "Attempt to give invade file name '$file'";
     }

     system("rm",  "/tmp/$file");
```