Perl Programming

Student Guide - Volume 1

DTP-250 Rev C

D61834GC20
Edition 2.0
May 2010
D66908

ORACLE

This page intentionally left blank.

This page intentionally left blank.

Table of Contents

About This Course

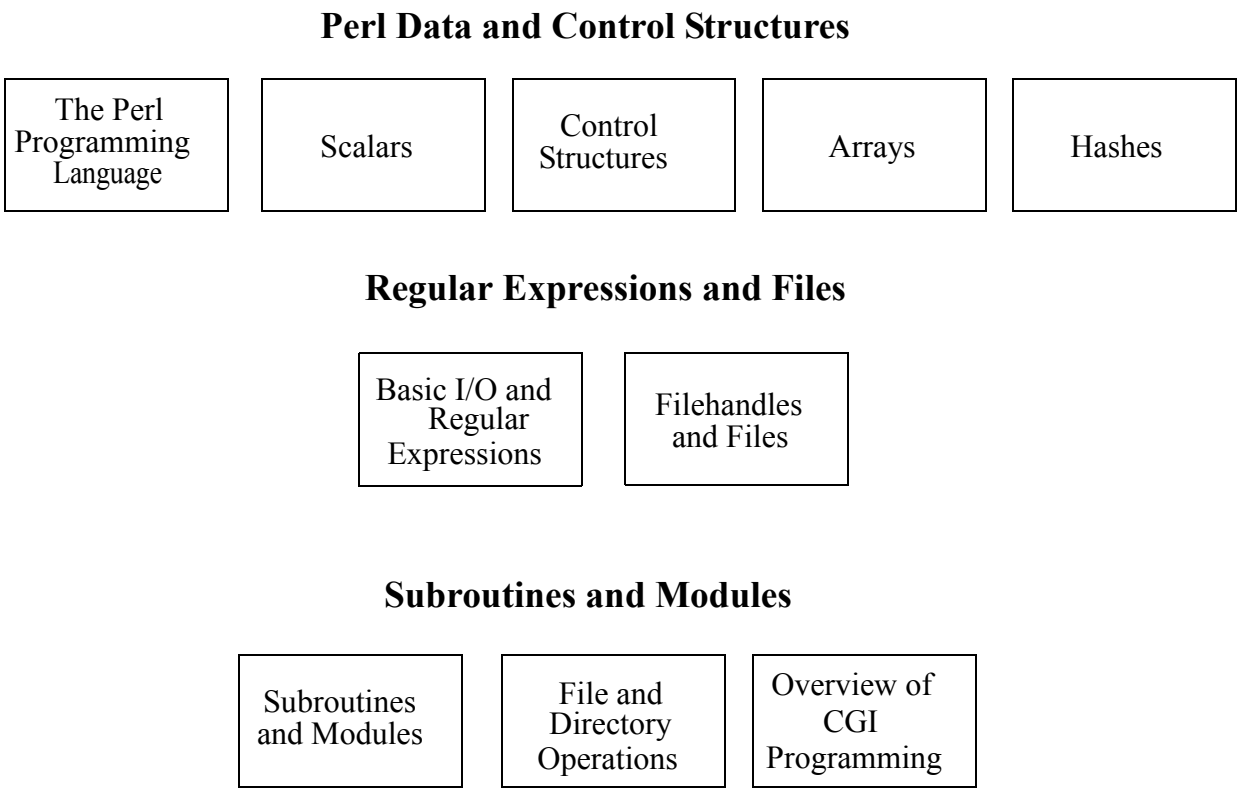
Course Goals

Upon completion of this course, you should be able to:

- Create Perl scripts using scalars, arrays, hashes, and control structures
- Create Perl scripts that perform file I/O and modification
- Create Perl subroutines, packages, and modules using the concepts and data structures described in the course

Course Map

The following course map enables you to see what you will accomplish and where you will go in reference to the course goals.



Topics Not Covered

This course does not cover the following topics. Many of these topics are covered in other courses offered by Sun Services:

- Shell programming – Covered in SA-245: *Shell Programming for System Administrators*

Refer to the Sun Services catalog for specific information and registration.

How Prepared Are You?

To be sure you are prepared to take this course, can you answer yes to the following questions?

- Can you use the `vi` text editor?
- Can you interact with the Solaris™ 10 Operating System as an end user?
- Can you use a graphical user interface?

Introductions

Now that you have been introduced to the course, introduce yourself to the other students and the instructor, addressing the items listed.

- Name
- Company affiliation
- Title, function, and job responsibility
- Experience related to topics presented in this course
- Reasons for enrolling in this course
- Expectations for this course

How to Use Course Materials

To enable you to succeed in this course, these course materials use a learning module that is composed of the following components:

- **Goals** – You should be able to accomplish the goals after finishing this course and meeting all of its objectives.
- **Objectives** – You should be able to accomplish the objectives after completing a portion of instructional content. Objectives support goals and can support other higher-level objectives.
- **Lecture** – The instructor will present information specific to the objective of the module. This information will help you learn the knowledge and skills necessary to succeed with the activities.
- **Activities** – The activities take on various forms, such as an exercise, self-check, discussion, and demonstration. Activities are used to facilitate mastery of an objective.
- **Visual aids** – The instructor might use several visual aids to convey a concept, such as a process, in a visual form. Visual aids commonly contain graphics, animation, and video.

Conventions

The following conventions are used in this course to represent various training elements and alternative learning resources.

Icons

Additional resources – Indicates other references that provide additional information on the topics described in the module.

Discussion – Indicates a small-group or class discussion on the current topic is recommended at this time.

Note – Indicates additional information that can help students but is not crucial to their understanding of the concept being described. Students should be able to understand the concept or complete the task without this information. Examples of notational information include keyword shortcuts and minor system adjustments.

Typographical Conventions

`Courier` is used for the names of commands, files, directories, programming code, and on-screen computer output; for example:

> Use `ls -al` to list all files.
> `system% You have mail.`

`Courier` is also used to indicate programming constructs, such as variable name, subroutines, and keywords; for example:

> The `chomp` function can be used to remove all newline characters from incoming lines.
>
> The `print` statement can be used to print a string.

`Courier bold` is used for characters and numbers that you type; for example:

> To list the files in this directory, type:
> `# ls`

`Courier italic` is used for variables and command-line placeholders that are replaced with a real name or value; for example:

> To delete a file, use the `rm` *`filename`* command.

`Courier italic bold` is used to represent variables whose values are to be entered by the student as part of an activity; for example:

> Type **`chmod a+rwx filename`** to grant read, write, and execute rights for filename to world, group, and users.

Palatino italic is used for book titles, new words or terms, or words that need to be emphasized; for example:

> Read Chapter 6 in the *User's Guide*.
> These are called *class* options.

Module 1

The Perl Programming Language

Objectives

Upon completion of this module, you should be able to:

- Determine your version of Perl
- Identify the default directories searched for Perl library files
- Create a command-line script that prints a simple message
- Create a simple script that prints a simple message
- Test a script's syntax without executing the script

Relevance

Discussion – The following questions are relevant to understanding why Perl is important:

- Why is Perl so popular today?
- Why is Perl called a “glue” language?

Additional Resources

Additional resources – The following references provide additional information on the topics described in this course:

- Schwartz, Randal and Tom Christiansen. *Learning Perl*, 5th Edition. Sebastopol: O'Reilly and Associates, 2008.
- Wall, Larry, Tom Christiansen, and Jon Orwant. *Programming Perl*, 3rd Edition. Sebastopol: O'Reilly and Associates, 2000.
- Christiansen, Tom and Nathan Torkington. *Perl Cookbook*, 2nd Edition. Sebastopol: O'Reilly and Associates, 2003.
- Blank-Edelman, David. *Perl for System Administration*. Sebastopol: O'Reilly and Associates, 2000.
- Cozens, Simon. *Beginning Perl*. Birmingham, UK: Wrox Press, 2000.
- Guelich, Scott, et al., *CGI Programming with Perl*, 2nd Edition. Sebastopol: O'Reilly and Associates, 2000.
- O'Reilly Perl Web site: `http://www.perl.com.`
- Active state Web site: `http://www.activestate.com.`
- CPAN Web site: `http://www.cpan.org/.`
- Perl mongers Web site: `http://www.perl.org.`
- Sun Web Learning Center Web site: `http://suned.sun.com/WLC/.`

Introducing the Perl Programming Language

The Perl programming language was developed by Larry Wall in 1987. Wall was involved in a system administration project in which a homogenous controlling and reporting system was needed for a number of heterogeneous remote hosts. Because existing UNIX® tools, like `awk`, seemed insufficient for this purpose, Wall decided to create his own scripting language.

The new language was named Perl, which stands for Practical Extraction and Report Language. Soon afterward, it was published as public domain software. In the years that followed, Perl has been extended and developed into a general purpose scripting language used in system, network, and database administration. In addition, Perl builds
client-server applications and dynamic Web sites using the Common Gateway Interface (CGI).

Why are so many people interested in Perl? Perl is a very powerful language. It combines many features found in UNIX shell script languages and UNIX commands like `sed`, `awk`, `grep`, `sort`, and `tr`. Perl also includes a number of C library routines and system calls. From word processing functions to UNIX sockets and shared memory mechanisms, Perl provides solutions for nearly every imaginable problem.

A second advantage of Perl is its widespread availability. Perl is available on practically all operating systems. In fact, `http://cpan.org/ports/` lists 100 separate platforms for which precompiled versions exist, and the source code itself is available as a free download. Apart from operating-system-specific functions, Perl programs are portable between different platforms.

Another reason for Perl's popularity is undoubtedly the enormous collection of additional modules. More than 1,500 modules are freely available from the main Perl Web site, `http://www.perl.com`. Whatever the problem, it is likely that someone has already encountered the problem and left a solution in the Perl module collection.

The Perl Interpreter

Perl is not, strictly speaking, an interpreted language: the scripts are both interpreted and compiled. When executed, Perl interprets the commands in the script, compiles them into a platform-independent bytecode, and executes the bytecode in a virtual machine. Compared to UNIX shell scripts, Perl scripts perform hundreds or thousands of times faster because they are compiled first and then executed.

Unlike compiled languages like C, no independent binary program is produced with Perl. This is the main disadvantage of Perl scripts. A Perl script is executable only on hosts on which both the Perl interpreter and the required libraries are present. In addition, performance is impacted because the code has to be recompiled every time the program is started. Both problems will lose their significance when a genuine Perl compiler becomes available. Since Perl version 5.005, a compiler has been included in the Perl distribution as an experimental feature.

Availability

Starting with the Solaris™ 8 Operating System, Perl is included in the Solaris Operating System distribution. For earlier releases of Solaris, Perl can be downloaded from the Sun freeware server, `http://www.sunfreeware.com`, or from the main Web server of the Perl community, `http://www.perl.com`.

Script Basics

To write Perl scripts, you must follow some basic rules, which are described in the following sections.

Statements

Perl uses semicolons as statement separators (not terminators - the last statement in a file, block, or eval does not require one.) So Perl statements look as follows:

```
$x = 0;
print $x;
```

Comments

In Perl, the pound symbol (#) indicates a comment. Everything after a # up to the end of the line is considered a comment and is ignored when a script is executed. Here are some examples.

```
e0101a.plx
  1 #!/usr/bin/perl
  2 # This entire line is a comment
  3 $x = 10; # Only from here to the end is a comment
  4 print $x;

$ e0101a.plx
10 $
```

Line 1 above shows a special comment. This line identifies the location of the Perl interpreter to the operating system. When the file is marked executable, if `#!` is found on the first line of the file, the path that follows is used to compile and execute the code in the file. This line is included in all Perl scripts.

Line 2 shows a comment that takes an entire line. Line 3 show a very common way to use comments. The statement to the left of # executes, and a description of the statement appears to the right of #.

Notice that when the script executes, the value of `$x` is printed, but the prompt appears on the same line. To make our output look better, a `newline` character must be added to it.

Printing

The `print` command sends data to the standard output device, which is usually your screen. The command is very flexible and can handle a number of different data types.

The script that follows updates the previous script and uses a `newline` character to clean up the output.

```
e0101b.plx
  1 #!/usr/bin/perl
  2 # This whole line is a comment
  3 $x = 10; # Only from here to the end is a comment
  4 print $x, "\n";
  5 print "$x\n";

$ e0101b.plx
10
10
$
```

On Line 4, a `newline` character, `\n`, is added to the print statement. This puts the output from `print` on a line by itself. Values separated by commas are printed on the same line. However, the more aesthetically appealing approach is shown on Line 5. Here, the variable and the `newline` character are included in quotes. This approach is used throughout this course for printing out values. Various methods of printing are explained in more detail in Module 2, “Scalars.”

In addition to printing the `newline` character and numbers, strings can be printed. Some examples are shown below.

```
e0102.plx
  1 #!/usr/bin/perl
  2
  3 # Print hello friends
  4 print "Hello friends\n" ;
  5 print "Hello " ;
  6 print "friends\n" ;
  7
  8 # Print 16
  9 $x = 16;
 10 print "$x\n";

$ e0102.plx
Hello friends
Hello friends
16
```

Notice in Lines 4–6, strings are printed using one or two statements. Lines 9–10 demonstrate printing a number using a variable.

Variables

Simple variables that contain a single value (for example, a string or a number) are called *scalar variables*. Scalar variables are identified by placing a dollar sign before the variable name. Below are some example scalar variables.

```
e0103.plx
  1 #!/usr/bin/perl
  2 # Assign integer values
  3 $age = 34 ;
  4 $y = -23 ;
  5
  6 # Assign floating point or scientific notation
  7 $pi = 3.14159 ;
  8 $C = 1.6021e-19 ;
  9
 10 # Assign Strings
 11 $name1 = "Henry" ;
 12 $name2 = "Joe Slow" ;
 13
 14 # Print
 15 print "$age $y\n";
 16 print "$pi $C\n";
 17 print "$name1 $name2\n";

$ e0103.plx
34 -23
3.14159 1.6021e-19
Henry Joe Slow
```

Lines 3–4 and 7–8 demonstrate the different types of numbers that can be assigned to variables. Lines 11–12 assign string values to variables. Notice that variables names use the same format whether they hold numbers or strings. Finally, the variables are printed out on Lines 15–17.

Simple Operators

Scalar variables containing numeric values can be involved in numeric operations. The simplest numeric operators are +, -, *, %, /, and **.

e0104.plx

```
 1 #!/usr/bin/perl
 2 # Simple addition and subtraction
 3 $b = 6 + 1 ;
 4 $new = $b - 3 ;
 5
 6 # Simple multiplication and division
 7 $c = $new * 4 ;
 8 $d = $c / 2  ;
 9
10 # Exponentiation
11 $q = $new ** 2 ;
12
13 #print
14 print "b = $b\n" ;
15 print "new = $new\n" ;
16 print "c = $c\n" ;
17 print "d = $d\n" ;
18 print "q = $q\n" ;
```

```
$ e0104.plx
b = 7
new = 4
c = 16
d = 8
q = 16
```

In addition to these symbolic operators, many built-in numeric functions exist, such as sqrt, exp, sin, and cos. These are described in “Arithmetic Functions” on page 2-15.

```
$x = sqrt(16) ; # square root of 16 equals 4
```

Notes

Invoking a Perl Script

Now that the basics of Perl scripts have been reviewed, it is time to actually run a script. There are three ways to run a Perl script.

- The first method uses the Perl interpreter on the command line along with the `-e` switch (execute switch).

```
$ perl -e 'print "Welcome to our Perl Course!\n"'
Welcome to our Perl Course!
```

  To test syntax or execute simple statements, use Perl with the `-e` switch. In the previous example, one `print` statement is executed.

- The second method passes the script file to the Perl interpreter as an argument.

```
e0106.plx
  1
  2 print "Welcome to our Perl Course!\n";

$ perl e0106.plx
Welcome to our Perl Course!
```

  The script is stored in a file and invoked with the file name as an argument. The interpreter and most UNIX operating systems do not require any specific file extensions for Perl scripts.

Note – On Microsoft Windows operating systems, Perl scripts require an extension to be associated with Perl. The extension `.plx` is preferred; the use of the formerly used `.pl` is deprecated, since it conflicts with Perl libraries.

- The third way is to make the script file executable. This requires a slight modification to the previous example.

```
e0107.plx
  1 #!/usr/bin/perl
  2 print "Welcome to our Perl Course!\n";

$ chmod +x e0107.plx
$ e0107.plx
Welcome to our Perl Course!
```

In addition to adding `#!/usr/bin/perl` to the first line, execute permissions must be assigned to the script with `chmod`. This allows the script to be invoked directly on the command line. The shell uses the first line to locate and call the Perl interpreter. The Perl interpreter then compiles and executes the script.

Command-Line Options

There are a number of command-line options that change Perl's behavior. This section reviews some of the more important switches. For a complete list of command-line switches, type:

```
perl -h
```

The -e Execute Option

This switch executes a script from the command line as described in “Invoking a Perl Script” on page 1-12.

The -w Warning Option

This is a useful switch for troubleshooting; in fact, it should be used every time that you execute a script! It provides the following information:

- Variables that are used only once
- Subroutines that are defined for a second time
- Syntactically correct, but suspicious constructions

This switch is described in more detail in “Initializing and Assigning Values” on page 2-4.

The -c Check Syntax Option

This switch causes Perl to check the syntax of a file without executing the script. It is the "dry run" equivalent of "-w".

```
$ perl -c e0107.plx
e0107.plx syntax OK
```

The -n Option

Use this switch to loop through the specified file. The commands that you specify will apply to every line of that file. By default, the lines will not be printed. For example:

```
$ perl -wne "print" e0107.plx
#!/usr/bin/perl -w
print "Welcome to our Perl Course!\n";
```

The previous command reads the file `e0107.plx` and prints each line of the file to standard output.

The -v Version Option

This switch displays the version Perl installed on the system.

```
$ perl -v

This is perl, v5.8.4 built for sun4-solaris-64int
(with 28 registered patches, see perl -V for more detail)

Copyright 1987-2004, Larry Wall

Perl may be copied only under the terms of either the Artistic License or
the GNU General Public License, which may be found in the Perl 5 source
kit.

Complete documentation for Perl, including FAQ lists, should be found on
this system using `man perl' or `perldoc perl'.  If you have access to
the Internet, point your browser at http://www.perl.com/, the Perl Home
Page.
```

The -V verbose Option

This switch displays detailed information about your Perl environment, including the operating system, the default compiler settings, and the default directories included in library searches.

```
$ perl -V
Summary of my perl5 (revision 5 version 8 subversion 4) configuration:
  Platform:
    osname=solaris, osvers=2.10, archname=sun4-solaris-64int
    uname='sunos localhost 5.10 sun4u sparc SUNW,Ultra-2'
    config_args=''
    hint=recommended, useposix=true, d_sigaction=define
    usethreads=undef use5005threads=undef useithreads=undef usemultiplicity=undef
    useperlio=define d_sfio=undef uselargefiles=define usesocks=undef
    use64bitint=define use64bitall=undef uselongdouble=undef
    usemymalloc=n, bincompat5005=undef
  Compiler:
    cc='cc', ccflags ='-D_LARGEFILE_SOURCE -D_FILE_OFFSET_BITS=64 -xarch=v8 -
D_TS_ERRNO',
    optimize='-xO3 -xspace -xildoff',
    cppflags=''
    ccversion='Sun WorkShop', gccversion='', gccosandvers=''
    intsize=4, longsize=4, ptrsize=4, doublesize=8, byteorder=87654321
    d_longlong=define, longlongsize=8, d_longdbl=define, longdblsize=16
    ivtype='long long', ivsize=8, nvtype='double', nvsize=8, Off_t='off_t', lseeksize=8
    alignbytes=8, prototype=define
  Linker and Libraries:
    ld='cc', ldflags =''
    libpth=/lib /usr/lib /usr/ccs/lib
    libs=-lsocket -lnsl -ldl -lm -lc
    perllibs=-lsocket -lnsl -ldl -lm -lc
    libc=/lib/libc.so, so=so, useshrplib=true, libperl=libperl.so
    gnulibc_version=''
  Dynamic Linking:
    dlsrc=dl_dlopen.xs, dlext=so, d_dlsymun=undef, ccdlflags='-R
/usr/perl5/5.8.4/lib/sun4-solaris-64int/CORE'
    cccdlflags='-KPIC', lddlflags='-G'

Characteristics of this binary (from libperl):
  Compile-time options: USE_64_BIT_INT USE_LARGE_FILES
  Locally applied patches:
        22667 The optree builder was looping when constructing the ops
...
        22715 Upgrade to FileCache 1.04
        22733 Missing copyright in the README.
        22746 fix a coredump caused by rv2gv not fully converting a PV
...
        22755 Fix 29149 - another UTF8 cache bug hit by substr.
        22774 [perl #28938] split could leave an array without ...
```

```
        22775 [perl #29127] scalar delete of empty slice returned garbage
        22776 [perl #28986] perl -e "open m" crashes Perl
        22777 add test for change #22776 ("open m" crashes Perl)
        22778 add test for change #22746 ([perl #29102] Crash on assign
...
        22781 [perl #29340] Bizarre copy of ARRAY make sure a pad op's
...
        22796 [perl #29346] Double warning for int(undef) and abs(undef)
...
        22818 BOM-marked and (BOMless) UTF-16 scripts not working
        22823 [perl #29581] glob() misses a lot of matches
        22827 Smoke [5.9.2] 22818 FAIL(F) MSWin32 WinXP/.Net SP1 (x86/1
cpu)
        22830 [perl #29637] Thread creation time is hypersensitive
        22831 improve hashing algorithm for ptr tables in perl_clone: ...
        22839 [perl #29790] Optimization busted: '@a = "b", sort @a' ...
        22850 [PATCH] 'perl -v' fails if local_patches contains code
snippets
        22852 TEST needs to ignore SCM files
        22886 Pod::Find should ignore SCM files and dirs
        22888 Remove redundant %SIG assignments from FileCache
        23006 [perl #30509] use encoding and "eq" cause memory leak
        23074 Segfault using HTML::Entities
        23106 Numeric comparison operators mustn't compare addresses of
...
        23320 [perl #30066] Memory leak in nested shared data structures
...
        23321 [perl #31459] Bug in read()
        SPRINTF0 - fixes for sprintf formatting issues - CVE-2005-3962
  Built under solaris
  Compiled at Feb 13 2006 05:12:02
  @INC:
    /usr/perl5/5.8.4/lib/sun4-solaris-64int
    /usr/perl5/5.8.4/lib
    /usr/perl5/site_perl/5.8.4/sun4-solaris-64int
    /usr/perl5/site_perl/5.8.4
    /usr/perl5/site_perl
    /usr/perl5/vendor_perl/5.8.4/sun4-solaris-64int
    /usr/perl5/vendor_perl/5.8.4
    /usr/perl5/vendor_perl
    .
```

The use of 5.010 Pragma

One of the features of Perl is its pragmas. Pragmas alter the compilation or execution of a Perl program. To have your script access the Perl 5.10 features, it is necessary to include the line: ' use 5.010;'. Without this line, the Perl compiler will not bring in some of the new features of Perl 5.10.

The developers of Perl did this to insure that older scripts could still run under Perl 5.10. For example, some servers may move to Perl 5.10 even though some of its users may not be ready to use it. Other users may want to take advantage of the faster execution and smaller memory footprint of Perl 5.10 for their older Perl scripts. However, some of the new features of Perl 5.10 could cause some of the older scripts to break. So, by default, scripts without this line will not be able to access the new features of Perl 5.10. It will not be necessary for users to maintain two versions of Perl.

Let's look at an example. Perl 5.10 has a new function: say. The say function works just like the print function except that it appends a new line at the end. An example is shown below.

e0108.plx

```
 1 #!/usr/bin/perl
 2
 3 use 5.010;
 4
 5 # Print hello friends
 6 print "Hello friends\n" ;
 7 say "Hello friends";
 8 $a = "Hello friends";
 9 print "$a\n";
10 say "$a";
```

```
$ e0108.plx
    Hello friends
    Hello friends
    Hello friends
    Hello friends
```

However, if you comment out line 3 and try to run the script, the following error message will be generated:

```
$ e0108b.plx

String found where operator expected at e0108.plx line 7, near "say
"Hello friends""
        (Do you need to predeclare say?)
String found where operator expected at e0108.plx line 10, near "say"$a""
        (Do you need to predeclare say?)
syntax error at e0108.plx line 7, near "say "Hello friends""
syntax error at e0108.plx line 10, near "say "$a""
Execution of e0108.plx aborted due to compilation errors.
```

Documentation and Help

Documentation and help information about Perl is included in a manual page (man page) collection. To view the documentation, start by typing the following:

```
man perl
```

Additional topics are described in more than 20 separate man pages. As a starting point, consider reviewing the following man pages:

- `perlrun` (provides information about execution)
- `perldata` (provides information about data structures)
- `perlsyn` (provides information about syntax)
- `perlop` (provides information about operators)
- `perlfunc` (provides information about built-in functions)
- `perlfaq` (provides information about frequently asked questions)
- `perlre` (provides information about regular expressions)

Configuring Man Pages for Perl

To use the Perl man pages, make sure you add the following directory to your `$MANPATH` environment variable.

```
/usr/perl5/man
```

For example:

```
MANPATH=$MANPATH:/usr/perl5/man
```

The perldoc Utility

Another way to access the man pages is by using the perldoc utility. Type perldoc and the name of a man page as an argument. For example:

```
perldoc perlrun
```

In addition to using the perldoc basic command, you can search the documentation using the -f option. For example, type the following to search the documentation for any page containing information about the print statement:

```
perldoc -f print
```

To search only the frequently asked questions, type the following:

```
perldoc -q print
```

Other Information Sources

Further information can be found on several Web servers, mainly from http://www.perl.com, the official Web site of the world-wide Perl community.

ActiveState (a provider of Linux, Solaris, and Microsoft Windows Perl distributions) provides copies of all the man pages online. You can find them at the following Universal Resource Locator (URL):

```
http://velocity.activestate.com/docs/ActivePerl
```

Several newsgroups focus on the Perl language too. Some groups you might find interesting include:

```
comp.lang.perl.misc
comp.lang.perl.moderated
comp.lang.perl.module
```

Exercise: Create Basic Perl Scripts

In this exercise, you perform the following tasks:

- Determine your version of Perl
- Identify the default directories searched for Perl library files
- Create a command-line script that prints a simple message
- Create a script that prints a simple message
- Test a script's syntax without executing the script

Preparation

Make sure you have a system with Perl 5 or later installed and a text editor you are comfortable using.

Tasks

Complete the following steps:

1. Determine the version of Perl that is installed on your system. Write down the version installed on your system and the command you used to determine this in the space provided below.

2. Determine what directories are searched for library files with your version of Perl. List the command used to find these values.

3. Execute a Perl script on the command line that prints a message like the following:

```
Welcome
to
Perl!
$
```

4. Create a script file that produces the same output as the previous example. Use the print function. Execute the script using the Perl interpreter on the command line.

```
$ perl lab0104.plx
Welcome
to
Perl!
$
```

5. Verify the syntax of the file you created previously without executing the script.

6. Make the script file you created previously executable. Execute the script file by itself on the command line.

7. Change the script file created previously to use the say function instead of the print function. Execute the script file by itself on the command line.

Exercise Summary

Discussion – Take a few minutes to discuss the experiences, issues, or discoveries you had during the lab exercises.

- Experiences
- Interpretations
- Conclusions
- Applications

Exercise Solutions

1. Determine the version of Perl that is installed on your system. Write down the version installed on your system and the command you used to determine this in the space provided below.

   Suggested solution:

```
$ perl -v
This is perl, version 5..8.4 built for sun4-solaris-64int (with 28
registered patches, see perl -V for more detail)

Copyright 1987-2004, Larry Wall

Perl may be copied only under the terms of either the Artistic License or
the GNU General Public License, which may be found in the Perl 5 source
kit.

Complete documentation for Perl, including FAQ lists, should be found on
this system using 'man perl' or 'perldoc perl'. If you have access to the
Internet, point your browser at http://www.perl.com/, the Perl Home Page.
```

2. Determine what directories are searched for library files with your version of Perl. List the command used to find these values.

   Suggested solution:

```
$ perl -V
Summary of my perl5 (revision 5 version 8 subversion 4) configuration:
  Platform:
    osname=solaris, osvers=2.10, archname=sun4-solaris-64int
    uname='sunos localhost 5.10 sun4u sparc SUNW,Ultra-2'
    config_args=''
    hint=recommended, useposix=true, d_sigaction=define
    usethreads=undef use5005threads=undef useithreads=undef
usemultiplicity=undef
    useperlio=define d_sfio=undef uselargefiles=define usesocks=undef
    use64bitint=define use64bitall=undef uselongdouble=undef
    usemymalloc=n, bincompat5005=undef
  Compiler:
    cc='cc', ccflags ='-D_LARGEFILE_SOURCE -D_FILE_OFFSET_BITS=64 -
xarch=v8 -D_TS_ERRNO',
    optimize='-xO3 -xspace -xildoff',
    cppflags=''
    ccversion='Sun WorkShop', gccversion='', gccosandvers=''
    intsize=4, longsize=4, ptrsize=4, doublesize=8, byteorder=87654321
    d_longlong=define, longlongsize=8, d_longdbl=define, longdblsize=16
    ivtype='long long', ivsize=8, nvtype='double', nvsize=8,
Off_t='off_t', lseeksize=8
    alignbytes=8, prototype=define
  Linker and Libraries:
    ld='cc', ldflags =''
    libpth=/lib /usr/lib /usr/ccs/lib
    libs=-lsocket -lnsl -ldl -lm -lc
    perllibs=-lsocket -lnsl -ldl -lm -lc
    libc=/lib/libc.so, so=so, useshrplib=true, libperl=libperl.so
    gnulibc_version=''
  Dynamic Linking:
    dlsrc=dl_dlopen.xs, dlext=so, d_dlsymun=undef, ccdlflags='-R
/usr/perl5/5.8.4/lib/sun4-solaris-64int/CORE'
    cccdlflags='-KPIC', lddlflags='-G'

Characteristics of this binary (from libperl):
  Compile-time options: USE_64_BIT_INT USE_LARGE_FILES
  Locally applied patches:
        22667 The optree builder was looping when constructing the ops
...
        22715 Upgrade to FileCache 1.04
        22733 Missing copyright in the README.
```

```
        22746 fix a coredump caused by rv2gv not fully converting a PV
...
        22755 Fix 29149 - another UTF8 cache bug hit by substr.
        22774 [perl #28938] split could leave an array without ...
        22775 [perl #29127] scalar delete of empty slice returned garbage
        22776 [perl #28986] perl -e "open m" crashes Perl
        22777 add test for change #22776 ("open m" crashes Perl)
        22778 add test for change #22746 ([perl #29102] Crash on assign
...
        22781 [perl #29340] Bizarre copy of ARRAY make sure a pad op's
...
        22796 [perl #29346] Double warning for int(undef) and abs(undef)
...
        22818 BOM-marked and (BOMless) UTF-16 scripts not working
        22823 [perl #29581] glob() misses a lot of matches
        22827 Smoke [5.9.2] 22818 FAIL(F) MSWin32 WinXP/.Net SP1 (x86/1
cpu)
        22830 [perl #29637] Thread creation time is hypersensitive
        22831 improve hashing algorithm for ptr tables in perl_clone: ...
        22839 [perl #29790] Optimization busted: '@a = "b", sort @a' ...
        22850 [PATCH] 'perl -v' fails if local_patches contains code
snippets
        22852 TEST needs to ignore SCM files
        22886 Pod::Find should ignore SCM files and dirs
        22888 Remove redundant %SIG assignments from FileCache
        23006 [perl #30509] use encoding and "eq" cause memory leak
        23074 Segfault using HTML::Entities
        23106 Numeric comparison operators mustn't compare addresses of
...
        23320 [perl #30066] Memory leak in nested shared data structures
...
        23321 [perl #31459] Bug in read()
        SPRINTF0 - fixes for sprintf formatting issues - CVE-2005-3962
  Built under solaris
  Compiled at Feb 13 2006 05:12:02
  @INC:
    /usr/perl5/5.8.4/lib/sun4-solaris-64int
    /usr/perl5/5.8.4/lib
    /usr/perl5/site_perl/5.8.4/sun4-solaris-64int
    /usr/perl5/site_perl/5.8.4
    /usr/perl5/site_perl
    /usr/perl5/vendor_perl/5.8.4/sun4-solaris-64int
    /usr/perl5/vendor_perl/5.8.4
    /usr/perl5/vendor_perl
    .
$
```

3. Execute a Perl script on the command line that prints a message like the following:

```
Welcome
to
Perl!
$
```

Suggested solution:

```
$ perl -we 'print "Welcome\nto\nPerl!\n";'
Welcome
to
Perl!
$
```

4. Create a script file that produces the same output as the previous example. Use the print function. Execute the script using the Perl interpreter on the command line.

```
$ perl lab0104.plx
Welcome
to
Perl!
$
```

Suggested solution:

```
lab0104.plx
  1 #!/usr/bin/perl -w
  2 print "Welcome\nto\nPerl!\n";
  3
```

5. Verify the syntax of the file you created previously without executing the script.

Suggested solution:

```
$ perl -c lab0104.plx
lab0104.plx syntax OK
$
```

6. Make the script file you created previously executable. Execute the script file by itself on the command line.

   Suggested solution:

```
$ ls -l
total 4
-rw-r--r--   1 mw119255 23966         47 Nov 29 13:19 lab0104.plx
-rwxr-xr-x   1 mw119255 23966         55 Nov 29 13:16 temp.plx

$ chmod a+x lab0104.plx
$ ls -l
total 4
-rwxr-xr-x   1 mw119255 23966         47 Nov 29 13:19 lab0104.plx
-rwxr-xr-x   1 mw119255 23966         55 Nov 29 13:16 temp.plx
$ lab0104.plx
Welcome
to
Perl!
$
```

7. Change the script file created previously to use the say function instead of the print function. Execute the script file by itself on the command line.

```
$ lab0107.plx
Welcome
to
Perl!
$
```

   Suggested solution:

```
#!/usr/bin/perl -w
use 5.010;
say "Welcome\nto\nPerl!\n";
```

Module 2

Scalars

Objectives

Upon completion of this module, you should be able to:

- Define and describe numeric and string scalar data
- Create string and numeric scalar variables
- Modify scalar variables using operators
- Print scalar values using single and double quotation marks
- Remove `\n` from user input using the `chomp` command
- Calculate a value using scalar operators based on user input

Relevance

Discussion – The following question is relevant to understanding scalar variables:

What is a variable used for?

Scalar Variables

What is a variable? It is a container that holds some value. In Perl, a scalar variable holds a single string or a numeric value.

Legal Variable Names

Scalar variables are created using the following rules:

- Each variable name is prefixed with a `$`.
- Variable names consist of alphanumeric characters or underscores.
- Variable names are case sensitive. For example, `$x` is different from `$X`.
- Variable names may not start with a digit.

A variable name can start with a letter or the underscore after the `$`. Because there are a number of defined special variables that use special characters, such as `_` and `/`, avoid using these characters as the first character in your variable name.

The following is a list of valid variables names:

```
$a
$A_long_var_name
$ALongVarName
$alongvarname
```

Here are some invalid variables names:

```
a              # No $
$Another-var   # A - is not allowed in a name
$another.var   # A . is not allowed in a name
```

Initializing and Assigning Values

Initializing and assigning values in Perl is pretty straightforward. List the variable name, an equals sign, and a value. Some examples of assigning numbers and strings are shown as follows.

```
$a = 1;
$num1 = 55.3;
$num2 = 445;

$string1 = "hello";
$string2 = "world";
$c = "A much bigger string";
$d = undef;
```

All the previous examples assign strings and numbers to variables except for the last example. On the last line, `$d` is set to `undef` or undefined. The value returned from a variable set to `undef` depends upon the context in which it is used. In a numeric context, `undef` returns a 0. In a string context, `undef` returns an empty string (`""`).

Perl doesn't require variables to be initialized. If a variable is used for the first time without being initialized, Perl automatically assigns `undef` to it. This feature in Perl can produce interesting results. For example, the following short script tries to increase the value of `$var` by 1.

```
e02w.plx
  1 #!/usr/bin/perl -w
  2 $var = 15;
  3 $var = $vrr + 1;
  4 print $var
```

The intent of the programmer is to add 1 to the current value of `$var` to make `$var` equal to 16. Because Perl allows uninitialized variables, the typographic error (typo) on Line 3 causes Perl to create a new variable, `$vrr`, and set its value to `undef` (`0` in this context.) Thus, `$var` is set to the value of `$vrr + 1`, which is (0 + 1) or 1. This is not the desired result.

Note – Context is a central concept in Perl. Perl's philosophy is that an operation determines the type of a value. That is, an operator or an assignment sets up a specific context, in that its arguments are interpreted. Perl decides by operators (`+`, `-`, `.`, or `x`) whether data is evaluated in a numeric or string context.

To catch a typo like this, Perl provides the `-w` switch, which was introduced in "The -w Warning Option" on page 1-14. By adding this switch to the `#!/usr/bin/perl` line of your script, Perl checks for three things.

- Variables that are used only once
- Subroutines that are defined for a second time
- Syntactically correct, but suspicious constructions

```
e02w.plx
  1 #!/usr/bin/perl -w
  2 $var = 15;
  3 $var = $vrr + 1;
  4 print $var
```

Notice on Line 1, the `-w` switch is added to the end of the line. Now, when the script is executed, Perl checks for variables that are used only once. When the script is executed, the following warning message appears.

```
$ e02w.plx
Name "main::vrr" used only once: possible typo at ./e02w.plx
line 3.
Use of uninitialized value at ./e02w.plx line 3.
```

The fact that `$vrr` is used only once (and probably is a typo) is caught by the `-w` switch. This switch helps when you are trying to troubleshoot a script. Many Perl developers always include this switch to catch this sort of problem. In fact, all the example scripts that follow include this switch.

Scalar Data

This section examines how values are assigned to scalar variables.

Numeric Data

All numbers in Perl are stored and manipulated as double-precision floating-point numbers - even integers. In almost all cases, ambiguities in Perl are internally resolved by "throwing memory at the problem", and this is no exception: floats take up a bit more room than integers. Fortunately, this is not a limitation on modern systems.

Floats are the Perl type used to store floating-point numbers. To make a scalar variable a float, declare a number with a decimal point or scientific notation. Positive or negative numbers can be represented. For example:

```
2.501          # Floating point number
1.5e11         # Float in Scientific notation
-2.501         # Negative number
-2.9e8         # Negative in Scientific notation
```

Integers

Integers are used to store whole numbers, both positive and negative. Large numbers cannot be separated by commas, but an underscore can be used instead to break up large numbers. For example:

```
25             # Integer
-35            # Negative integer
1_000_000      # Long Integer
```

Integers can be represented in numbering systems other than base 10. For example, numbers can also be represented in octal, hexadecimal, or binary.

```
021            # 17 in octal
0x11           # 17 in hexadecimal
0b10001        # 17 in binary
```

String Data

String data consists of sequences of characters that are used to represent a character, word, or phrase (for example, Hello World). Strings are from zero characters in length (represented by `""`) to the amount of available memory. In Perl, strings are defined using either single or double quotation marks. If you forget to quote them, the "-w" mechanism will issue a warning about "barewords".

e0201.plx

```
 1 #!/usr/bin/perl -w
 2
 3 $hellos='Hello World\n'; #Single-Quoted String
 4
 5 $hellod="Hello World\n"; #Double-Quoted String
 6
 7 print 'Single Quoted:\n';
 8 print $hellos;
 9
10 print "\n";
11 print "Double Quoted:\n";
12 print $hellod;
```

```
$ e0201.plx
Single Quoted:\nHello World\n
Double Quoted:
Hello World
```

Single Quotation Marks ("Strong" Quotes)

When strings are created with single quotation marks, all characters are taken literally. So on lines 7–8, instead of printing a newline character (`\n`), the characters are taken literally and a backslash and the letter "n" are printed out.

Double Quotation Marks ("Weak" Quotes)

With double quotation marks, special characters or escape characters are converted into the characters they represent, and variables (except for hashes, which are not meant to be printed directly) are interpolated and interpreted. In Lines 10--12, the `\n` character is converted into a newline. These special characters allow rudimentary formatting of text. In addition to `\n`, use a `\t` for a tab, a `\b` for a backspace, and a`\f` for a formfeed.

Escape Characters

Table 2-1 lists the escape characters available in double-quoted strings.

Table 2-1 Escape Characters

Escape Sequence	Formatting Function
`\n`	Newline
`\r`	Return
`\t`	Tab
`\f`	Formfeed
`\b`	Backspace
`\a`	Bell
`\e`	Escape
`\007`	Any octal ASCII value (007 = bell)
`\x0a`	Any hex ASCII value (0a = linefeed)
`\cZ`	Any control character, in this case, Control-Z
`\\`	Backslash
`\"`	Double quotation marks
`\l`	Put the next letter in lowercase
`\L`	Put all following letters until `\E` in lowercase
`\u`	Put the next letter in uppercase
`\U`	Put all following letters until `\E` in uppercase
`\Q`	Backslash-quote all nonalphanumerics until `\E`
`\E`	Terminate the `\L`, `\U`, or `\Q` characters described previously

Note – To display a `\` or `"` in a double-quoted string, use `\\` and `\"` respectively.

Variable Interpolation

Double-quoted strings provide one additional feature: variable interpolation. When a string variable is printed out inside of double quotation marks, the value the variable contains is substituted for the variable name. The following example demonstrates variable interpolation.

e0202.plx

```
1 #!/usr/bin/perl -w
2 # Variable Interpolation
3
4 $hello="Hello World"; # String var
5
6 print 'Single Quoted: $hello\n';
7
8 print "\n";
9 print "Double Quoted: $hello\n";
```

```
$ e0202.plx
Single Quoted: $hello\n
Double Quoted: Hello World
```

On Line 6, when the variable name is put in single quotation marks, the variable name is printed out. On Line 9, when the variable name is put in double quotation marks, the value the variable holds is printed out.

Quoting Operators

How can the same output be produced without using single or double quotation marks? The quoting operators perform the same functions as quotation marks but use a delimiter to enclose text. The quoting operators and their equivalents are:

`qq` – Performs the same function as double quotation marks

`q` – Performs the same function as single quotation marks

The following script produces the same output as the previous script, but uses the q and qq quoting operators.

```
e0203a.plx
  1 #!/usr/bin/perl -w
  2 # Variable Interpolation
  3
  4 $hello="Hello World"; # String var
  5
  6 print q/Single Quoted: $hello\n/;
  7
  8 print "\n";
  9 print qq/Double Quoted: $hello\n/;
```

```
$ e0203a.plx
Single Quoted: $hello\n
Double Quoted: Hello World
```

The same output is produced. What are the benefits of using these operators? They make it easier to print strings that contain quotation marks. The following script demonstrates how quotation marks can be displayed in a print statement.

```
e0203b.plx
  1 #!/usr/bin/perl -w
  2 # Variable Interpolation
  3
  4 $hello="Hello World"; # String var
  5
  6 print '$hello = \'Hello World\'';
  7 print "\n"; # Add a new line
  8 print q/$hello = 'Hello World'/;
  9
 10 print "\n"; # Add a new line
 11 print "\$hello = \"$hello\"\n";
 12 print qq/\$hello: "$hello"\n/;
```

```
$ e0203b.plx
$hello = 'Hello World'
$hello = 'Hello World'
$hello = "Hello World"
$hello: "He10
llo World"
```

Lines 6 and 11 demonstrate how quotation marks are escaped using a backslash. However, using the q and qq operators on Lines 8 and 12 simplifies the print statement. Note that special characters like $ can be displayed by adding a \ before them. Doing this masks their special meaning.

Scalar Operators

This section describes scalar operators and functions for both numbers and strings.

Arithmetic Operators and Functions

Table 2-2 shows the arithmetic operators.

Table 2-2 Arithmetic Operators

Operator	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Exponentiation
%	Modulus

Perl provides all the basic arithmetic operators. The only operator that might be unfamiliar to you is the modulus operator, %. The modulus operator returns the remainder from integer division.

```
$a = 27;
$b = $a % 6;      # $b becomes 3 since int (27 / 6) == 4
                  # with a remainder of 3
```

Assignment Operators

Table 2-3 shows the assignment operators.

Table 2-3 Assignment Operators

Operator	Example	Equivalent to
=	`$a = 4;`	`$a = 4;`
+=	`$a += 4;`	`$a = $a + 4;`
-=	`$a -= 4;`	`$a = $a - 4;`
*=	`$a *= 4;`	`$a = $a * 4;`
/=	`$a /= 4;`	`$a = $a / 4;`
**=	`$a **= 4;`	`$a = $a ** 4;`
%=	`$a %= 4;`	`$a = $a % 4;`

You have already used one assignment operator frequently in previous examples. The = operator assigns a value to a variable. In addition, a number of common assignment operations perform standard arithmetic operations. For example:

```
e0204.plx
  1 #!/usr/bin/perl -w
  2 $a = 10;
  3 $a = $a + 2; # $a now equals 12
  4 $a += 2; # $a now equals 14
  5 $a = $a - 4;  # $a now equals 10
  6 $a -= 4; # $a now equals 6
  7 print "$a\n";

$ e0204.plx
6
```

Autoincrement and Autodecrement Operators

Table 2-4 shows the autoincrement and autodecrement operators.

Table 2-4 Autoincrement and Autodecrement Operators

Example	Value Returned
`$a++;`	`$a` before it is incremented
`++$a;`	`$a` after it is incremented
`$a--;`	`$a` before it is decremented
`--$a;`	`$a` after it is decremented

Like the assignment operators, the autoincrement and autodecrement operators simplify incrementing and decrementing variables. They are equivalent to the following statements.

```
$a = $a + 1; # equivalent to $a++;
$a = $a - 1; # equivalent to $a--;
```

The autoincrement operator (`++`) counts a variable up by 1. The autodecrement operator (`--`) counts it down by 1.

```
e0204b.plx
   1 #!/usr/bin/perl
   2 $a = 10 ;
   3 $a += 3 ;  # $a now equals 13
   4 $a -= 7 ;  # $a now equals  6
   5 $a **= 2 ; # $a now equals 36
   6 print "$a\n";

$ e0204b.plx
36
```

However, when these operators are used in an expression, the placement of the operator determines the value returned. The autoincrement and autodecrement operators are used in the following example.

```
e0205.plx
  1 #!/usr/bin/perl -w
  2 # Using counters
  3 $a = 5 ;
  4
  5 $b = $a++ ;    # $b equals 5, $a equals 6
  6 print "a = $a  b = $b\n";
  7
  8 $b = ++$a ;    # $a equals 7, $b equals 7
  9 print "a = $a  b = $b\n";
 10
 11 $b = $a++ * 3 ;# $b equals 7*3=21, $a equals 8
 12 print "a = $a  b = $b\n";
```

```
$ e0205.plx
a = 6  b = 5
a = 7  b = 7
a = 8  b = 21
```

Notice that in Line 5, the value for `$a` is returned before the operation takes place. This differs from Line 8, in which `$a` is incremented before it is assigned to `$b`. In Line 11, even though `$a++` comes first in the expression, `$a` is incremented after the value of `$a * 3` is assigned to `$b`.

Arithmetic Functions

In addition to the arithmetic operators described in “Simple Operators” on page 1-10, there are a number of built-in arithmetic functions: `sin`, `cos`, `atan2`, `sqrt`, `exp`, `log`, `abs`, `int`, `rand`, and `srand`.

```
$a = 64;
$b = sqrt($a); # square root of 64 is 8
```

For further information, type “`perldoc perlfunc`” on the command line.

Bitwise Operators

Numbers can be processed by bit operators (`&` , `|` , `^` , `>>` , `<<`). Because they are so specialized, bitwise operators are not described here. Table 2-9 on page 2-29 lists the Perl operators.

Boolean Expressions

Boolean expressions return a true or false value. For example `(5 > 4)` returns true, and `(5 < 4)` returns false. Perl can judge any expression to be true or false.

But what do true and false really mean? The truth of a statement is determined by the following rules:

- An expression is considered false when it evaluates to the number 0, the string “0”, the empty string `""` or `"undef"`
- An expression that evaluates to any other value is considered true

Comparison Operators

Table 2-5 describes the comparison operators.

Table 2-5 Comparison Operators

Numeric Operator	String Operator	Meaning
>	gt	Greater than
<	lt	Less than
>=	ge	Greater than or equal to
<=	le	Less than or equal to
==	eq	Equal to
!=	ne	Not equal to
<=>	cmp	Boolean equality operators

Comparison operators compare two values. There are two sets of comparison operators, one for numbers and one for strings. Because data is stored as a string, the operator tells Perl whether two numbers or two strings are being compared. As an example, is 34567 greater than the number 5? In a numeric context, yes. In a string context, no.

```
e0206a.plx
  1 #!/usr/bin/perl -w
  2
  3 $a = (7 < 50);      # true, numeric comparison
  4 $b = ("7" < "50");  # true, still a numeric comparison
  5 print "\$a = $a \$b = $b\n";
  6
  7 $c = ("berlin" lt "bonn"); # true, string comparison
  8 $d = ("7" lt "50");        # false, string comparison
  9 print "\$c = $c \$d = $d\n";

$ e0206a.plx
$a = 1 $b = 1
$c = 1 $d =
```

When the script executes, the comparisons that are true return a 1. The false comparisons return a 0 for numeric comparisons and "" for string comparisons. Notice in Lines 3 and 4 that it makes no difference how the variables are defined. The operator determines whether a numeric or string comparison is performed. On line 5, the string comparison results in a false value of "".

The Boolean equality operators return a 1, 0, or -1 depending on the values compared. The following example demonstrates this.

```
e0206b.plx
  1 #!/usr/bin/perl -w
  2
  3 # Numeric
  4 $a = (3 <=> 2); # Left side greater
  5 $b = (3 <=> 3); # Equal
  6 $c = (3 <=> 4); # Right side greater
  7 print "\$a = $a \$b = $b \$c = $c\n";
  8
  9 # Strings
 10 $a = ("c" cmp "b"); # Left side greater
 11 $b = ("c" cmp "c"); # Equal
 12 $c = ("c" cmp "d"); # Right side greater
 13 print "\$a = $a \$b = $b \$c = $c\n";

$ e0206b.plx
$a = 1 $b = 0 $c = -1
$a = 1 $b = 0 $c = -1
```

The <=> operator performs numeric comparisons and the cmp operator compares strings. If the left side of the comparison is greater, a 1 is returned. If the left and right are equal, a 0 is returned. If the right side is greater, a -1 is returned.

Logical Operators

Table 2-6 describes the logical operators.

Table 2-6 Logical Operators

Symbol	Word	Meaning
&&	and	Logical and
||	or	Logical or
!	not	Logical not
^	xor	Exclusive or

Logical operators perform standard boolean `and`, `or`, and `not` functions. When there are two representations for a logical operator (for example, `or` and `||`), the only difference between the two is their precedence.

The `not` (or `!`) operator negates an expression, as shown below.

```
e0207a.plx
  1 #!/usr/bin/perl -w
  2
  3 # Logical Not
  4 print "Not 1 = ", !(1), "\n";
  5 print "Not 0 = ", !(0), "\n";
  6 print "Not snoopy =", !("snoopy"), "\n";
  7 print "Not \"\" = ", !(""), "\n";

$ e0207a.plx
Not 1 =
Not 0 = 1
Not snoopy =
Not "" = 1
```

On Line 4, the negation of the true value `1`, returns `""`. On Line 5, the negation of a false value `0` returns a `1`. On Line 6, the negation of a true value `"snoopy"` (remember anything not `0`, `"0,"` or `""` is true) returns a false value of `""`. Finally on Line 7, the negation of a false value of `""` returns a true value of `1`.

Short-Circuit Testing

When Perl evaluates an expression that uses a logical operator, the left side of an expression is checked first. If the left side of the expression determines the truth for the entire expression, then the right side of the expression is never evaluated. Thus, the right side is skipped or
"short-circuited." The expression returns the value from the side of the expression that is evaluated last.

The following script provides some short-circuit operator examples. Notice that the value returned is determined by which side of the expression is evaluated last.

```
e0207b.plx
  1 #!/usr/bin/perl -w
  2
  3 # Logical and
  4 $x = ( 4 - 4  && "snoopy"); # Left false, right skipped
  5 print "\$x = $x\n"; # $x = 0
  6
  7 $x = ( 4 + 2 && "snoopy"); # Left true, right evaluated
  8 print "\$x = $x\n"; # $x = "snoopy"
  9
 10 $x = ( 4 + 2 && 5 - 2); # Left true, right evaluated
 11 print "\$x = $x\n"; # $x = 3
 12
 13 # Logical or
 14 $x = ( 4 + 2 || 5 - 2); # Left true, right skipped
 15 print "\$x = $x\n"; # $x = 6
 16
 17 $x = ( 4 - 4 || "snoopy"); # Left false, right evaluated
18 print "\$x = $x\n"; # $x = snoopy

$ e0207b.plx
$x = 0
$x = snoopy
$x = 3
$x = 6
$x = snoopy
```

Lines 4 and 14 demonstrate short-circuiting. On Line 4, the left side of the statement is false, thus the `and` expression is false. The right side is skipped, and 0 is returned. On Line 14, `4+2` evaluates to `6`, which makes the `or` expression true and `6` is returned. The other examples show that the value from the last side evaluated (the right side) is returned.

Undef-or Testing

Perl 5.10 made it easier to assign a value to a variable only if it didn't have one. If it had been assigned a value of '', which could evaluate to false, the variabe would not be assigned a new value. The Undef-or operator is //. It is intentionally like the logical or, which it extends.

The following script shows the use of the new test. Note that the "use 5.010" pragma can be introduced later in the script.

e0207c.plx

```
# Logical and
$x = ( 4 - 4 && "snoopy"); # Left false, right skipped
print "\$x = $x\n"; # $x = 0

$x = ( 4 + 2 && "snoopy"); # Left true, right evaluated
print "\$x = $x\n"; # $x = "snoopy"

$x = ( 4 + 2 && 5 - 2); # Left true, right evaluated
print "\$x = $x\n"; # $x = 3

# Logical or
$x = ( 4 + 2 || 5 - 2); # Left true, right skipped
print "\$x = $x\n"; # $x = 6

$x = ( 4 - 4 || "snoopy"); # Left false, right evaluated
print "\$x = $x\n"; # $x = snoopy

use 5.010;
$x = '';
$x = $x // 6;
say "\$x is $x";  #$x is still ''
$x = $x || 6;
say "\$x is $x";  #$x is 6, the '' was clobbered
$x = undef;
$x = $x // 6;
say "\$x is $x";  #$x is 6
$x = (4-4) // "snoopy";  #false is not undef
say "\$x is $x";  #$x is 0
```

```
$ e0207c.plx
$x = 0
$x = snoopy
$x = 3
$x = 6
$x = snoopy
$x is
$x is 6
$x is 6
$x is 0
```

String Operators

Table 2-7 shows the string operators.

Table 2-7 String Operators

Function	Meaning
.	Concatenates two strings and returns the resulting string; the original strings remains unchanged
x	Repeats the string several times and returns the resulting string; the original string remains unchanged
.=	String assignment operators for .
x=	String assignment operators for x

Just like numbers, strings also have a basic set of operators. The concatenation operator is probably the most commonly used. In many computer languages, concatenation uses the same operator for concatenation as is used for numeric addition (+). In Perl, a period (.) is used to concatenate two strings.

Here are some examples.

```
e0208.plx
  1 #!/usr/bin/perl -w
  2 $a = "hello";
  3 $b = "world";
  4
  5 # Concatenate
  6 $c = $a . $b;     # $c equals "helloworld"
  7 print "\$c = " . "$c\n";
  8
  9 # Append world to $a using .=
 10 $a .= " world";    # $a equals "hello world"
 11 print "\$a = " . "$a\n";

$ e0208.plx
$c = helloworld
$a = hello world
```

Notice in Lines 6–7 that variables or string literals can be concatenated. In Line 10, a literal is appended to the end of an existing string.

The other string operator is the repetition operator. This operator allows strings to be repeated.

```
e0209.plx
  1 #!/usr/bin/perl -w
  2 $a = "hello";
  3 $a x= 2;   # $a equals "hellohello"
  4 print "$a\n";
  5
  6 # Practical use
  7 $width = 80;
  8 $line = "-" x $width;
  9 print "$line\n";

$ e0209.plx
hellohello
------------------------------------------------------------
```

Line 3 shows the default use of the repetition operator. The operator at first glance does not seem very useful, but look at Lines 7–9. The operator makes tasks, such as drawing a line across the screen, very easy.

An Example

The following script converts a distance in miles to kilometers. This is a pretty straightforward problem. For each mile, there are 1.6 kilometers. So the formula to determine kilometers is:

kilometers = miles * 1.6

Here is the script.

e0210a.plx

```
1 #!/usr/bin/perl -w
2 $miles = 3;
3 $kilometers = $miles * 1.6;
4 print "$miles miles is $kilometers km.\n";
```

```
$ e0210a.plx
3 miles is 4.8 km.
```

The script sets the number of miles, calculates the kilometers, and then prints the results using variable interpolation. However, as it is written, the source code needs to be modified every time the script is run. This makes the script less useful.

Reading <STDIN> With the chop and chomp Functions

To make our above example more useful, the script must get input from the keyboard. In Perl, keyboard input values are retrieved using the STDIN filehandle. When used in a scalar context, <STDIN> reads one line of input and stores the result in a variable. That is, the input operator <..> reads from the filehandle STDIN. For example, the following script reads in a line and prints out the line it read.

e0211.plx
```
1 #!/usr/bin/perl -w
2 # Read a value using <STDIN>
3 print "What is your favorite color? ";
4 $in = <STDIN>;
5 print "Your favorite color is: $in\n";
6
```

```
$ e0211.plx
What is your favorite color? blue
Your favorite color is: blue

$
```

The statement in Line 4 halts the script's execution and waits for a line of input. When Return is pressed, the program stores the value entered into $in and continues. Line 5 prints the result.

Notice that the sample output contains an extra blank line. Why? Well, when <STDIN> reads a line, a newline character is used to mark the end of input. This character is added when Return is pressed. After "blue" is typed, a \n is added to $in. Therefore, the print command in Line 5 prints two \n characters: the one in the print statement and one contained in $in. To remove \n from user's input, use the chomp function.

The chop and chomp Functions

The syntax for the chop and chomp functions is shown in Table 2-8.

Table 2-8 chop and chomp String Functions

Function	Meaning
chop(string)	Cuts off the last character of a string, whatever it is, and returns that character.
chomp(string)	Cuts off the last character of a string only if it is defined as the newline character. Chomping a list chomps each element. It returns the number of removed characters.

You can use these two functions to remove a newline character from a string. The chop function removes the last character no matter what it is. The chomp function removes the last character only if it is a newline. To demonstrate, the previous script has been rewritten to call chop and chomp twice.

e0212.plx

```
 1 #!/usr/bin/perl -w
 2 # Read a value using <STDIN>
 3 print "What is your favorite color? ";
 4 $in = <STDIN>;
 5 $a = $in;
 6 $b = $in;
 7
 8 # chop $a twice and print
 9 chop($a);
10 chop($a);
11 print "Your favorite color is: $a\n";
12
13 # chomp $b twice and print
14 chomp($b);
15 chomp($b);
16 print "Your favorite color is: $b\n";
```

```
$ e0212.plx
What is your favorite color? blue
Your favorite color is: blu
Your favorite color is: blue
```

Look at the output. Notice that when `chop` is used, the last character is removed no matter what it is. If, for some reason, there is no newline character at the end of a string, `chop` starts deleting characters from the string.

The `chomp` function is much safer. Because there are no newline characters the second time the function is called, no change is made to the string. The `chomp` function is the preferred method for removing newline characters from input.

Notice that the example uses two lines to remove the newline character from the string.

```
$in = <STDIN>;
chomp($in);
```

There is a shorthand method to perform the same function in one line. The previous lines can be replaced with:

```
chomp($in = <STDIN>);
```

Miles-to-Kilometer Example Revisited

The following is an updated version of the miles-to-kilometer converter. One line is added to the script to read the input value from the keyboard. With the change and some extra `print` statement formatting, the script looks like this.

```
e0210b.plx
  1 #!/usr/bin/perl -w
  2 print "\nPlease enter the number of miles: ";
  3 chomp($miles=<STDIN>);
  4 $kilometers = $miles * 1.6;
  5 print "$miles miles is $kilometers km.\n\n";

$ e0210b.plx

Please enter the number of miles: 34
34 miles is 54.4 km.
```

This is a much more useful script.

Precedence and Associativity

With complex expressions, rules determine the order or precedence of operations. Table 2-9 on page 2-29 lists operators in descending order of their precedence.

For example in `"2 + 3 ** 2 * 4"`, three operations are executed. The `**` operator has the highest precedence and it evaluates first. Next, `*` is executed, and then `+`.

This means `2 + ((3**2) * 4)` equals 38.

The precedence list is very detailed and long. Why? Because, all precedence rules are inherited from the C programming language. Additional Perl operators have their own precedence rules and must be added to the existing list.

Note – It is possible to explicitly determine the order of operations using parentheses.

Operators are executed in the order of their precedence, but what about expressions? In expressions with more than one identical operator, which operator is executed first (for example, `6/4/3` or `2**3**2`)? The rules used to determine which operator is executed first are called associativity rules.

Since the `**` operator is right-associative, in `2**3**4`, the right operator is applied first. Thus, `3**4` is calculated first. Written another way, the expression is calculated like this: `2**(3**4)`.

Table 2-9 Precedence and Associativity

Operators in Descending Precedence	Associativity	Explanation
`()`	Left	Parentheses
Terms and list operators (leftward)	Left	"hello", print
`->`	Left	Infix dereference operator (IDO)
`++ --`	Nonassociative	Autoincrement, autodecrement
`**`	Right	Exponentiation
`! ~ \ + -`	Right	Logical not, bitwise not, referenciation, unary +, unary -
`=~ !~`	Left	Match, does not match
`* / % x`	Left	Multiplication, division, modulus, replicate
`+ - .`	Left	Addition, subtraction, concatenation
`<< >>`	Left	Bitwise shift
Named unary operators	Nonassociative	`sin`, `chr`, and so on
`< > <= >= lt gt le ge`	Nonassociative	Comparison operators
`== != <=> eq ne cmp`	Nonassociative	Comparison operators
`&`	Left	Bitwise and
`| ^`	Left	Bitwise or and xor
`&&`	Left	Logical and
`||`	Left	Logical or
`..  ...`	Nonassociative	Range operator (inclusive/noninclusive)
`?:`	Right	Ternary operator: if ... then ... else
`=, +=, -=, *=,` and so on	Right	Assignment and binary assignments
`, =>`	Left	Comma, “Fat” Comma
List operators (rightward)	Nonassociative	Print `$a`, `$b`
`not`	Right	Logical not
`and`	Left	Logical and
`or xor`	Left	Logical or and xor

Exercise: Create Scripts Using Scalars

In this exercise, you complete the following tasks:

- Define and describe numeric and string scalar data
- Create string and numeric scalar variables
- Modify scalar variables using operators
- Print scalar values using single and double quotation marks
- Remove `\n` from user input using the `chomp` function
- Calculate a value using scalar operators based on user input

Preparation

To prepare for this exercise, obtain a text editor you are comfortable working with.

Tasks

Complete the following steps:

1. Create a script that performs the operations that follow on the variables `$a`, `$b`, `$c`, `$d`, and `$e`.

   - Initialize `$a` to `2` and `$b` to `3`.
   - Add `$a` and `$b`, and print the result.
   - Multiply `$a` and `$b`, and print the result.
   - Multiply `$a` by `$b`, and assign the result to `$c`.
   - Add 3 to `$c` using `+`, and print the result.
   - Add 3 to `$c` using `+=`, and print the result.
   - Assign `$c` to `$d`, and increment `$c` in the same statement. Print `$c` and `$d`.
   - Increment `$c`, and then assign it to `$e`. Print `$c` and `$e`.

   Your output might look like this:

```
a + b = 5
a x b = 6
c = a * b
c = 6
c = c + 3
c = 9
c += 3
c = 12
d = c++
d = 12 and c= 13
e = ++c
e = 14 and c= 14
```

2. Create a script that performs the operations described below on the variables `$a` and `$b`.

   - Set `$a` equal to `"Hello "` and `$b` equal to `"World!"`.
   - Add `$a` to `$b` using `+`, and print the result.
   - Contatenate `$a` and `$b`, and assign the result to `$d`. Print `$d`.
   - Add `\n` to `$d`.
   - Print `$d` four times using the `x` operator.
   - Use the `chop` function to remove `\n`.
   - Print `$d` four times using the `x` operator.

   Your output might look like the following:

```
Add $a + $b and you get: 0
Concatenate $a and $b and you get: Hello World!
Add \n to $d
Print $d four times using the x operator:
Hello World!
Hello World!
Hello World!
Hello World!
Remove \n with chop and print $d 4 times:
Hello World!Hello World!Hello World!Hello World!$
```

3. Modify the script you created in Step 1. Allow a user to input `$a` and `$b` using `<STDIN>`.

4. Modify the script you created in Step 2. Allow a user to input `$a` and `$b` using `<STDIN>`.

5. Write a script that gets two numbers from <STDIN>: $a and $b. Have the script multiply the two numbers and print the result.

   The output might look like this:

   ```
   Please enter two values:
   The value for a is 2
   The value for b is 3
   2 x 3 = 6
   ```

   After you have your script working, try multiplying the following values:

   - 3 and 4
   - 4.1 and 2.3
   - b and a
   - 4b and 3a
   - b4 and a3

6. Using the knowledge you have acquired so far, create a script that converts a temperature in Celsius into a temperature in Fahrenheit. The script should produce the following output:

   ```
   Please enter the temperature in Celsius: 25
   25 degrees Celsius is 77 degrees Fahrenheit
   ```

   The conversion formula is:

   Fahrenheit = (9/5) * Celsius + 32

7. Using the knowledge you have acquired so far, create a script that converts a temperature in Fahrenheit into a temperature in Celsius. The script output should look something like this:

   ```
   Please enter the temperature in Fahrenheit: 77
   77 degrees Fahrenheit is 25 degrees Celsius
   ```

   The conversion formula is:

   Celsius = (Fahrenheit - 32) * (5/9)

String Functions

A number of functions exist for manipulating strings. The sections that follow review these functions.

String Search Functions

Table 2-10 describes the string search functions.

Table 2-10 String Search Functions

Function	Meaning
`index (string, target[, start])`	Returns the position where a `target` string is found in `string`. By default, the search starts at Position 0. If the starting position is specified by `start`, the search starts there. If `target` is not found, -1 is returned.
`rindex (string, target [, position])`	Like `index`, but searches from the right. If the rightmost position is not specified by `position`, the search starts from the end of the string, but the returned value and the argument position are still counted from the beginning of the string.

Search functions search strings for a character or substring. Once found, the functions return the location of the string found.

e0213.plx

```
  1 #!/usr/bin/perl -w
  2 $address = "Doe, Jane, Washington, DC";
  3
  4 # Search $address starting at 0 character
  5 $i = index($address,",");
  6 print "1st comma is at position $i\n"; # $i = 3
  7
  8 # Search $address from  first comma
  9 $i = index($address,",",$i+1);
 10 print "2nd comma is at position $i\n"; # $i = 9
 11
 12 # Search $address for a starting from right
 13 $i = rindex($address,",");
 14 print "Last comma is at position $i\n"; # $i = 21
```

```
$ e0213.plx
1st comma is at position 3
2nd comma is at position 9
Last comma is at position 21
```

Both functions return integer numbers indicating the position of the string if it is found.

String Extraction Functions

To extract a substring from another string, use the `substr` function. This function comes in two forms as shown in Table 2-11.

Table 2-11 Extraction String Functions

Function	Meaning
`substr (string,start[, length])`	rvalue: `$s = substr(...)` Returns the part of the string that begins at `start` and has a length of `length` characters. The original string remains unchanged. If `length` is omitted, the substring reaches to the end of the string.
`substr (string,start[, length])`	lvalue: `substr($s,..,...) = "A string";` Replaces what would have been returned by the function with the value on the right.

The following script shows the basics of using `substr`.

```
e0214a.plx
  1 #!/usr/bin/perl -w
  2 $address = "Doe,Jane,Washington,DC";
  3
  4 $lastname = substr($address, 0, 3); # Get last name
  5 $firstname = substr($address, 4, 4); # Get first name
  6 $state = substr($address, 20); # Get State
  7
  8 #print Results
  9 print "Name: $firstname $lastname\n";
 10 print "State: $state\n";

$ e0214a.plx
Name: Jane Doe
State: DC
```

Lines 4 and 5 show the standard way of using `substr`. You specify the string being searched, the starting position, and the length to extract.
Line 6 shows how to extract from a starting position to the end of the string.

Using the second form of `substr`, you can substitute one string for another.

e0214b.plx

```
1 #!/usr/bin/perl -w
2 $address = "Doe,Jane,Washington,DC";
3
4 substr($address, 0, 3) = "Smith"; # New last name
5 print "Address now: $address\n";
6
7 $replaced = substr($address, 6, 4, "John"); # New first
8 print "Address now: $address\n";
9 print "What was replaced? $replaced\n";
```

```
$ e0214b.plx
Address now: Smith,Jane,Washington,DC
Address now: Smith,John,Washington,DC
What was replaced? Jane
```

Line 4 shows the standard way to use `substr` to replace a selected substring. Line 7 shows an alternative way to do the same thing. The advantage of this method is that the value being replaced is returned by `substr`.

Often, the `substr` function is used with `index` and `rindex`. The following example demonstrates this by rewriting `e0214.plx` to use variables to locate substrings.

```
e0214c.plx
  1 #!/usr/bin/perl -w
  2 $address = "Doe,Jane,Washington,DC";
  3
  4 # Find the commas
  5 $firstcomma = index($address, ",");
  6 $secondcomma = index($address, ",", $firstcomma + 1);
  7 $lastcomma = rindex($address, ",");
  8
  9 $lastlength = $firstcomma - 0; # Calc length of last name
 10 $lastname = substr($address, 0, $lastlength); # Get last name
 11
 12 # Get First name
 13 $firstlength = ($secondcomma - 1) - $firstcomma; # Calc length
 14 $firstname = substr($address,$firstcomma + 1, $firstlength);
 15
 16 # Grab from last comma to end of string
 17 $state = substr($address,$lastcomma + 1);
 18
 19 #print Results
 20 print "Name: $firstname $lastname\n";
 21 print "State: $state\n";

$ e0214c.plx
Name: Jane Doe
State: DC
```

In this example, the locations of the commas are calculated first. Those values are then used to calculate the position and length of the strings to extract. The advantage of this method is that it will work even if the names, city, or state values change in size.

General String Functions

The general string functions shown in Table 2-12 take a string and return a characteristic of the string, or they return the string in a different format.

Table 2-12 General String Functions

Function	Meaning
`length $string`	Returns the number of characters in a string.
`reverse $string`	Returns a string in reversed order. The original string remains unchanged.
`lc $string`	Returns the string in lowercase. The original string remains unchanged.
`lcfirst $string`	Returns the string with the first character in lowercase. The original string remains unchanged.
`uc $string`	Returns the string in uppercase. The original string remains unchanged.
`ucfirst $string`	Returns the string with the first character in uppercase. The original string remains unchanged.
`ord $char`	Returns the numeric ASCII value of the character.
`chr $number`	Returns the character represented by the ASCII value.

The following script demonstrates each of the general string functions.

e0215.plx

```
 1 #!/usr/bin/perl -w
 2 $address = "Doe,Jane,Washington,DC";
 3
 4 # Calculate length
 5 print "\$address's length is ", length($address), "\n";
 6
 7 $rev = reverse($address); # Reverse $address
 8 print "\$rev = $rev\n";
 9
10 $lower = lc($address); # Make lowercase
11 print "\$lower = $lower\n";
12
13 $upper = uc($address); # Make uppercase
14 print "\$upper = $upper\n";
15
16 $n = ord("A"); # ASCII of A
17 print "\$n = $n\n";# $n = 65
18
19 $c = chr(65); # ASCII 65 is A
20 print "\$c = $c\n"; # $c = A
```

```
$ e0215.plx
$address's length is 22
$rev = CD,notgnihsaW,enaJ,eoD
$lower = doe,jane,washington,dc
$upper = DOE,JANE,WASHINGTON,DC
$n = 65
$c = A
```

With functions, the parentheses around the arguments are not necessary because internally Perl handles functions as operators.

```
length($str) ; # With parentheses
length $str ; # Without parentheses. It works either way.
```

But bear in mind, the order of the operations can change if the parentheses are left off: `(3 + 4) * 2` does not equal `3 + 4 * 2`.

Exercise: Manipulate Strings Using String Functions

In this exercise, you complete the following tasks:

- Find a character in a string using `index` and `rindex`
- Extract a part of a string using `substr`

Tasks

Complete the following steps:

8. Given a string variable with the following value, perform the operations that follow.

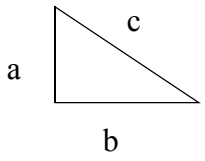
   ```
   $phone = "Washington,George,302-555-1212";
   ```

   Perform these operations:

   - Find the first comma, and print its location in the string. Use the `index` function.
   - Find the last dash, and print its location in the string. Use the `rindex` function.
   - Print out the name (George Washington). Use the `substr` function.

9. (Optional) Write a script that computes the hypotenuse of a right triangle. If you remember your geometry, the Pythagorean Theorem states that the square of the hypotenuse can be calculated by summing the squares of the other two sides of the triangle. Thus, for a given triangle with sides a and b, the hypotenuse c would be calculated as follows:

   $c^2 = a^2 + b^2$

   Write a script that obtains the values of a and b and computes c. The sample output that follows includes values you can use to test your script.

Sample output:

```
Enter a value for each side of the triangle:
Side A = 3
Side B = 4
If side A is 3 and B is 4, then the hypotenuse is 5
```

10. (Optional) Create a script that reads in a fully qualified domain name and breaks the name into its component parts. The output should be similar to the following:

```
$ domain.plx
Enter domain name: ou.oracle.com
Host: ou
Domain: oracle.com
Top Level domain: com
```

Exercise Summary

Discussion – Take a few minutes to discuss the experiences, issues, or discoveries you had during the lab exercises.

- Experiences
- Interpretations
- Conclusions
- Applications

Exercise Solutions

1. Create a script that performs the operations that follow on the variables `$a`, `$b`, `$c`, `$d`, and `$e`.

   - Initialize `$a` to `2` and `$b` to `3`.
   - Add `$a` and `$b`, and print the result.
   - Multiply `$a` and `$b`, and print the result.
   - Multiply `$a` by `$b`, and assign the result to `$c`.
   - Add 3 to `$c` using `+`, and print the result.
   - Add 3 to `$c` using `+=`, and print the result.
   - Assign `$c` to `$d`, and increment `$c` in the same statement. Print `$c` and `$d`.
   - Increment `$c`, and then assign it to `$e`. Print `$c` and `$e`.

   Your output might look like this:

```
a + b = 5
a x b = 6
c = a * b
c = 6
c = c + 3
c = 9
c += 3
c = 12
d = c++
d = 12 and c= 13
e = ++c
e = 14 and c= 14
```

Suggested solution:

lab0201.plx

```
 1 #!/usr/bin/perl -w
 2 $a = 2; $b = 3;
 3
 4 print "a + b = ", $a + $b, "\n";
 5 print "a x b = ", $a * $b, "\n";
 6
 7 # Assign C a value
 8 $c=$a * $b;
 9 print "c = a * b\n";
10 print "c = $c\n";
11
12 # Add three with + operator
13 $c = $c + 3;
14 print "c = c + 3\n";
15 print "c = $c\n";
16
17 # Add 3 using the binary assignment operator
18 $c += 3;
19 print "c += 3\n";
20 print "c = $c\n";
21
22 # Add 1 to $c after assigning $c's value to $d
23 $d = $c++;
24 print "d = c ++\n";
25 print "d = $d and c= $c\n";
26
27 # Increment $c, then assign the value to $e
28 $e = ++$c;
29 print "e = ++c\n";
30 print "e = $e and c= $c\n";
```

2. Create a script that performs the operations described below on the variables $a and $b.

   - Set $a equal to "Hello " and $b equal to "World!".
   - Add $a to $b using +, and print the result.
   - Contatenate $a and $b, and assign the result to $d. Print $d.
   - Add \n to $d.
   - Print $d four times using the x operator.
   - Use the chop function to remove \n.
   - Print $d four times using the x operator.

   Your output might look like the following:

```
Add $a + $b and you get: 0
Concatenate $a and $b and you get: Hello World!
Add \n to $d
Print $d four times using the x operator:
Hello World!
Hello World!
Hello World!
Hello World!
Remove \n with chop and print $d 4 times:
Hello World!Hello World!Hello World!Hello World!$
```

   Suggested solution:

lab0202.plx

```
 1 #!/usr/bin/perl -w
 2 $a = "Hello "; $b = "World!";
 3
 4 $c = $a + $b;
 5 print "Add \$a + \$b and you get: $c\n";
 6
 7 $d = $a.$b;
 8 print "Concatenate \$a and \$b and you get: $d\n";
 9
10 $d = $d."\n";
11 print "Add \\n to \$d\n";
12 print "Print \$d four times using the x operator:\n";
13 print $d x 4;
14
15 chop($d);
16 print "Remove \\n with chop and print \$d 4 times:\n";
17 print $d x 4;
```

3. Modify the script you created in Step 1. Allow a user to input $a and $b using <STDIN>.

   Suggested solution:

lab0203.plx

```
 1 #!/usr/bin/perl -w
 2 print "Please enter a value for a and press enter: ";
 3 chomp($a = <STDIN>);
 4
 5 print "Please enter a value for b and press enter: ";
 6 chomp($b = <STDIN>);
 7
 8 print "a + b = ", $a + $b, "\n";
 9 print "a * b = ", $a * $b, "\n";
10
11 # Assign C a value
12 $c=$a * $b;
13 print "If c = a * b, c = ", $c, "\n";
14
15 # Add three with + operator
16 $c = $c + 3;
17 print "c = c + 3, c now = ", $c, "\n";
18
19 # Add 3 using the binary assignment operator
20 $c += 3;
21 print "c += 3, c now = ", $c, "\n";
22
23 # Add 1 to $c after assigning $c's value to $d
24 $d = $c++;
25 print "d = c ++ d = ", $d," and c= ", $c, "\n";
26
27 # Increment $c, then assign the value to $e
28 $e = ++$c;
29 print "e = ++c e = ", $e, " and c= ", $c, "\n";
```

4. Modify the script you created in Step 2. Allow a user to input $a and $b using <STDIN>.

   Suggested solution:

lab0204.plx

```
 1 #!/usr/bin/perl -w
 2
 3 print "Please enter a value for \$a and press enter: ";
 4 chomp($a = <STDIN>);
 5
 6 print "Please enter a value for \$b and press enter: ";
 7 chomp($b = <STDIN>);
 8
 9 $c = $a + $b;
10 print "Add \$a + \$b and you get: $c\n";
11
12 $d = $a . $b;
13 print "Concatenate \$a and \$b and you get: $d\n";
14
15 $d = $d . "\n";
16 print "Add \\n to \$d and use the x operator to print \$d 4
times:\n";
17 print $d x 4;
18
19 chop($d);
20 print "Remove \\n with chop and print 4 times you get:\n";
21 print $d x 4;
```

5. Write a script that gets two numbers from <STDIN>, $a and $b. Have the script multiply the two numbers and print the result.

   The output might look like this:

```
Let's multiply two values. Please enter two values:
The value for a is 2
The value for b is 3
2 x 3 = 6
```

   After you have your script working, try multiplying the following values:

   - 3 and 4
   - 4.1 and 2.3
   - b and a
   - 4b and 3a
   - b4 and a3

   Suggested solution:

lab0205.plx

```
 1 #!/usr/bin/perl -w
 2 # Dynamic Type Conversion
 3
 4 print "Please enter two values:\n" ;
 5
 6 # read in two values, one per line
 7 print "The value for a = ";
 8 chomp($a = <STDIN>);
 9
10 print "The value for b = ";
11 chomp($b = <STDIN>);
12
13 # Multiplication
14 $p = $a * $b;
15
16 # output with variable interpolation and newline
17 print "$a x $b = $p\n";
```

6. Using the knowledge you have acquired so far, create a script that converts a temperature in Celsius into a temperature in Fahrenheit. The script output should look like the following:

```
Please enter the temperature in Celsius: 25
25 degrees Celsius is 77 degrees Fahrenheit
```

The conversion formula is:

Fahrenheit = (9/5) * Celsius + 32

Suggested Solution:

lab0206.plx

```
1 #!/usr/bin/perl -w
2 print "\n";
3
4 print "Please enter the temperature in Celsius: ";
5 chomp($celsius = <STDIN>);
6
7 $fahr = (9/5) * $celsius + 32;
8
9 print "$celsius degrees Celsius is $fahr degrees Fahrenheit\n\n";
```

7. Using the knowledge you have acquired so far, create a script that converts a temperature in Fahrenheit into a temperature in Celsius. The script output should produce the following output:

```
Please enter the temperature in Fahrenheit: 77
77 degrees Fahrenheit is 25 degrees Celsius
```

The conversion formula is:

Celsius = (Fahrenheit - 32) * (5/9)

Suggested solution:

lab0207.plx

```
1 #!/usr/bin/perl -w
2 print "\n";
3
4 print "Please enter the temperature in Fahrenheit: ";
5 chomp($fahr = <STDIN>);
6
7 $celsius = ($fahr - 32) * (5/9);
8
9 print "$fahr degrees Fahrenheit is $celsius degrees Celsius\n\n";
```

8. Given a string variable with the following value, perform the operations that follow.

```
$phone = "Washington,George,302-555-1212";
```

Perform these operations:

- Find the first comma, and print its location in the string. Use the `index` function.
- Find the last dash, and print its location in the string. Use the `rindex` function.
- Print out the name (George Washington). Use the `substr` function.

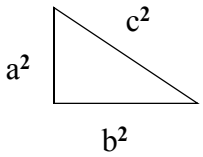
Suggested solution:

lab0208.plx

```
 1 #!/usr/bin/perl -w
 2
 3 $phone = "Washington,George,302-555-1212";
 4
 5 # Find first comma
 6 $i = index($phone,",");
 7 print "1st comma is at position $i\n"; # $i = 10
 8
 9 # Find first dash from right
10 $i = rindex($phone,"-");
11 print "Last dash is at position $i\n"; # $i = 25
12
13 # Print name, first last
14 $first = substr($phone,11,6);
15 $last = substr($phone,0,10);
16 print "Name: $first $last\n";
```

9. (Optional) Write a script that computes the hypotenuse of a right triangle. If you remember your geometry, the Pythagorean Theorem states that the square of the hypotenuse can be calculated by summing the squares of the other two sides of the triangle. Thus, for a given triangle with sides a and b, the hypotenuse c would be calculated as follows:

$c^2 = a^2 + b^2$



Write a script that obtains the values of a and b and computes c. The sample output that follows includes values you can use to test your script.

Sample Output:

```
Enter a value for each side of the triangle:
Side A = 3
Side B = 4
If side A is 3 and B is 4, then the hypotenuse is 5
```

Suggested solution:

lab0209.plx

```
 1 #!/usr/bin/perl -w
 2 # Pythagorean Theorem
 3 print "Enter a value for each side of the triangle:\n";
 4
 5 # read in two values, one per line
 6 print "Side A = ";
 7 chomp($a = <STDIN>);
 8
 9 print "Side B = ";
10 chomp($b = <STDIN>);
11
12 # Pythagorean Theorem
13 $c = sqrt ($a ** 2 + $b ** 2);
14
15 # output with variable interpolation and newline
16 print "If side A is $a and B is $b, then the hypotenuse is $c\n";
```

10. (Optional) Create a script that reads in a fully qualified domain name and breaks the name into its component parts. The script output should be similar to the following:

```
$ domain.plx
Enter domain name: ou.oracle.com
Host: ou
Domain: oracle.com
Top Level domain: com
```

Suggested solution:

lab0210.plx

```
 1 #!/usr/bin/perl -w
 2 # Domain Splitter
 3 print "Enter domain name: ";
 4 chomp( $name = <STDIN> );
 5
 6 # Find dots
 7 $first = index($name, ".");
 8 $second = rindex($name, ".");
 9
10 # Split apart
11 $host = substr($name, 0, $first);
12 $domain = substr($name, ($first + 1));
13 $top = substr($name, ($second + 1));
14
15 # Print Results
16 print "Host: $host\n";
17 print "Domain: $domain\n";
18 print "Top Level domain: $top\n";
```

Control Structures

Objectives

Upon completion of this module, you should be able to:

- Print a message using an `if` statement
- Print a message using an `if/else` statement
- Print a message using a compound `if/elsif/else` statement
- Use a `for` loop to display a list of numbers
- Use a `foreach` loop to perform calculations on a list of numbers
- Use `while` loops to repeatedly perform logical tests using an `if/elsif/else` statement
- Print formatted text with a Here document
- Exit from a loop using loop controls and statement modifiers
- Use a switch construct in Perl to print a message

Relevance

Discussion – The following questions are relevant to understanding how control structures are used:

- How are decisions made in a program?
- What is iteration, and why is it needed in a program?

Introducing Control Structures

This module describes the control structures used to regulate the flow of a program. Conditional structures from `if` statements to loops are described. In addition, the logical and comparison operators introduced in Module 2, “Scalars,” are put into practice in this module.

Truth?

Before starting this module, the rules for determining if an expression is true or false must be reviewed one more time. The following concept is used repeatedly in the examples and exercises that follow:

- An expression is considered to be false when it evaluates to the number 0, the empty string `""` (equivalent to `undef`), or the string `"0"`.
- An expression that evaluates to any other value is considered true.

The if Statements

The if statement is the basic construct used to make conditional decisions. The following is a simple if statement:

```
if (condition) {statement block}
```

If the condition contained in the () is true, then the statements contained in the statement block are executed. If the condition is false, then the statements are skipped.

A statement block is a series of statements contained in a { } pair. For example, the following script includes a simple if statement.

```
e0301.plx
  1 #!/usr/bin/perl -w
  2 # Simple if statement
  3
  4 $color = "blue";
  5
  6 if ($color eq "blue"){
  7   print "The sky is blue\n";
  8 }
  9

$ e0301.plx
The sky is blue
```

Line 6 shows the if statement with a small code block. If the value of $color is blue, the print statement is executed.

The if/else Statement

The next form of the if statement is the if/else statement. With this statement, if a condition is false, instead of doing nothing, a statement block is executed. The statement takes the following form:

```
if (condition){
   statement block 1
}
else {
   statement block 2
}
```

This provides an either-or situation. One of the statement blocks is executed: the first block if the condition is true or the second block if the condition is false. The following script is an example of an `if/else` statement.

```
e0302.plx
  1 #!/usr/bin/perl -w
  2
  3 $color = "green";
  4
  5 if ($color eq "blue") {
  6   print "The sky is blue\n";
  7 }
  8 else {
  9   print "The color is $color\n";
 10 }

$ e0302.plx
The color is green
```

Because the color is green, the condition on Line 5 is false. The statement block following the `else` statement is executed, which prints the message.

The `if/elsif/else` Statement

The `if/elsif/else` construct creates compound `if` statements that can perform multiple tests. The statement is constructed as follows.

```
if (condition 1){
    statement block 1
}
elsif (condition 2){
    statement block 2}
elsif (condition 3){
    statement block 3}
else {
    final statement block
}
```

With this structure, multiple conditions can be tested (much like a case statement in other languages). The previous example is modified to test for multiple conditions.

e0303.plx

```
 1 #!/usr/bin/perl -w
 2
 3 $color = "green";
 4
 5 if ($color eq "blue") {
 6   print "The sky is blue\n";
 7 }
 8 elsif ($color eq "green") {
 9   print "The tree is green\n";
10 }
11 elsif ($color eq "brown") {
12   print "The cow is brown\n";
13 }
14 else {
15   print "The color is $color\n";
16 }
```

```
$ e0303.plx
The tree is green
```

Notice that each condition is tested separately. If the condition is true, then the block of code associated with that condition is executed and any remaining blocks are skipped. The `else` clause catches all conditions not tested.

Note – The keyword is `elsif`. A common mistake is to spell out the word "elseif," which produces a syntax error.

The unless Statement

The `unless` statement is the exact equivalent of an "`if (!condition)`" statement, and is used to improve code readability. The following example shows the structure of the `unless` statement.

```
unless (condition) {statement block}
```

The statement block is executed only if the statement is false. Below is an example.

```
e0304.plx
  1 #!/usr/bin/perl -w
  2 # Simple unless statement
  3
  4 $color = "brown";
  5
  6 unless ($color eq "blue") {
  7   print "The sky is not blue\n";
  8 }
  9

$ e0304.plx
The sky is not blue
```

If the color is something other than blue, then the statement block is executed. If it is blue, then the block is skipped.

An `else` clause can be added to an `unless` statement to essentially create a reversed `if/else` statement. Usually, an "`if/else`" would make more sense in these conditions, but Perl provides you with the option to choose the approach you want (this is yet another example of TMTOWTDI, "There's more than one way to do it.")

```
unless (condition) {statement block}
else               {statement block}
```

If the statement is true, the `else` statement block is executed. If the statement is false, the first statement block is executed. For example:

```
e0305.plx
  1 #!/usr/bin/perl -w
  2
  3 $color = "green";
  4
  5 unless ($color ne "blue") {
  6   print "The sky is blue\n";
  7 }
  8 else {
  9   print "The color is $color\n";
 10 }

$ e0305.plx
The color is green
```

Multiple Conditions

In addition to the simple logical tests shown so far, Perl offers a way to test multiple conditions using the operators introduced in the previous chapter:

- The logical operators and and or
- Their symbolic equivalents, && and ||

The || Operator (or)

The following script demonstrates the use of the || operator. If either condition is true, the first message is printed.

```
e0306.plx
  1 #!/usr/bin/perl -w
  2
  3 $color = "blue";
  4 $day = "Tuesday";
  5
  6 if ($color eq "blue" || $day eq "Monday" ) {
  7   print "The sky is blue or the day is Monday\n";
  8 }
  9 else {
 10   print "The color on $day is $color\n";
 11 }

$ e0306.plx
The sky is blue or the day is Monday
```

On Line 6, because one condition is true, the message is printed.

The && Operator (and)

With the && operator, both sides of an expression must be true for the expression to be true. The previous example is shown here using the && operator.

```
e0307.plx
  1 #!/usr/bin/perl -w
  2
  3 $color = "blue";
  4 $day = "Tuesday";
  5
  6 if ($color eq "blue" && $day eq "Monday" ) {
  7   print "The sky is blue and the day is Monday\n";
  8 }
  9 else {
 10   print "The color on $day is $color\n";
 11 }

$ e0307.plx
The color on Tuesday is blue
```

On Line 6, because both conditions are not true, the else clause is executed.

Note – The difference between &&/|| and and/or is that the former has a much higher precedence. The "hard" logicals (&& and ||) are used for conditional decision-making in tests, while the "soft" ones are used for control flow. In particular, using the soft logicals in assignment is unlikely to do what you expect.

Numeric Comparisons

In most of the examples used so far in this module, string operators were used to test conditions. But you can also make comparisons using the numeric operators. For example, you can test a number's relative location with an `if/elsif/else` statement.

e0308.plx

```
 1 #!/usr/bin/perl -w
 2
 3 print "Enter a number: ";
 4 chomp($num=<STDIN>);
 5
 6 if ($num == 0) {
 7   print "You entered a 0\n";
 8 }
 9
10 elsif ($num >= 10) {
11   print "$num is equal to or greater than 10\n";
12 }
13
14 elsif ($num < 10 && $num > 0) {
15   print "$num is between 1 and 9\n";
16 }
17
18 else {
19   print "$num is less than zero\n";
20 }
```

```
$ e0308.plx
Enter a number: 2
2 is between 1 and 9
```

Loops

While `if` and `unless` statements control decision making and program flow, loops control iteration in a Perl program. The section that follows reviews the kinds of loops available in Perl programs.

The `while` Loop

A `while` loop keeps executing a statement block as long as the specified condition is true; for example, "`while (1){ ... }`" is an infinite loop. The `while` loop takes the following form.

```
while (condition) {
    statement block
}
```

As an example, an improved version of the miles-to-kilometer conversion program follows.

```
e0309.plx
  1 #!/usr/bin/perl -w
  2 $miles = "";
  3
  4 while ($miles ne "0"){
  5
  6   print "Enter 0 to exit\n";
  7   print "Please enter the number of miles: ";
  8   chomp($miles=<STDIN>);
  9
 10   $kilometers = $miles * 1.6;
 11   print "$miles miles is $kilometers km.\n\n";
 12 }
```

```
$ e0309.plx
Enter 0 to exit
Please enter the number of miles: 10
10 miles is 16 km.

Enter 0 to exit
Please enter the number of miles: 0
0 miles is 0 km.
```

The `while` loop, Lines 4 and 12, surrounds the conversion script. The loop tests for a 0. If 0 is entered, the loop exits. Otherwise, the loop continues to prompt for input.

The `until` Loop

The `until` statement is the opposite of the while statement and is equivalent to "`while ( !condition )`". An `until` statement continues executing the block until the specified condition becomes true - so "`until (0){ ... }`" is an infinite loop.

```
until (condition){
    statement block
}
```

To demonstrate the difference, the previous miles-to-kilometer conversion script is rewritten using an `until` statement.

e0310.plx

```
 1 #!/usr/bin/perl -w
 2 $miles = "";
 3
 4 until ($miles eq "0"){
 5
 6   print "Enter 0 to exit\n";
 7   print "Please enter the number of miles: ";
 8   chomp($miles=<STDIN>);
 9
10   $kilometers = $miles * 1.6;
11   print "$miles miles is $kilometers km.\n\n";
12 }
```

```
$ e0310.plx
Enter 0 to exit
Please enter the number of miles: 8
8 miles is 12.8 km.

Enter 0 to exit
Please enter the number of miles: 0
0 miles is 0 km.
```

The script is pretty much the same. However, now it loops until `$miles` equals 0, which is the opposite of the previous script.

The do Loop

The while and the until loops iterate only if their condition is met. If they are not, the loops do not execute at all. The do loop iterates at least once - since the condition is tested at the end of the block. The do statement is used with both while and until statements. The do loop is constructed as follows:

```
do {
    statement block
} while(condition)
```

The two scripts created previously can be modified as follows to use the do statement.

e0311.plx

```
 1 #!/usr/bin/perl -w
 2
 3 do {
 4    print "Enter 0 to exit\n";
 5    print "Please enter the number of miles: ";
 6    chomp($miles=<STDIN>);
 7
 8    $kilometers = $miles * 1.6;
 9    print "$miles miles is $kilometers km.\n\n";
10
11 } while ($miles ne "0");
```

e0312.plx

```
 1 #!/usr/bin/perl -w
 2
 3 do {
 4    print "Enter 0 to exit\n";
 5    print "Please enter the number of miles: ";
 6    chomp($miles=<STDIN>);
 7
 8    $kilometers = $miles * 1.6;
 9    print "$miles miles is $kilometers km.\n\n";
10
11 } until ($miles eq "0");
```

The for Loop

A for loop iterates a given number of times as determined by the specified condition; if no condition is specified, it loops infinitely. The for loop has the following structure:

```
for (initialization; condition; increment){
    statement block
}
```

The for statement in Perl is very similar to for loops in the Java™ programming language and the C programming language. In fact, the for statement is rarely used in Perl except by programmers who are used to this syntax (the foreach loop is much more common). An example is shown as follows:

```
e0313.plx
  1 #!/usr/bin/perl -w
  2
  3 for ($i=0; $i <= 10; $i++) {
  4
  5   print "\$i = $i\n";
  6
  7 }

$ e0313.plx
$i = 0
$i = 1
$i = 2
$i = 3
$i = 4
$i = 5
$i = 6
$i = 7
$i = 8
$i = 9
$i = 10
```

The statement sets $i equal to 0 and then increments $i by 1 each time the loop is executed. The loop continues to iterate until $i equals 10. The script prints the value of $i for each iteration.

It is possible to add a little complexity to a `for` loop. In a `for` loop, the following facts are important to keep in mind:

- Initialization takes place once before the first iteration.
- The condition is evaluated once before every iteration.
- If the condition is true, the loop block is executed.
- Variables are incremented after every iteration.

With these facts in mind, each part of a `for` statement can contain multiple expressions, which are separated by commas. For example:

e0314.plx

```
1 #!/usr/bin/perl -w
2
3 for ($i=0, $j=10; $i <= 10; $i++, $j--) {
4
5   print "Going up: $i  Going down: $j\n" ;
6
7 }
```

This `for` loop uses two counters: one to count up and one to count down. The output of this script is shown as follows:

```
$ e0314.plx
Going up: 0  Going down: 10
Going up: 1  Going down: 9
Going up: 2  Going down: 8
Going up: 3  Going down: 7
Going up: 4  Going down: 6
Going up: 5  Going down: 5
Going up: 6  Going down: 4
Going up: 7  Going down: 3
Going up: 8  Going down: 2
Going up: 9  Going down: 1
Going up: 10  Going down: 0
```

The foreach Loop

A foreach loop is actually just an alias for for. Perl recognizes the actual loop type from the semantics used. This loop iterates through all the elements of a list, and is the most commonly used loop construct in Perl. The statement is structured as follows:

```
foreach $loop (LIST){
    statement block
}
```

or

```
for $loop (LIST){
    statement block
}
```

Each time the loop iterates, one of the elements in the list is assigned to a temporary variable ($loop in the previous example). The $loop variable is not a copy of the list element but a reference to the element. Therefore, the list itself is changed when $loop is changed. Moreover $loop is a local variable within the loop; it is not available after the loop is terminated. The following is an example script that uses a foreach statement.

e0315a.plx

```
 1 #!/usr/bin/perl -w
 2
 3 print "Enter three words to be reversed\n";
 4 print "Please enter the first word: ";
 5 chomp($word1 = <STDIN>);
 6
 7 print "Please enter the second word: ";
 8 chomp($word2 = <STDIN>);
 9
10 print "Please enter the third word: ";
11 chomp($word3 = <STDIN>);
12
13 print "Here are the results:\n";
14
15 # Use loop iterate through the list
16 for $word ($word1, $word2, $word3) {
17   reversed = reverse($word);
18   print "$word backwards is: $reversed\n";
19 }
```

```
$ e0315a.plx
Enter three words to be reversed
Please enter the first word: los angeles
Please enter the second word: bob
Please enter the third word: cat
Here are the results:
los angeles backwards is: selegna sol
bob backwards is: bob
cat backwards is: tac
```

Lines 16–19, on the preceding page, are where the action takes place. Notice from the output that each value entered is processed in turn. The temp variable `$word` is used each time instead of three variables, such as `$word1`, `$word2`, and `$word3`, being used.

Here Documents

Here documents (or "heredocs") provide one of the ways to output formatted text from a Perl script. For example, the following script displays a small piece of verse from a famous play.

e0315b.plx

```
  1 #!/usr/bin/perl -w
  2
  3 print "Start here\n\n";
  4
  5 print <<end_of_text;
  6
  7 Our revels now are ended. These our actors,
  8 As I foretold you, were all spirits and
  9 Are melted into air, into thin air:
 10 And, like the baseless fabric of this vision,
 11 The cloud-capp'd towers, the gorgeous palaces,
 12 The solemn temples, the great globe itself,
 13 Ye all which it inherit, shall dissolve
 14 And, like this insubstantial pageant faded,
 15 Leave not a rack behind. We are such stuff
 16 As dreams are made on, and our little life
 17 Is rounded with a sleep.
 18
 19 end_of_text
 20
 21 print "End here\n";
```

```
$ e0315b.plx
Start here

Our revels now are ended. These our actors,
As I foretold you, were all spirits and
Are melted into air, into thin air:
And, like the baseless fabric of this vision,
The cloud-capp'd towers, the gorgeous palaces,
The solemn temples, the great globe itself,
Ye all which it inherit, shall dissolve
And, like this insubstantial pageant faded,
Leave not a rack behind. We are such stuff
As dreams are made on, and our little life
Is rounded with a sleep.

End here
```

The format of a Here document is essentially this:

```
print <<"some_tag";
    <The text to be printed goes here>
some_tag
```

The text between the tags is printed out exactly as formatted. If the opening tag is not quoted or is enclosed in double-quotes, the contents are interpolated (that is, the values of any variables contained in the text are printed). If the tag is enclosed in single quotes, no interpolation takes place. Back-quoting passes the heredoc contents to the shell for execution. Note also that there is no space between the `<<` and the tag: if a space or a null string (`""`) is used as the tag, or a space precedes the tag, then the next blank line will terminate the heredoc.

Here documents allow several `print` statements to be combined or simple menus to be created. For example, a script with a simple menu might look like this:

e0315c.plx

```
  1 #!/usr/bin/perl -w
  2 $choice = "";
  3
  4 print <<end_of_menu;
  5 Please select one of the following choices:
  6   A. Print this
  7   B. Print that
  8
  9 Your choice:
 10 end_of_menu
 11 chomp ($choice = <STDIN>);
 12 $choice = uc($choice);
 13
 14 if ($choice eq "A") {
 15   print "this\n";
 16 }
 17 elsif ($choice eq "B") {
 18   print "that\n";
 19 }
 20 else {
 21   print "Enter A or B please.\n";
 22 }
```

```
$ e0315c.plx
Please select one of the following choices:
  A. Print this
  B. Print that

Your choice:
a
this
```

The menu with the Here document is used instead of several print statements. Notice that indentation is maintained when the script executes.

The end tag must be on a line by itself, without any white space either preceding or following. Therefore, to get the input prompt displayed on the same line as "Your Choice:" a separate `print` statement is needed.

Exercise: Create Scripts Using `if` Statements and Loops

In this exercise, you complete the following tasks:

- Print a message using an `if` statement
- Print a message using an `if/else` statement
- Print a message using a compound `if/elsif/else` statement
- Use a `for` loop to display a list of numbers
- Use a `foreach` loop to perform calculations on a list of numbers
- Use `while` loops to repeatedly perform logical tests using an `if/elsif/else` statement
- Print formatted text with a Here document

Tasks

Complete the following steps:

1. Create a script that finds out who is buried in King Tut's tomb. Use an `if` statement to test the answer to see if it is correct.

```
Who is buried in King Tut's tomb?
  A. Elvis
  B. Kilroy
  C. King Tut
  D. King Arthur

Your answer: c
Correct, King Tut is buried in King Tut's tomb.
```

2. Modify the script you created in Step 1 to display a message if an incorrect answer is entered.

```
Who is buried in King Tut's tomb?
  A. Elvis
  B. Kilroy
  C. King Tut
  D. King Arthur

Your answer: a
Incorrect answer, try again.
```

3. Modify the script you created in Step 2 to display a witty answer for each incorrect answer. For example:

```
Who is buried in King Tut's tomb?
  A. Elvis
  B. Kilroy
  C. King Tut
  D. King Arthur

Your answer: a
Elvis lives in Michigan silly!
```

4. Modify the script from Step 3 to loop around the question until a correct answer is entered. Create a separate script for each of the following loop types:

   a. Create the script using a `while` loop.

   b. Create the script using an `until` loop.

   c. Create the script using a `do/while` loop.

   d. Create the script using a `do/until` loop.

   The output from the script should look like this:

```
Who is buried in King Tut's tomb?
  A. Elvis
  B. Kilroy
  C. King Tut
  D. King Arthur

Your answer: b
Kilroy visited the tomb, but he's not there.
Who is buried in King Tut's tomb?
  A. Elvis
  B. Kilroy
  C. King Tut
  D. King Arthur

Your answer: c
Correct, King Tut is buried in King Tut's tomb.
```

5. Write a simple script that uses a loop to print "Hello World! " the number of times specified by the user. For example, the output of your script should look something like this:

```
Enter the number of times to display: 3
Hello World! Hello World! Hello World!
```

   a. Use a `while` loop to create the script.

   b. Use a `for` loop to create the script.

6. Create an ASCII table with Perl.

   a. Write a script that given a letter displays the ASCII table from that letter to the ASCII value 127. Use the `ord` function to convert a letter into its ASCII value. Use `chr` to convert an ASCII value to a letter. The output should look like this:

```
Enter a letter: a
Letter  Value
  a      97
  b      98
  c      99
  d      100
  e      101
and so on ...
```

   b. Modify the script to accept an ASCII value instead of a letter.

```
Enter an ASCII value: 70
Value  Letter
  70      F
  71      G
  72      H
  73      I
and so on ...
```

7. Create a script that reads in three numbers and then computes the square, the cube, and the square root of each number using a `foreach` loop. Your output might look something like this:

   Sample output:

```
Please enter three numbers.
First number: 4
Second number: 9
Third number: 16
The results for 4 are:
4**2 = 16
4**3 = 64
sqrt(4) = 2

The results for 9 are:
9**2 = 81
9**3 = 729
sqrt(9) = 3

The results for 16 are:
16**2 = 256
16**3 = 4096
sqrt(16) = 4
```

Statement Modifiers

Remember how the numeric operators had a shorthand for common operations. For example:

```
$x = $x + 2; # is equivalent to
$x += 2;
```

Well, loops and `if` statements also have a shorthand called *statement modifiers*. At first glance, statement modifiers look like the condition and the statement are mixed up by mistake. Take a look.

```
statement if condition;
statement unless condition;
statement while condition;
statement until condition;
statement for condition;
```

Here are some examples:

```
e0316.plx
  1 #!/usr/bin/perl -w
  2 $a = "Hello";
  3 $b = 1;
  4
  5 print "$a World!\n" if $a eq "Hello";
  6 print $b++,"\n" until $b == 6;

$ e0316.plx
Hello World!
1
2
3
4
5
```

Line 5 shows a basic string test. Line 6 shows a simple counter in action.

With statement modifiers, simple `if` expressions and loops are written in an optimized form. The only restriction on statement modifiers is that statements must be simple expressions; no statement blocks are allowed.

Note – Do not confuse the statement modifiers with `do`. With statement modifiers, the condition is always checked before any statements are executed.

Loop Controls

Sometimes, execution within a block or a loop must be aborted. For example, when a test condition is met and the rest of the statements in the block need to be skipped. Perl offers three statements to control a loop, which are described in Table 3-1.

Table 3-1 Loop Controls

Control	Description
`last`	Exit the loop
`next`	Skip the rest of the block and begin the next iteration
`redo`	Restart the current iteration

The following script demonstrates the use of these commands.

```
e0317.plx
  1 #! /usr/bin/perl -w
  2 # Test behavior of loop
  3 for ($i=0 ; $i <= 10 ; $i++) {
  4     print "$i ";
  5 #    next;
  6 #    redo;
  7 #    last;
  8     print $i*10," ";
  9 }
 10
 11 print "Goodbye...\n";

$ e0317.plx
0 0 1 10 2 20 3 30 4 40 5 50 6 60 7 70 8 80 9 90 10 100
Goodbye...
```

The script is a simple `for` loop that prints the incremented variable `$i` and `$i * 10`. The following sections describe what the output would be if Lines 5–7 were uncommented in turn.

The next Loop Control

If you uncomment the line containing next, the following output is produced:

```
$ e0317a.plx
0 1 2 3 4 5 6 7 8 9 10 Goodbye...
```

After the script hits the next command on Line 5, the script skips down to Line 9 and then iterates again. Thus, only the first print statement is executed with each iteration. After the for loop is complete, the final print command on Line 11 is executed.

The redo Loop Control

If you uncomment the line containing redo, the following output is produced.

```
$ e0317b.plx
0 0 0 0 0 0 0 0....
```

With redo, the script jumps to Line 4, which is the first statement of the for statement block. However, the loop does not iterate. The script remains in the same iteration in which it started. Thus, $i never gets incremented and it stays at 0. Essentially, this results in an infinite loop.

The last Loop Control

If you uncomment the line containing the last statement, the following output is produced:

```
$ e0317c.plx
0 Goodbye...
```

When the last statement is encountered in the loop, Perl immediately jumps to the end of the loop and executes the first statement after the loop. The first print statement is executed only once.

Using Loop Control and Statement Modifiers

With the concepts described in the last two sections, the previous miles-to-kilometer conversion script can be updated to function better. For example, the following script has been updated to use "q" to exit the script instead of 0. Using `last` and statement modifiers makes this possible.

e0318.plx

```
1 #!/usr/bin/perl -w
2
3 while ( 1 ){ # Infinite loop
4    print "Enter q to exit\n";
5    print "Please enter the number of miles: ";
6    chomp($miles = <STDIN>);
7
8    last if $miles eq "q";
9
10   $kilometers = $miles * 1.6;
11   print "$miles miles is $kilometers km.\n\n";
12 }
```

```
$ e0318.plx
Enter q to exit
Please enter the number of miles: 3
3 miles is 4.8 km.

Enter q to exit
Please enter the number of miles: q
```

Searching is another good use of `last` and statement modifiers. The following script uses the `rand` function to randomly generate a number. Then a loop is used until the number is guessed.

e0319.plx

```
  1 #!/usr/bin/perl -w
  2 # Number guesser
  3 $max = 100;
  4
  5 $num = int(rand $max) + 1;
  6
  7 print "I have picked a number between 1 and $max.\n";
  8
  9 while (1){
 10    print "Guess ",++$guessNum,": ";
 11    chomp($guess=<STDIN>);
 12
 13    print "Higher!\n" if $guess < $num;
 14    print "Lower!\n" if $guess > $num;
 15    last if $guess == $num;
 16 }
 17 print "Right! You needed $guessNum guesses to guess $num\n"
```

```
$ e0319.plx
I have picked a number between 1 and 100.
Guess 1: 50
Lower!
Guess 2: 25
Lower!
Guess 3: 12
Higher!
Guess 4: 18
Higher!
Guess 5: 22
Right! You needed 5 guesses to guess 22
```

The `last` statement creates the exit condition within the infinite loop. The script keeps looping until a match is found. The logic of the `print` statements is made much easier because of the statement modifiers.

Labels

Perl labels allow blocks of code or loops to be given a name. By convention, labels are written in uppercase to prevent conflicts with Perl reserved words. For example, the previous example can be rewritten to use labels like this:

e0320a.plx
```
 1 #! /usr/bin/perl -w
 2 # Test behavior of loop
 3 TOP: for ($i=0; $i <= 10; $i++) {
 4     print "$i ";
 5
 6     next TOP;
 7
 8     print "Got here";
 9 }
10
11 print "Goodbye...\n";
```

```
$ e0320a.plx
0 1 2 3 4 5 6 7 8 9 10 Goodbye...
```

This can make program flow more explicit.

The following example demonstrates how you can use labels to create loops.

e0320b.plx

```
 1 #! /usr/bin/perl -w
 2 $i = 1;
 3
 4 TOP: {
 5   print "$i ";
 6   $i++;
 7   last TOP if ($i == 10);
 8   redo TOP;
 9
10   print "Got here";
11 }
12
13 print "Goodbye...\n";
```

```
$ e0320b.plx
1 2 3 4 5 6 7 8 9 Goodbye...
```

The `redo` statement on line 8 makes the loop infinite. The statement modifier on line 7 jumps out of the loop.

Labels are commonly used with loops embedded in another loop. Labeling the loop allows more control with the `next`, `last`, and `redo` statements. Without a label, the three commands always apply to the innermost loop.

e0321.plx

```
 1 #!/usr/bin/perl -w
 2
 3 # Outer and Inner Loops
 4 # Find first x * y > 6
 5 $x = 0;
 6
 7 OUTER: while ($x < 10) {
 8   $x++;
 9   INNER: for($y=0; $y < 5; $y++){
10     last OUTER  if (($x * $y) > 6);
11     print "Inner: x=$x y=$y x*y=", $x*$y, "\n";
12   }
13   print "Outer: x=$x y=$y x*y=", $x*$y, "\n";
14 }
15 print "Jumped out.\n";
16 print "x = $x and y = $y\n";
17 print "$x * $y = ",$x*$y,"\n";
```

```
$ e0321.plx
Inner: x=1 y=0 x*y=0
Inner: x=1 y=1 x*y=1
Inner: x=1 y=2 x*y=2
Inner: x=1 y=3 x*y=3
Inner: x=1 y=4 x*y=4
Outer: x=1 y=5 x*y=5
Inner: x=2 y=0 x*y=0
Inner: x=2 y=1 x*y=2
Inner: x=2 y=2 x*y=4
Inner: x=2 y=3 x*y=6
Jumped out.
x = 2 and y = 4
2 * 4 = 8
```

When the product of `$x` and `$y` is greater than 6, the `last` statement causes the script to jump out of both loops. Note that the `print` statement in the outer loop is, in fact, skipped when `last` is called.

Switch or Case Constructs in Perl

Many programming languages contain a conditional statement, such as switch or case, that allows branching based on a single value. There is no official switch or case statement in Perl because there are already several ways to write the equivalent construct. The examples that follow show different ways such a construct can be implemented in Perl.

e0322a.plx

```
  1 #!/usr/bin/perl -w
  2
  3 print "Enter a number from 1 to 3: ";
  4 chomp($a = <STDIN>);
  5
  6  if    ($a == 1){ print "1 selected\n" }
  7  elsif ($a == 2){ print "2 selected\n" }
  8  elsif ($a == 3){ print "3 selected\n" }
  9  else           { print "None of the above\n"  }
  10
  11 print "End of switch\n";
```

```
$ e0322a.plx
Enter a number from 1 to 3: 2
2 selected
End of switch
```

This example shows the use of `if/elsif/else` statements to create a case structure.

Another method is to use the `last` statement within a code block.

e0322b.plx

```
1  #!/usr/bin/perl -w
2
3  print "Enter a number from 1 to 3: ";
4  chomp($a = <STDIN>;
5
6  { # The "last" statement applies to this block
7    ($a == 1) && do { print "1 selected\n"; last };
8    ($a == 2) && do { print "2 selected\n"; last };
9    ($a == 3) && do { print "3 selected\n"; last };
10   print "None of the above\n";
11 }
12
13 print "End of switch\n";
```

```
$ e0322b.plx
Enter a number from 1 to 3: 5
None of the above
End of switch
```

If the statement is true, the `do` block gets executed. If it is false, the script skips to the next statement.

Switch or Case Constructs in Perl 5.10

The given-when is Perl's new version of the switch statement. It can operate like a switch statement or it can be a little more versatile. Lets look at an example.

e0323.plx

```
1  #!/usr/bin/perl -w
2
3  use 5.010;
4
5  say "Enter a number from 1 to 3: ";
6  chomp($a = <STDIN>);
7
8  given ( $a ) {
9
10   when ($a == 1){ say "1 selected" ;  }
11   when ($a == 2){ say "2 selected" ;  }
12   when ($a == 3){ say "3 selected" ; }
13   default { say "None of the above" ; }
14 }
15
16 say "End of switch";
```

The example shows that a variable can be introduced in the given and then each of the when statements is checked until one of them evaluates to true. When one evaluates to true, its code is executed and the given block is exited. If none of them evaluate to true, the default will be selected. The default is optional. It behaves like a when block which always evaluates to true.

Since this could have been accomplished with if-elsif-else code, there was no strong need to introduce a new control structure. However, the given-when can do more.

The code above uses breaks. It is as if the given-when code were actually written like this:

```
given ( $a ) {

     when ($a == 1){ say "1 selected" ; break;  }
     when ($a == 2){ say "2 selected" ; break;  }
     when ($a == 3){ say "3 selected" ; break;  }
     default { say "None of the above" ; break; }
   }
```

Suppose Perl wanted to find a multiple of 2, 3 or 5. Code like the example above does not report that a number was a multiple of more than factor. However, if continue is used at the end of the appropriate when blocks, Perl can continue to examine other when conditions. The code to accomplish this search for multiple whens can be written like this:

e0324.plx

```
1  #!/usr/bin/perl -w
2
3  use 5.010;
4
5  say "Enter a number: ";
6  chomp($a = <STDIN>);
7
8   given ( $a ) {
9
10    when ($a % 2 == 0){ say "multiple of 2 selected" ; continue }
11    when ($a % 3 == 0){ say "multiple of 3 selected" ; continue }
12    when ($a % 5 == 0){ say "multiple of 5 selected" }
13    #default { say "Not a multiple of 2, 3, or 5." }
14  }
15
16  say "End of given-when search";
```

Notice that the logical use of the default block is gone. If the default block is reached, it is always executed. A continue would always allow us to reach this block.

Exercise: Create Scripts Using Control Structures

In this exercise, you complete the following tasks:

- Exit from a loop using loop controls and statement modifiers
- Use a switch construct in Perl to print a message

Tasks

Complete the following steps:

8. Modify the Celsius-to-Fahrenheit converter you created previously. Use a switch construct to print the following messages based on the converted Fahrenheit value.

   - If the temperature is greater than 88, print “It’s too hot!”.
   - If the temperature is between 69 and 88, print “Ah, Just right!”.
   - If the temperature is between 33 and 68, print “Kinda cool.”
   - If the temperature is below 33 degrees, print “Brrrr, it’s cold!”.

   Allow the user to enter different values until “quit” is entered. Use a statement modifier to exit the loop.

   Sample output:

```
Please enter the temperature in Celsius: 33
33 degrees Celsius is 91.4 degrees Fahrenheit
It’s hot!

Please enter the temperature in Celsius: 22
22 degrees Celsius is 71.6 degrees Fahrenheit
Ah, just right!

Please enter the temperature in Celsius: quit
```

9. Write a script that checks if a word is a Palindrome. A Palindrome is a word or phrase that is identical when it is spelled forward or backward. Use the `lc` function to make sure case is not an issue. Allow the script to loop until "quit" is entered. Use a statement modifier to exit the loop.

   Sample output:

```
Enter a word (or 'quit' to leave) not a ton
not a ton is a Palindrome!!

Enter a word (or 'quit' to leave) ere i ere
ere i ere is a Palindrome!!

Enter a word (or 'quit' to leave) i ere
Not a Palindrome-----

Enter a word (or 'quit' to leave) Otto
otto is a Palindrome!!

Enter a word (or 'quit' to leave) quit
```

10. (Optional) Write a script that plays a simple version of the dice game craps. The rules for the game are simplified from the real game.

    Your script should use the following rules.

    - The player can bet some amount of currency to start the game. The player then begins rolling the dice.
    - If the player rolls a 7 or 11, the player wins and doubles the amount of money bet.
    - If the player rolls a 2 or 12, the player loses all the money bet.
    - If the player does not roll a 2, 7, 11, or 12 the player continues to roll until one of the winning or losing numbers is rolled.
    - After the current game is over, if the player won, the player can let the winnings ride (bet all the money) and play again or quit.
    - After losing, a player can start over or quit.

    Hints: Use an outer loop to control if the player plays again. Use an inner loop to play each game and roll the dice.

Sample output:

```
$ craps.plx

Welcome to the Perl craps game
Please enter the amount you wish to bet: 45
Here is the roll...
You rolled a 1 and a 6 ...
You have won 90 credits. Let it ride? (y/n) y
Here is the roll...
You rolled a 3 and a 5 ...
Press return to roll again.

Here is the roll...
You rolled a 6 and a 5 ...
You have won 180 credits. Let it ride? (y/n) n
You keep 180 credits! See you next time.
$



$ craps.plx

Welcome to the Perl craps game
Please enter the amount you wish to bet: 25
Here is the roll...
You rolled a 4 and a 5 ...
Press return to roll again.

Here is the roll...
You rolled a 3 and a 4 ...
You have won 50 credits! Let it ride? (y/n) y
Here is the roll...
You rolled a 2 and a 3 ...
Press return to roll again.

Here is the roll...
You rolled a 1 and a 2 ...
Sorry, but you've lost. Would you like to play again?
(y/n)n
Better luck next time.
$
```

11. Write a script that asks a user to input a number. If the number is a multiple of three, print foo. If it is a multiple of five, print bar. If it is a multiple of seven, print baz.

Sample output:

```
Enter a number.  60
foo bar
```

Exercise Summary

Discussion – Take a few minutes to discuss the experiences, issues, or discoveries you had during the lab exercises.

- Experiences
- Interpretations
- Conclusions
- Applications

Exercise Solutions

1. Create a script that finds out who is buried in King Tut's tomb. Use an `if` statement to test the answer to see if it is correct.

```
Who is buried in King Tut's tomb?
  A. Elvis
  B. Kilroy
  C. King Tut
  D. King Arthur

Your answer: c
Correct, King Tut is buried in King Tut's tomb.
```

Suggested solution:

lab0301.plx

```
 1 #!/usr/bin/perl -w
 2 $answer = "";
 3
 4 print <<end_of_text;
 5 Who is buried in King Tut's tomb?
 6   A. Elvis
 7   B. Kilroy
 8   C. King Tut
 9   D. King Arthur
10
11 end_of_text
12
13 print "Your answer: ";
14 chomp($answer = <STDIN>);
15
16 if (uc($answer) eq "C"){
17   print "Correct, King Tut is buried in King Tut's tomb.\n ";
18 }
```

2. Modify the script you created in Step 1 to display a message if an incorrect answer is entered.

```
Who is buried in King Tut's tomb?
  A. Elvis
  B. Kilroy
  C. King Tut
  D. King Arthur

Your answer: a
Incorrect answer, try again.
```

Suggested solution:

lab0302.plx

```
 1 #!/usr/bin/perl -w
 2 $answer = "";
 3
 4 print <<end_of_text;
 5 Who is buried in King Tut's tomb?
 6   A. Elvis
 7   B. Kilroy
 8   C. King Tut
 9   D. King Arthur
10
11 end_of_text
12
13 print "Your answer: ";
14 chomp($answer = <STDIN>);
15
16 if (uc($answer) eq "C"){
17   print "Correct, King Tut is buried in King Tut's tomb.\n ";
18 }
19
20 else {
21   print "Incorrect answer, try again.\n";
22 }
```

3. Modify the script you created in Step 2 to display a witty answer for each incorrect answer. For example:

```
Who is buried in King Tut's tomb?
  A. Elvis
  B. Kilroy
  C. King Tut
  D. King Arthur

Your answer: a
Elvis lives in Michigan silly!
```

Suggested solution:

lab0303.plx

```
 1 #!/usr/bin/perl -w
 2 $answer = "";
 3
 4 print <<end_of_text;
 5 Who is buried in King Tut's tomb?
 6   A. Elvis
 7   B. Kilroy
 8   C. King Tut
 9   D. King Arthur
10
11 end_of_text
12
13 print "Your answer: ";
14 chomp($answer = <STDIN>);
15 $answer = uc($answer);
16
17 if ($answer eq "C"){
18   print "Correct, King Tut is buried in King Tut's tomb.\n ";
19 }
20
21 elsif ($answer eq "A") {
22   print "Elvis lives in Michigan silly!\n";
23 }
24
25 elsif ($answer eq "B") {
26   print "Kilroy visited the tomb, but he's not there.\n";
27 }
28
29 elsif ($answer eq "D") {
30   print "King Arthur is buried in England.\n";
31 }
32
33 else {
34   print "Incorrect answer, try A, B, C, or D\n";
35 }
```

4. Modify the script from Step 3 to loop around the question until a correct answer is entered. Create a separate script for each of the following loop types:

   a. Create the script using a `while` loop.

   b. Create the script using an `until` loop.

   c. Create the script using a `do/while` loop.

   d. Create the script using a `do/until` loop.

   The output from the script should look like this:

```
Who is buried in King Tut's tomb?
  A. Elvis
  B. Kilroy
  C. King Tut
  D. King Arthur

Your answer: b
Kilroy visited the tomb, but he's not there.
Who is buried in King Tut's tomb?
  A. Elvis
  B. Kilroy
  C. King Tut
  D. King Arthur

Your answer: c
Correct, King Tut is buried in King Tut's tomb.
```

Suggested solutions:

lab0304a.plx

```
 1 #!/usr/bin/perl -w
 2 $answer = "";
 3
 4 while ($answer ne "C") {
 5 print <<end_of_text;
 6 Who is buried in King Tut's tomb?
 7   A. Elvis
 8   B. Kilroy
 9   C. King Tut
10   D. King Arthur
11
12 end_of_text
13
14   print "Your answer: ";
15   chomp($answer = <STDIN>);
16   $answer = uc($answer);
17
18   if ($answer eq "C"){
19     print "Correct, King Tut is buried in King Tut's tomb.\n ";
20   }
21
22   elsif ($answer eq "A") {
23     print "Elvis lives in Michigan silly!\n";
24   }
25
26   elsif ($answer eq "B") {
27     print "Kilroy visited the tomb, but he's not there.\n";
28   }
29
30   elsif ($answer eq "D") {
31     print "King Arthur is buried in England.\n";
32   }
33
34   else {
35     print "Incorrect answer, try A, B, C, or D\n";
36   }
37 }
```

lab0304b.plx

```
 1 #!/usr/bin/perl -w
 2 $answer = "";
 3
 4 until ($answer eq "C") {
 5 print <<end_of_text;
 6 Who is buried in King Tut's tomb?
 7   A. Elvis
 8   B. Kilroy
 9   C. King Tut
10   D. King Arthur
11
12 end_of_text
13
14   print "Your answer: ";
15   chomp($answer = <STDIN>);
16   $answer = uc($answer);
17
18   if ($answer eq "C"){
19     print "Correct, King Tut is buried in King Tut's tomb.\n ";
20   }
21
22   elsif ($answer eq "A") {
23     print "Elvis lives in Michigan silly!\n";
24   }
25
26   elsif ($answer eq "B") {
27     print "Kilroy visited the tomb, but he's not there.\n";
28   }
29
30   elsif ($answer eq "D") {
31     print "King Arthur is buried in England.\n";
32   }
33
34   else {
35     print "Incorrect answer, try A, B, C, or D\n";
36   }
37 }
```

lab0304c.plx

```
 1 #!/usr/bin/perl -w
 2 $answer = "";
 3
 4 do {
 5 print <<end_of_text;
 6 Who is buried in King Tut's tomb?
 7   A. Elvis
 8   B. Kilroy
 9   C. King Tut
10   D. King Arthur
11
12 end_of_text
13
14   print "Your answer: ";
15   chomp($answer = <STDIN>);
16   $answer = uc($answer);
17
18   if ($answer eq "C"){
19     print "Correct, King Tut is buried in King Tut's tomb.\n ";
20   }
21
22   elsif ($answer eq "A") {
23     print "Elvis lives in Michigan silly!\n";
24   }
25
26   elsif ($answer eq "B") {
27     print "Kilroy visited the tomb, but he's not there.\n";
28   }
29
30   elsif ($answer eq "D") {
31     print "King Arthur is buried in England.\n";
32   }
33
34   else {
35     print "Incorrect answer, try A, B, C, or D\n";
36   }
37 }while ($answer ne "C");
38
```

lab0304d.plx

```
 1 #!/usr/bin/perl -w
 2 $answer = "";
 3
 4 do {
 5 print <<end_of_text;
 6 Who is buried in King Tut's tomb?
 7   A. Elvis
 8   B. Kilroy
 9   C. King Tut
10   D. King Arthur
11
12 end_of_text
13
14   print "Your answer: ";
15   chomp($answer = <STDIN>);
16   $answer = uc($answer);
17
18   if ($answer eq "C"){
19     print "Correct, King Tut is buried in King Tut's tomb.\n ";
20   }
21
22   elsif ($answer eq "A") {
23     print "Elvis lives in Michigan silly!\n";
24   }
25
26   elsif ($answer eq "B") {
27     print "Kilroy visited the tomb, but he's not there.\n";
28   }
29
30   elsif ($answer eq "D") {
31     print "King Arthur is buried in England.\n";
32   }
33
34   else {
35     print "Incorrect answer, try A, B, C, or D\n";
36   }
37 } until ($answer eq "C");
```

5. Write a simple script that uses a loop to print "Hello World! " the number of times specified by the user. For example, the output of your script should look something like this:

```
Enter the number of times to display: 3
Hello World! Hello World! Hello World!
```

   a. Use a `while` loop to create the script.

   b. Use a `for` loop to create the script.

   Suggested solutions:

lab0305a.plx

```
 1 #!/usr/bin/perl -w
 2
 3 print "Enter the number of times to display: ";
 4 chomp($num = <STDIN>);
 5
 6 $x = 1;
 7
 8 while ($x <= $num) {
 9   print "Hello World! ";
10   $x++;
11 }
12 print "\n"
```

lab0305b.plx

```
 1 #!/usr/bin/perl -w
 2
 3 print "Enter the number of times to display: ";
 4 chomp($num = <STDIN>);
 5
 6 for ($x = 1; $x <= $num; $x++) {
 7   print "Hello World! ";
 8 }
 9 print "\n";
```

6. Create an ASCII table with Perl.

   a. Write a script that given a letter displays the ASCII table from that letter to the ASCII value 127. Use the `ord` function to convert a letter into its ASCII value. Use `chr` to convert an ASCII value to a letter. The output should look like this:

```
Enter a letter: a
Letter  Value
  a      97
  b      98
  c      99
  d      100
  e      101
and so on...
```

   b. Modify the script to accept an ASCII value instead of a letter.

```
Enter an ASCII value: 70
Value  Letter
  70      F
  71      G
  72      H
  73      I
and so on...
```

Suggested solutions:

lab0306a.plx

```
 1 #!/usr/bin/perl -w
 2
 3 print "Enter a letter: ";
 4 chomp($letter = <STDIN>);
 5
 6 print "Letter  Value\n";
 7
 8 for ( $a = ord($letter); $a <= 127 ; $a++){
 9   $next = chr($a);
10   print "  $next      $a\n";
11 }
```

lab0306b.plx

```
 1 #!/usr/bin/perl -w
 2
 3 print "Enter an ASCII value: ";
 4 chomp($number = <STDIN>);
 5
 6 print "Value  Letter\n";
 7
 8 for ( $a = $number; $a <= 127 ; $a++) {
 9   $next = chr($a);
10   print "  $a      $next\n";
11 }
```

7. Create a script that reads in three numbers and then computes the square, the cube, and the square root of each number using a `foreach` loop. Your output might look something like this:

   Sample output:

```
$ lab0307.plx
Please enter three numbers.
First number: 4
Second number: 9
Third number: 16
The results for 4 are:
4**2 = 16
4**3 = 64
sqrt(4) = 2

The results for 9 are:
9**2 = 81
9**3 = 729
sqrt(9) = 3

The results for 16 are:
16**2 = 256
16**3 = 4096
sqrt(16) = 4
```

Suggested solution:

lab0307.plx

```
 1 #!/usr/bin/perl -w
 2
 3 print "Please enter three numbers.\n";
 4 print "First number: ";
 5 chomp($num1 = <STDIN>);
 6
 7 print "Second number: ";
 8 chomp($num2 = <STDIN>);
 9
10 print "Third number: ";
11 chomp($num3 = <STDIN>);
12
13 foreach $num ($num1,$num2,$num3){
14   print "The results for $num are:\n";
15   print "$num^2 = ", ($num**2), "\n";
16   print "$num^3 = ",($num**3),"\n";
17   print "sqrt($num) = ",(sqrt($num)),"\n\n";
18 }
```

8. Modify the Celsius-to-Fahrenheit converter you created previously. Print the following messages based on the converted Fahrenheit value.

   - If the temperature is greater than 88, print "It's too hot!".
   - If the temperature is between 69 and 88, print "Ah, Just right!".
   - If the temperature is between 33 and 68, print "Kinda cool."
   - If the temperature is below 33 degrees, print "Brrrr, it's cold!".

   Allow the user to enter different values until "quit" is entered. Use a statement modifier to exit the loop.

   Sample output:

```
$ lab0308.plx
Please enter the temperature in Celsius: 33
33 degrees Celsius is 91.4 degrees Fahrenheit
It's hot!

Please enter the temperature in Celsius: 22
22 degrees Celsius is 71.6 degrees Fahrenheit
Ah, just right!

Please enter the temperature in Celsius: quit
```

Suggested solution:

lab0308.plx

```
 1 #!/usr/bin/perl -w
 2 $celsius = "";
 3
 4 while ($celsius ne "quit"){
 5
 6   print "\nPlease enter the temperature in Celsius: ";
 7   chomp($celsius = <STDIN>);
 8   last if ( lc($celsius) eq "quit" );
 9
10   $fahr = (9/5) * $celsius + 32;
11   print "$celsius degrees Celsius is $fahr degrees Fahrenheit\n";
12
13   SWITCH:{
14
15     if ($fahr > 88) { print "It's hot!\n\n"; last SWITCH;}
16     if ($fahr > 68) { print "Ah, just right!\n\n"; last SWITCH; }
17     if ($fahr > 32) { print "Kinda cool.\n\n"; last SWITCH;  }
18     print "Brrrrr, it's cold!\n\n";
19   }
20 } # end while
21
```

9. Write a script that checks if a word is a Palindrome. A Palindrome is a word or phrase that is identical when it is spelled forward or backward. Use the `lc` function to make sure case is not an issue. Allow the script to loop until "quit" is entered. Use a statement modifier to exit the loop.

   Sample output:

```
Enter a word (or 'quit' to leave) not a ton
not a ton is a Palindrome!!

Enter a word (or 'quit' to leave) ere i ere
ere i ere is a Palindrome!!

Enter a word (or 'quit' to leave) i ere
Not a Palindrome-----

Enter a word (or 'quit' to leave) Otto
otto is a Palindrome!!

Enter a word (or 'quit' to leave) quit
```

Suggested solution:

lab0309.plx

```
 1 #!/usr/bin/perl -w
 2 $word = "";
 3
 4 while ($word ne "quit") {
 5   print "Enter a word (or 'quit' to leave) ";
 6   $word = <STDIN>;
 7   chomp($word) ;
 8
 9   $word = lc($word) ; # Make lowercase
10   last if ($word eq "quit"); # Exit if quit
11
12   if ($word eq reverse($word)) {
13     print "$word is a Palindrome!!\n\n";
14   }
15   else { print "Not a Palindrome-----\n\n"; }
16 }
```

10. (Optional) Write a script that plays a simple version of the dice game craps. The rules for the game are simplified from the real game.

    Your script should use the following rules.

    - The player can bet some amount of currency to start the game. The player then begins rolling the dice.
    - If the player rolls a 7 or 11, the player wins and doubles the amount of money bet.
    - If the player rolls a 2 or 12, the player loses all the money bet.
    - If the player does not roll a 2, 7, 11, or 12 the player continues to roll until one of the winning or losing numbers is rolled.
    - After the current game is over, if the player won, the player can let the winnings ride (bet all the money) and play again or quit.
    - After losing, a player can start over or quit.

    Hints: Use an outer loop to control if the player plays again or not. Use an inner loop to play each game and roll the dice.

Sample output:

```
$ craps.plx

Welcome to the Perl craps game
Please enter the amount you wish to bet: 45
Here is the roll...
You rolled a 1 and a 6 ...
You have won 90 credits. Let it ride? (y/n) y
Here is the roll...
You rolled a 3 and a 5 ...
Press return to roll again.

Here is the roll...
You rolled a 6 and a 5 ...
You have won 180 credits. Let it ride? (y/n) n
You keep 180 credits! See you next time.
$



$ craps.plx

Welcome to the Perl craps game
Please enter the amount you wish to bet: 25
Here is the roll...
You rolled a 4 and a 5 ...
Press return to roll again.

Here is the roll...
You rolled a 3 and a 4 ...
You have won 50 credits! Let it ride? (y/n) y
Here is the roll...
You rolled a 2 and a 3 ...
Press return to roll again.

Here is the roll...
You rolled a 1 and a 2 ...
Sorry, but you've lost. Would you like to play again?
(y/n)n
Better luck next time.
$
```

Suggested solution:

craps.plx

```
  1 #!/usr/bin/perl -w
  2
  3 # Simple Craps Game script
  4 print "\nWelcome to the Perl craps game\n";
  5 $bet = 0;
  6 $quit = "n";
  7
  8 # Outer loop controls whether another game is played our not.
  9 do {
 10   if ($bet == 0) {
 11     print "Please enter the amount you wish to bet: ";
 12     chomp($bet = <STDIN>);
 13   }
 14
 15   $gameover = "false";
 16
 17   #Inner loop controls the current game.
 18   do {
 19     print "Here is the roll...\n";
 20     $dice1 = int(rand(6))+1;
 21     $dice2 = int(rand(6))+1;
 22
 23     print "You rolled a $dice1 and a $dice2 ...\n";
 24     $total = $dice1 + $dice2;
 25
 26     if ($total == 2 || $total == 3 || $total == 7 || $total == 11 ||
$total == 12){
 27       $gameover = "true";
 28
 29     } else {
 30       print "Press return to roll again.\n";
 31       $_ = <STDIN>;
 32     }
 33
 34   } while ($gameover eq "false"); # End Inner Loop
 35
 36   if ($total == 7 || $total == 11){
 37     $bet = $bet * 2;
 38     print "You have won $bet credits! Let it ride? (y/n) ";
 39     chomp($quit=<STDIN>);
 40
 41   }
 42   else {
```

```
 43
 44     $bet = 0;
 45     print "Sorry, but you've lost. Would you like to play again?
(y/n)";
 46     chomp($quit=<STDIN>);
 47
 48   }
 49
 50 }while ($quit eq "y"); # End Outer Loop
 51
 52 if ($bet > 0){
 53   print "You keep $bet credits! See you next time.\n";
 54 }
 55 else {
 56   print "Better luck next time.\n";
 57 }
```

11. Write a script that asks a user to input a number. If the number is a multiple of three, print foo. If it is a multiple of five, print bar. If it is a multiple of seven, print baz.

Sample output:

```
Enter a number.  60
foo bar
```

Suggested solutions:

```
1  #!/usr/bin/perl -w
2
3  use 5.010;
4  $output="";
5  say "Enter a number: ";
6  chomp($a = <STDIN>);
7
8  given ( $a ) {
9
10     when ($a % 3 == 0){ $output .= " foo"; continue }
11     when ($a % 5 == 0){ $output .= " bar" ; continue }
12     when ($a % 7 == 0){ $output .= " baz" }
13  }
14  if (!$output) {$output = "Not a multiple of 3, 5, or
7";}
15  say "$output";
```

Arrays

Objectives

Upon completion of this module, you should be able to:

- Create an array variable and assign scalar values to the array
- Determine the length of an array using an array operator
- Use array slices to assign new values to an array
- Determine the length of an array using a scalar variable
- Reverse the contents of an array
- Use `pop` to modify the contents of an array
- Sort an array
- Create an array from scalar using `split`
- Process the values passed in the command-line array, `@ARGV`
- Read the output of a UNIX command into an array

Relevance

Discussion – The following question is relevant to understanding arrays:

Discuss some lists in the real world. How do they work?

Introducing Arrays

The second basic variable type in Perl is the array. Array names start with an @. For example, the following statement assigns an array to a list of numbers:

```
@arr = (1, 3, 5, 7, 11, 13, 17);
```

An array contains a list of scalars. The single values are accessed by a subscript starting with zero.

Before examining initialization and array operations, how are lists defined?

Arrays Are Named Lists

In Perl, a list is a series of scalar values in which the order of the values is preserved. The most common way to construct a list is by placing a comma-separated group of scalars in parentheses. The comma is a list constructor operator in Perl, but since its precedence is low (the "=" operator, for example, has greater precedence), you have to enclose it in parentheses in order to make the list a single entity. For example:

```
(1, 3, 5, 7, 11, 13, 17)
```

A list may contain both numeric and string values.

```
(13, 23, "Smith", "Berlin")
```

Elements whose values are not yet determined can be set to `undef`.

```
(13, 23, undef, "Berlin")
```

A list may be defined as a range of values. For example, the following list includes values from 1 to 5.

```
(1 .. 5)
```

An empty list is defined as:

```
()
```

Initialization and Access

Like scalars, arrays do not have to be declared, nor is it necessary to allocate memory for them. Perl grows and shrinks arrays automatically as required.

The following example shows a number of ways an array can be defined.

```
e0401.plx
  1 #!/usr/bin/perl -w
  2 print "\nEach array contains the following values:\n";
  3
  4 @arrayA = ( 1, 4, 9, 16, 25, 36 );  # List of Numbers
  5 @arrayB = ( 1 .. 6);                # Range of values
  6 @arrayC = (13, 23, "Berlin", 34);   # Mixed values
  7 @arrayD = @arrayC;
  8
  9 print "ArrayA: @arrayA\n";
 10 print "ArrayB: @arrayB\n";
 11 print "ArrayC: @arrayC\n";
 12 print "ArrayD: @arrayD\n";

$ e0401.plx

Each array contains the following values:
ArrayA: 1 4 9 16 25 36
ArrayB: 1 2 3 4 5 6
ArrayC: 13 23 Berlin 34
ArrayD: 13 23 Berlin 34
```

First, notice that if an array is printed in a string context, each element in the array is printed separated by a space. This can be a very helpful feature. The following shows each assignment:

- Line 4 shows the assignment of numeric scalars to an array. An example like this has been shown before.
- Line 5 shows the assignment of values using a range. After the assignment, the array contains the numbers from 1 to 6.
- Line 6 shows an assignment using a mixture of numeric and string scalar values.
- Line 7 shows how one array can be copied to another.

Quoting Operator

With lists that contain a lot of strings, it is tedious to enter a long list of values. For example:

```
@arrayA = ("Berlin", "Paris", "London", "Rome");
```

So that you do not have to type all those double quotation marks and commas, a quoting operator for arrays is included in the language. For example, the above statement can be rewritten like this:

```
@arrayA = qw/Berlin Paris London Rome/;
```

However, the qw operator does not perform variable interpolation like the qq operator. Look at the following example. What do you think the output will be?

e0402.plx

```
 1 #! /usr/bin/perl -w
 2 # Demonstrates ways to declare arrays
 3 $x = "John"; $y = "Joe"; $z = "Jake";
 4 print "\n";
 5
 6 @a = ($x, $y, $z);
 7 print "a: @a\n\n";
 8
 9 @b = ('$x', '$y', '$z');
10 print "b: @b\n\n";
11
12 # Use qw
13 @c = qw/John Joe Jake/;
14 print "c: @c\n\n";
15
16 @d = qw/$x $y $z/;
17 print "d: @d\n\n";
18
19 @e = qw/"$x" "$y" "$z"/;
20 print "e: @e\n\n";
```

```
$ e0402.plx

a: John Joe Jake

b: $x $y $z

c: John Joe Jake

d: $x $y $z

e: "$x" "$y" "$z"
```

Notice that when variables are assigned without quotes on Line 6, the values are interpolated. However, when using qw on Lines 16 and 19, they are not.

Accessing Single Elements

Single elements of an array are accessed by a subscript in square brackets. The array index starts with 0, not 1. Each single element is a scalar value, so a dollar sign is used to refer to individual elements; for example, `$arr[3]`.

e0403.plx

```
1 #!/usr/bin/perl -w
2
3 @arr = ("mercury", "venus", "earth", "mars");
4
5 $x = $arr[0]; # mercury
6 $y = $arr[2]; # earth
7
8 print "$x $y\n";
```

```
$ e0403.plx
mercury earth
```

With a negative index, the array is referenced from its end. For example, `$arr[-2]` accesses the second-to-last element. The special symbol `$#` refers to the last index in an array. So, the last element can be accessed by `$arr[$#arr]`, but using `$arr[-1]` is simpler.

e0404a.plx

```
 1 #!/usr/bin/perl -w
 2
 3 @arr = ("mercury", "venus", "earth", "mars");
 4
 5 $a = $arr[-1] ;
 6 $b = $arr[-3] ;
 7
 8 $last = $#arr ;
 9
10 $c = $arr[$#arr] ;
11
12 print "$a $b $c $last\n";
```

```
$ e0404a.plx
mars venus mars 3
```

Adding Elements

To add additional elements to an array, assign the new element plus the old array to the array.

```
e0404b.plx
  1 #!/usr/bin/perl -w
  2
  3 # Adding elements
  4 @arr = ("mercury", "venus", "earth", "mars");
  5
  6 @arr = (@arr, "jupiter"); # Add to End
  7
  8 @arr = ("sun", @arr); # Add to front
  9
 10 print "\@arr = @arr\n";

$ e0404b.plx
@arr = sun mercury venus earth mars jupiter
```

Line 6 adds an element to the end of the array. Line 8 adds an element to the front of the array.

Determining the Length of an Array

The number of elements of an array is `$#arr + 1`, because the index starts at zero. Another way to get the number of elements in an array is to assign an array to a scalar variable. In this context, the array length is returned.

```
e0405.plx
  1 #!/usr/bin/perl -w
  2
  3 @arr = ("mercury", "venus", "earth", "mars");
  4
  5 $length1 = $#arr + 1; # Array length
  6
  7 $length2 = @arr; # Returns Array length
  8
  9 # Two numbers should be the same
 10 print "$length1 $length2\n";

$ e0405.plx
4 4
```

Iterations

Loops perform the same operations on all elements of an array. The `for` loop and the `foreach` loop are best suited to iterating through an array.

Because the subscript ranges from zero to `$#arr`, a typical `for` loop looks like this:

```
e0406.plx
  1 #!/usr/bin/perl -w
  2
  3 @arr = ("mercury", "venus", "earth", "mars");
  4
  5 for ($i=0; $i<=$#arr; $i++) {
  6   print "Element $i contains $arr[$i]\n";
  7 }

$ e0406.plx
Element 0 contains mercury
Element 1 contains venus
Element 2 contains earth
Element 3 contains mars
```

With a `foreach` loop, subscripts or ranges are not necessary. The following example prints out the contents of the array using `foreach`.

```
e0407a.plx
  1 #!/usr/bin/perl -w
  2
  3 @arr = ("mercury", "venus", "earth", "mars");
  4
  5 print "The array contains the following elements:\n";
  6
  7 foreach $index (@arr) {
  8   print "$index ";
  9 }
 10
 11 print "\n";

$ e0407a.plx
The array contains the following elements:
mercury venus earth mars
```

The Default Variable

Perl includes a number of special variables that Perl reserves for special purposes. The most frequently used of these variables is `$_`, the default variable. The `$_` variable is automatically created and available inside a loop. For instance, the previous example can be written as follows:

```
e0407b.plx
  1 #!/usr/bin/perl -w
  2
  3 @arr = ("mercury", "venus", "earth", "mars");
  4
  5 print "The array contains the following elements:\n";
  6
  7 for (@arr) {
  8   print "$_ "; # Note the space
  9 }
 10
 11 print "\n";
```

```
$ e0407b.plx
The array contains the following elements:
mercury venus earth mars
```

The script looks like it did before. The only difference is that `$index` is replaced by `$_`. So what?

In Perl programming, excessive and unnecessary use of parentheses and other punctuation is explicitly deprecated. The real power of the $_ variable is that it allows programmers to use shorthand in scripts. When no variable is specified in a for statement, $_ is assumed. Knowing this fact, you can save some typing and rewrite the script like this:

e0407c.plx

```
 1 #!/usr/bin/perl -w
 2
 3 @arr = ("mercury", "venus", "earth", "mars");
 4
 5 print "The array contains the following elements:\n";
 6
 7 foreach (@arr) {
 8   print;
 9 }
10
11 print "\n";
```

```
$ e0407c.plx
The array contains the following elements:
mercuryvenusearthmars
```

All references to the array are gone in the loop. Line 7 implies that $_ is the variable that will be operated on. Line 8 implies that current array element. There is no need to type out an array reference when you do not need to.

The $_ variable is a powerful part of the standard Perl idiom and something that is used often in Perl scripts.

Flat Lists

If you assign a list to an array, elements of this list may include other arrays. However, the resulting array is not multidimensional. Instead, all elements of the combined lists are integrated into one list. The result is a “flat” list, where all elements are sequential. The associations to the original sublists are lost. An example is shown as follows.

e0408.plx

```
1 #!/usr/bin/perl -w
2
3 @arra = ("mercury", "venus", "earth", "mars");
4 @arrb = ("jupiter", "saturn");
5
6 @arrc = ("sun", @arra, @arrb, "uranus");
7 print "@arrc\n";
```

```
$ e0408.plx
sun mercury venus earth mars jupiter saturn uranus
```

Slices

Subranges of an array or a list of individual elements but not all elements of an array are known as array slices. Because a slice contains more than one element, it is a list and is marked with an @. In slices, the subscripts are declared as a comma-separated list. Some examples are shown as follows:

e0409a.plx

```
 1 #!/usr/bin/perl -w
 2
 3 @arra = ("mercury", "venus", "earth", "mars");
 4
 5 @slicea = @arra[1, 3];
 6 @sliceb = @arra[0 .. 2];
 7
 8 print "a: @slicea\n";
 9 print "b: @sliceb\n";
10
11 @arra[0, 1, 3] = ("jupiter", "neptune", "pluto");
12 print "\@arra: @arra\n";
13
14 # Swap elements 1 and 3
15 @arra[1, 3] = @arra[3, 1];
16 print "\@arra: @arra\n";
```

```
$ e0409a.plx
a: venus mars
b: mercury venus earth
@arra: jupiter neptune earth pluto
@arra: jupiter pluto earth neptune
```

Line 11 demonstrates how values are easily assigned to individual elements using slices. Line 15 shows how easy it is to swap values in an array using slices.

Looking At <> In List Context

Filehandles return the list of all their contents, one line per element, when read in list context. This is a convenient feature for quickly testing a script or creating an array on the fly. The following script demonstrates this mechanism.

e0409b.plx

```
1 #!/usr/bin/perl -w
2
3 @all = <>;
4
5 print "$all[2]\n";
```

```
$ e0409b.plx e0409b.plx
@all = <>;
```

When the script is executed, the lines in <> are stored as elements in the array. In the example, the third line (2nd element) of the array is printed out. A word of caution: make sure that the size of the file you read does not exceed (or even approach) the limits of your system's memory!

The Special Array @ARGV

Perl includes a special array that contains all arguments passed on the command line, @ARGV. For example:

```
$ myscript.plx -read month.dat july
```

If the script myscript.plx is invoked with the arguments as shown previously, Perl initializes the array @ARGV like this:

```
@ARGV = qw/-read month.dat july/
```

The elements in the array are easily accessed. For example:

```
e0410.plx
  1 #!/usr/bin/perl -w
  2 # List Parameters passed in @ARGV
  3
  4 $i=0;
  5
  6 for (@ARGV) {
  7 print "Option ", ++$i,": $_\n";
  8}
```

```
$ e0410.plx -read month.dat july
Option 1: -read
Option 2: month.dat
Option 3: july
```

A popular way of accessing command-line parameters is using the shift operator. The shift operator is described in “The shift Function” on page 4-20. Here, shift gets the first element of @ARGV.

```
$opt = shift @ARGV;
```

Note – The @ARGV array is very different from 'argv' in the C programming language. $ARGV[0] contains the first command-line argument and not the program name. The program name in Perl is found in the special variable $0. "scalar @ARGV" in Perl produces the equivalent of 'argc' in C.

Multidimensional Arrays

In Perl, multidimensional arrays are created with references. The array itself is a flat list that represents the outermost dimension. But instead of ordinary values, its elements contain references (pointers) to further arrays that represent inner dimensions. With two dimensions, the latter arrays contain the proper values.

Multidimensional arrays are prefixed like regular arrays by an @. At initialization, the elements of the inner dimensions are put in square brackets. (The list reference operator in Perl.)

```
@marr = ( [11,12,13] , [14,15,16] ) ;
# a two-dimensional array of 2 x 3 elements
```

To address single elements, a notation similar to the notation used with simple arrays is used, but two subscripts are needed. The first one indicates the outer dimension, and the second one indicates the inner dimension.

```
$x = $marr[1][2] ; #  16
```

Again, loops can access all elements of the array. Because there is more than one dimension, multiple nested loops are used.

e0411.plx

```
 1 #!/usr/bin/perl -w
 2
 3 # a two-dimensional array of 2 x 3 elements
 4 @marr = ( [11,12,13], [14,15,16]);
 5
 6 for ($i=0 ; $i<=1 ; $i++) { # row
 7
 8    for ($j=0 ; $j<= 2 ; $j++) { # column
 9       print "$i,$j: $marr[$i][$j]  ";
10    }
11
12    print "\n" ;
13 }
```

```
$ e0411.plx
0,0: 11  0,1: 12  0,2: 13
1,0: 14  1,1: 15  1,2: 16
```

Exercise: Create and Manipulate Arrays

In this exercise, you complete the following tasks:

- Create an array variable and assign scalar values to the array
- Determine the length of an array using an array operator
- Use array slices to assign new values to an array
- Determine the length of an array using a scalar variable

Tasks

Complete the following steps:

1. Given the following array definition, perform the listed operations.

```
@arr1 = qw/mercury venus earth/;
```

   - Print the list separated by spaces.
   - Print the length of the list using a scalar variable.
   - Reverse the list using array slices.
   - Print the list separated by spaces.
   - Add “mars” to the front of the array.
   - Print out the array again.
   - Print its length again.

   Sample output:

```
Array 1: mercury venus earth
Length: 3
Reversed: earth venus mercury
Array 1: mars earth venus mercury
Length: 4
```

2. Write a script that gets a list of numbers that is ended by pressing Control-D. Store the list in an array. Print the array, the sum of the numbers, and the average of the numbers.

   Sample output:

```
Enter a list of numbers: 75
84
88
92
98
<Ctrl-D>
Numbers entered: 75 84 88 92 98
The total is: 437
The average is: 87.4
```

3. Given the following array definition, count the number of a's, b's, and c's in the array, the total number of other characters, and the total number of characters in the array. Also display the last character in the list.

```
@letterlist = qw/a b c d e f g a b c d e/;
```

   Hint: Use an array to count the a's, b's, c's and other characters. Use a loop around a switch construct to increment the counters.

   Sample output:

```
For this list: a b c d e f g a b c d e
2 a's.
2 b's.
2 c's.
6 others.
12 characters in the array.
Last character is: e
```

Array Functions

There are many array and list operators and functions in Perl. These functions are explained in the pages that follow.

The `shift` Function

This function removes the first element of `@arr` and returns it. All remaining elements are moved left by one position, so that the array starts again at zero. If the array is not specified, "`shift`" operates on the special variable `@ARGV`.

e0412.plx

```
1 #!/usr/bin/perl -w
2
3 @planet = qw/mercury venus earth mars/;
4
5 $first = shift @planet;
6
7 print "$first was removed\n";
8 print "\@planet = @planet\n";
```

```
$ e0412.plx
mercury was removed
@planet = venus earth mars
```

The unshift Function

This function prepends one or more elements to the front of an array. All previous elements are moved right, and the array grows. The number of elements in the new array is returned.

e0413.plx

```
1 #!/usr/bin/perl -w
2
3 @planet = qw/mercury venus earth mars/;
4
5 # Add planetx to the front
6 $count = unshift @planet, "planetx";
7
8 print "\@planet contains: @planet\n";
9 print "There are now $count elements\n";
```

```
$ e0413.plx
@planet contains: planetx mercury venus earth mars
There are now 5 elements
```

The pop Function

This function removes the last element of @arr and returns it. The length of the array is reduced by 1.

e0414.plx

```
1 #!/usr/bin/perl -w
2
3 @planet = qw/mercury venus earth mars/;
4
5 $last = pop @planet;
6
7 print "$last was removed\n";
8 print "\@planet = @planet\n";
```

```
$ e0414.plx
mars was removed
@planet = mercury venus earth
```

The push Function

This function pushes one or more elements onto the end of @arr, and the array grows. It returns the new length of the array.

e0415.plx

```
1 #!/usr/bin/perl -w
2
3 @planet = qw/mercury venus earth mars/;
4
5 $newlen = push @planet, "jupiter";
6
7 print "\@planet = @planet\n";
8 print "\@planet length = $newlen\n";
```

```
$ e0415.plx
@planet = mercury venus earth mars jupiter
@planet length = 5
```

The splice Function

This function removes length elements of the array, starting at startpos. The removed sublist is returned. If length is omitted, all elements from startpos through the last element are removed. It is also possible to specify a list subst, which replaces the removed part. The array can grow or shrink respectively. The splice operator is usually saved for jobs that cannot be done by push, pop, and so on.

e0416.plx

```
 1 #!/usr/bin/perl -w
 2
 3 @planet = qw/mercury venus earth mars/;
 4
 5 # Remove 0 - 1
 6 @hot = splice @planet, 0, 2;
 7 print "\@planet = @planet\n";
 8 print "Line 8: \@hot = @hot\n";
 9
10 # Swap venus for mars
11 splice @hot, 1, 1, "mars";
12 print "Line 12: \@hot = @hot\n";
13
14 # Add venus and earth at position 1
15 splice @hot, 1, 0, qw/venus earth/;
16 print "Line 16: \@hot = @hot\n";
```

```
$ e0416.plx
@planet = earth mars
Line 8: @hot = mercury venus
Line 12: @hot = mercury mars
Line 16: @hot = mercury venus earth mars
```

The reverse Function

The reverse operator was first introduced with string operators. In list context, reverse returns the given list in reverse order.

e0417.plx

```
  1 #!/usr/bin/perl -w
  2
  3 @planet = qw/mercury venus earth mars/;
  4
  5 # Reverse @planet
  6 @rev = reverse @planet;
  7 print "\@rev = @rev\n";
  8
```

```
$ e0417.plx
@rev = mars earth venus mercury
```

The print Function

In the previous examples, the print operator is used within double quotation marks. This prints each element in the list separated by a space. If print is used outside of double quotation marks in a list context, all elements of the list are printed successively without any separator.

```
e0418a.plx
  1 #!/usr/bin/perl -w
  2
  3 @planet = qw/mercury venus earth mars/;
  4
  5 print "\@planet = @planet\n";
  6 print "\@planet = ",@planet, "\n";
```

```
$ e0418a.plx
@planet = mercury venus earth mars
@planet = mercuryvenusearthmars
```

You can change what is printed between each element of the array by modifying the $" special variable. For example, the following script prints "--" between each element.

```
e0418b.plx
  1 #!/usr/bin/perl -w
  2
  3 @planet = qw/mercury venus earth mars/;
  4
  5 $" = "--";
  6
  7 print "\@planet = @planet\n";
```

```
$ e0418b.plx
@planet = mercury--venus--earth--mars
```

The split Function

The split function splits a string into separate substrings by scanning it for a separator. The separator is specified within slashes. It can be an ordinary character or string or a complex regular expression. (Regular expressions are described in Module 6, "Basic I/O and Regular Expressions.") The split operator returns the isolated substrings as a list.

```
e0419a.plx
  1 #!/usr/bin/perl -w
  2
  3 #Split String on colons
  4 $passwd = "alf:x:86:30:Alfons:/export/home/alf:/bin/ksh";
  5
  6 @passwd = split /:/, $passwd;
  7
  8 for ($x = 0; $x <= $#passwd; $x++){
  9   print "Element[$x]: $passwd[$x]\n";
 10 }
```

```
$ e0419a.plx
Element[0]: alf
Element[1]: x
Element[2]: 86
Element[3]: 30
Element[4]: Alfons
Element[5]: /export/home/alf
Element[6]: /bin/ksh
```

Line 6 shows the standard split behavior. A single character is removed between words, and each word becomes an array element.

The next example splits on a space.

```
e0419b.plx
  1 #!/usr/bin/perl -w
  2
  3 # Split on a space
  4 $line = "It's time to say goodbye";
  5
  6 @words = split / /, $line;
  7
  8 for ($x = 0; $x <= $#words; $x++){
  9
 10   print "Element[$x]: $words[$x]\n";
 11 }
```

```
$ e0419b.plx
Element[0]: It's
Element[1]: time
Element[2]: to
Element[3]: say
Element[4]: goodbye
```

As a final example, if you split with no character between //, each character in the string is split out as an array element.

```
e0419c.plx
  1 #!/usr/bin/perl -w
  2
  3 # Split word apart
  4 $word = "Gandalf" ;
  5
  6 @chars = split //, $word;
  7
  8 for ($x = 0; $x <= $#chars; $x++){
  9
 10   print "Element[$x]: $chars[$x]\n";
 11 }
```

```
$ e0419c.plx
Element[0]: G
Element[1]: a
Element[2]: n
Element[3]: d
Element[4]: a
Element[5]: l
Element[6]: f
```

Splitting With ' '

In addition to //, another character performs a special function when used with split. A single-quoted space, ' ', automatically removes leading spaces from a string. To illustrate this concept, look at the following examples.

e0420a.plx

```
 1 #! /usr/bin/perl -w
 2
 3 # Split using / /
 4 $string=" a   b c";
 5
 6 print "Split '$string'\n";
 7
 8 @letters = split / /, $string;
 9
10 for ($x = 0; $x <= $#letters; $x++) {
11
12   print "Element[$x] = $letters[$x]\n";
13 }
```

```
$ e0420a.plx
Split " a   b c"
Element[0] =
Element[1] = a
Element[2] =
Element[3] =
Element[4] = b
Element[5] = c
```

In this example, leading spaces are not removed but are considered delimiters between null elements.

With a space and single quotes, the result is different.

e0420b.plx

```
  1 #! /usr/bin/perl -w
  2
  3 # Split using ' '
  4 $string=" a   b c";
  5
  6 print "Split '$string'\n";
  7
  8 @letters = split ' ', $string;
  9
 10 for ($x = 0; $x <= $#letters; $x++) {
 11
 12   print "Element[$x] = $letters[$x]\n";
 13 }
```

```
$ e0420b.plx
Split " a   b c"
Element[0] = a
Element[1] = b
Element[2] = c
```

Any white space is treated as a single delimiter and only non-white space becomes elements in the array.

The split Function

To split the first few fields from a string, a limit is specified that indicates the number of substrings split off. The rest of the string is returned as the last field. Because split is more elegant than a loop around substr commands and consecutive index calls, it is very popular.

e0421.plx

```
  1 #!/usr/bin/perl -w
  2
  3 #Split String on :
  4 $passwd = "alf:x:86:30:Alfons:/export/home/alf:/bin/ksh";
  5
  6 @passwd = split /:/, $passwd, 4;
  7
  8 for ($x = 0; $x <= $#passwd; $x++){
  9   print "Element[$x]: $passwd[$x]\n";
 10 }
```

```
$ e0421.plx
Element[0]: alf
Element[1]: x
Element[2]: 86
Element[3]: 30:Alfons:/export/home/alf:/bin/ksh
```

The join Function

The join function is complementary to the split function. It concatenates the elements of a list using a specified character or string as a separator. The resulting string is returned.

e0422.plx

```
 1 #!/usr/bin/perl -w
 2
 3 # Join sentence using two colons ::
 4
 5 $line = "It's time to say goodbye";
 6
 7 @words = split / /, $line;
 8
 9 for ($x = 0; $x <= $#words; $x++){
10
11   print "Element[$x]: $words[$x]\n";
12 }
13
14 $line = join "::", @words; # Join @words
15
16 print "\$line now is: $line\n"
```

```
$ e0422.plx
Element[0]: It's
Element[1]: time
Element[2]: to
Element[3]: say
Element[4]: goodbye
$line now is: It's::time::to::say::goodbye
```

The sort Function

Perl offers a very powerful sort function. By default, Perl sorts in ASCII order. The sorted list is returned. The original list remains unchanged.

```
e0423.plx
  1 #!/usr/bin/perl -w
  2
  3 # Default Sort
  4 @arr = qw/bb aa dd cc 1 12 2 23 4/;
  5
  6 @sortedarr = sort @arr;
  7
  8 print "The sorted list is: @sortedarr\n";
```

```
$ e0423.plx
The sorted list is: 1 12 2 23 4 aa bb cc dd
```

What if you wanted to sort in reverse order? What about numeric sorting?

String Sorting

To explicitly sort in alphabetical order, which is the default sort method, use the comparison operator cmp.

```
e0424.plx
  1 #!/usr/bin/perl -w
  2
  3 # String Sort (Default)
  4 @arr = qw/bb aa dd cc 1 12 2 23 4/;
  5
  6 @sarr = sort { $a cmp $b } @arr;
  7
  8 print "The sorted list is: @sarr\n";
```

```
$ e0424.plx
The sorted list is: 1 12 2 23 4 aa bb cc dd
```

To sort in descending order, one change is needed, as shown in the following script.

e0425.plx
```
  1 #!/usr/bin/perl -w
  2
  3 # Descending string sort
  4 @arr = qw/bb aa dd cc 1 12 2 23 4/;
  5
  6 @sortedarr = sort { $b cmp $a } @arr;
  7
  8 print "The sorted list is: @sortedarr\n";
```

```
$ e0425.plx
The sorted list is: dd cc bb aa 4 23 2 12 1
```

Numeric Sorting

To sort numbers in ascending order, use the spaceship operator <=>.

e0426.plx
```
  1 #!/usr/bin/perl -w
  2
  3 # Numeric Sort, Ascending
  4 @arr = qw/1 12 2 23 4/;
  5
  6 @sarr = sort { $a <=> $b } @arr;
  7
  8 print "The sorted list is: @sarr\n";
```

```
$ e0426.plx
The sorted list is: 1 2 4 12 23
```

Sorting Numbers and Strings

What if you want to sort strings and numbers? To do this, make one small change to the `sort` operator.

```
e0427.plx
  1 #!/usr/bin/perl -w
  2
  3 # Sort numbers and strings!
  4 @arr = qw/bb aa dd cc 1 12 2 23 4/;
  5
  6 @sortedarr = sort { $a <=> $b || $a cmp $b } @arr ;
  7
  8 print "The sorted list is: @sortedarr\n";

$ e0427.plx
The sorted list is: aa bb cc dd 1 2 4 12 23
```

How does this work? When two values are identical, the spaceship operator returns a 0. This causes the right side of the statement to be executed. The values are then sorted using the string `cmp` operator.

Note – This is a good example of a situation where "`-w`" becomes very useful. The function above appears to work fine, yet "`-w`" generates errors when `<=>'` is used to test the letters. Why? Because the construction is a dubious one. It fails in degenerate cases (such as the inclusion of negative numbers) and needs to be rewritten correctly. Turning off "`-w`" because it generates errors is like turning off the fire alarm because your smoking causes it to ring!

The grep Function

The grep function performs an implicit loop over an array or a list and returns only the elements which match the specified expression. Each element in turn is contained in $_ during the iteration. The script below returns any element that begins with the letter "a." The output prints out the number of elements found and the elements.

e0429.plx

```
  1 #!/usr/bin/perl -w
  2
  3 @arr = qw/ab abc abcd bc c/;
  4
  5 @garr = grep /^a/, @arr;
  6 $count = @garr;
  7
  8 print "$count matches were found\n";
  9 print "Words that start with a: @garr\n";
```

```
$ e0429.plx
3 matches were found
Words that start with a: ab abc abcd
```

Back Quotes and Command Execution

One of the powerful features of Perl is its ability to interact with the host operating system. System commands are executed by enclosing the command in back quotes. The output can be stored in a string or array and manipulated programatically with Perl. For example, you can process the output of the `ls` command to create the exact output you want.

e0430.plx

```
  1 #!/usr/bin/perl -w
  2
  3 # A Perl ls command
  4
  5 @files = `/usr/bin/ls`;
  6 chomp @files;
  7
  8 $count = @files;
  9
 10 print "@files\n";
 11 print "$count files listed\n"
```

```
$ e0430.plx
examples labs
2 files listed
```

This script reads the output from the `ls` command and stores the results in an array. The variable `$count` stores the length of the array and print the number of files or directories stored in the directory.

Something a bit more advanced can be created using some of the operators described previously. For example, the following script uses the `ls -l` command to count the number of files and total number of bytes in a directory.

e0431.plx

```
 1 #!/usr/bin/perl -w
 2
 3 # Advanced Perl ls
 4 $bytes = 0;
 5
 6 @files = `ls -l`;
 7 shift @files; # Remove first line of ls
 8
 9 $count = @files;
10
11 # Bytes are 4th element, Name 8th
12 foreach $fileline (@files){
13
14   @line = split ' ', $fileline; # Split Line
15   $bytes = $bytes + $line[4]; # Sum bytes
16   push(@fileout,$line[8]); # Add filename to fileout
17 }
18
19 print "@fileout\n";
20 print "$count files or directories\n";
21 print "$bytes bytes\n";
```

```
$ e0431.plx
e0431.plx examples labs
3 files or directories
1959 bytes
```

Using the various array operators, there are a number of Perl command-line utilities like this one that you can write.

The qx Operator

As an alternative to backquotes, the qx operator can be used to execute a command. This operator works just like the qq and q operators described in “Quoting Operators” on page 2-9, except the text included in its delimiters is a command. The script that follows demonstrates the use of the qx operator. The script executes the uptime command and displays the results in a different way.

e0432.plx
```
 1 #!/usr/bin/perl -w
 2
 3 # split uptime
 4 $upstats = qx/uptime/;
 5
 6 print "Command Output: $upstats\n";
 7
 8 @temp = split /,/, $upstats; # Remove commas
 9 $tempstr = join " ", @temp; # Rejoin with spaces
10
11 @up = split ' ', $tempstr; # Remove all spaces
12
13 print "==Perl uptime==\n";
14 print "Up Time: $up[2]\n";
15 print "# of Users: $up[5]\n";
16 print "Load average: $up[9] $up[10] $up[11]\n" ;
```

```
$ e0432.plx
Command Output:   1:19pm  up  4:35,  20 users,  load
average: 0.32, 0.43, 0.52

==Perl uptime==
Up Time: 4:35
# of Users: 20
Load average: 0.32 0.43 0.52
```

Removing specific parts of a string can be accomplished more easily with regular expressions (which are described in Module 6, “Basic I/O and Regular Expressions”). However, this brute force method, using split and join, works as well.

Exercise: Create Scripts Using Array Functions

In this exercise, you complete the following tasks:

- Reverse the contents of an array
- Use `pop` to modify the contents of an array
- Sort an array
- Create an array from scalar using `split`
- Process the values passed in the command-line array, `@ARGV`
- Read the output of a UNIX command into an array

Tasks

Complete the following steps:

4. Write a script that allows the user to enter a list of numbers. Then, print the array in reverse order using `reverse`.

   Sample output:

   ```
   Enter a list of numbers: 5
   45
   65
   2
   1
   The array in order is: 5 45 65 2 1
   The array in reverse order: 1 2 65 45 5
   ```

5. (Optional) Modify the previous script to print the list in reverse order using `pop`.

6. Modify the previous script to print the array sorted in ascending order and in descending order.

   Sample output:

```
Enter a list of numbers: 5
45
65
2
1
Original list: 5 45 65 2 1
Ascending sort: 1 2 5 45 65
Descending sort: 65 45 5 2 1
```

7. Modify the previous script so that each element is separated by three dashes. Use `join` to produce the output.

   Sample output:

```
Enter a list of numbers: 5
45
65
2
1
Original list: 5---45---65---2---1
Ascending sort: 1---2---5---45---65
Descending sort: 65---45---5---2---1
```

8. Write script that will read in a line of text. Split the line on spaces, and print each element of the resulting array.

   Sample output:

```
Enter a sentence: Live long and prosper
$arr1[0] = Live
$arr1[1] = long
$arr1[2] = and
$arr1[3] = prosper
```

9. (Optional) Write a script that reads in a list of hosts passed on the command line and then display the number of users logged into each host.

   Use the command `finger @host` to obtain a list of the users logged into the host. Remove the first line or two from the command so that the lines are not included in your count.

   Sample Output:

```
$ lab0409.plx host1 host2
The user load for each server is:
host1 has 7 users
host2 has 2 users
```

Exercise Summary

Discussion – Take a few minutes to discuss the experiences, issues, or discoveries you had during the lab exercises.

- Experiences
- Interpretations
- Conclusions
- Applications

Exercise Solutions

1. Given the following array definition, perform the listed operations.

```
@arr1 = qw(mercury venus earth);
```

   - Print the list separated by spaces.
   - Print the length of the list using a scalar variable.
   - Reverse the list using array slices.
   - Print the list separated by spaces.
   - Add “mars” to the front of the array.
   - Print out the array again.
   - Print its length again.

   Sample output:

```
Array 1: mercury venus earth
Length: 3
Reversed: earth venus mercury
Array 1: mars earth venus mercury
Length: 4
```

Suggested solution:

lab0401.plx

```
 1 #!/usr/bin/perl -w
 2
 3 @arr1 = qw/mercury venus earth/;
 4
 5 # Print Arr1
 6 print "Array 1: @arr1\n";
 7
 8 #print length
 9 $length = @arr1;
10 print "Length: $length\n";
11
12 # Reverse with slices
13 @arr1[0, 1, 2] = @arr1[2, 1, 0];
14 print "Reversed: @arr1\n";
15
16 # Add mars
17 @arr1 = ("mars", @arr1);
18 print "Array 1: @arr1","\n";
19
20 #print length
21 $length = $#arr1 + 1;
22 print "Length: $length\n";
```

2. Write a script that gets a list of numbers that is ended by pressing Control-D. Store the list into an array. Print the array, the sum of the numbers, and the average of the numbers.

   Sample output:

```
Enter a list of numbers: 75
84
88
92
98
<Ctrl-D>
Numbers entered: 75 84 88 92 98
The total is: 437
The average is: 87.4
```

   Suggested solution:

```
lab0402.plx
  1 #!/usr/bin/perl -w
  2
  3 # Sum a list of numbers
  4 print "Enter a list of numbers: ";
  5 chomp(@arr1 = <STDIN>);
  6
  7 $length = @arr1; # Get length
  8
  9 for (@arr1){ # Compute sum
 10      $sum += $_;
 11 }
 12
 13 print "Numbers entered: @arr1\n";
 14 print "The total is: $sum\n";
 15 print "The average is: ",($sum / $length),"\n";
```

3. Given the following array definition, count the number of a's, b's, and c's in the array, the total number of other characters, and the total number of characters in the array. Also display the last character in the list.

```
@letterlist = qw/a b c d e f g a b c d e/;
```

Hint: Use an array to count the a's, b's, c's and other characters. Use a loop around a switch construct to increment the counters.

Sample output:

```
For this list: a b c d e f g a b c d e
2 a's.
2 b's.
2 c's.
6 others.
12 characters in the array.
Last character is: e
```

Suggested solution:

lab0403.plx

```
  1 #!/usr/bin/perl -w
  2
  3 @letterlist = qw/a b c d e f g a b c d e/;
  4 $len = @letterlist;
  5
  6 for (@letterlist){
  7 SWITCH:{
  8
  9 if ($_ eq "a") { $counter[0]++; last SWITCH;};
 10 if ($_ eq "b") { $counter[1]++; last SWITCH;};
 11 if ($_ eq "c") { $counter[2]++; last SWITCH;};
 12 $counter[3]++;
 13
 14 } # End switch
 15 } # end foreach
 16
 17 print "For this list: @letterlist\n";
 18 print "$counter[0] a's.\n";
 19 print "$counter[1] b's.\n";
 20 print "$counter[2] c's.\n";
 21 print "$counter[3] others.\n";
 22 print "$len characters in the array.\n";
 23 print "Last character is: ",($letterlist[$len -
1]),"\n";
```

4. Write a script that allows the user to enter a list of numbers. Then, print the array in reverse order using `reverse`.

   Sample output:

```
Enter a list of numbers: 5
45
65
2
1
<ctrl-d>
The array in order is: 5 45 65 2 1
The array in reverse order: 1 2 65 45 5
```

Suggested solution:

lab0404.plx

```
 1 #!/usr/bin/perl -w
 2
 3 # Reverse the array
 4
 5 print "Enter a list of numbers: ";
 6 chomp(@input=<STDIN>);
 7
 8 print "The array in order is: @input\n";
 9
10 @input = reverse @input; # Reverse Array
11
12 print "The array in reverse order: @input\n";
```

5. (Optional) Modify the previous script to print the list in reverse order using `pop`.

Suggested solution:

```
lab0405.plx
  1 #!/usr/bin/perl -w
  2
  3 # Reverse the array
  4 print "Enter a list of numbers: ";
  5 chomp(@input=<STDIN>);
  6
  7 @copy = @input;
  8
  9 print "The array in order is: @input\n";
 10
 11 print "The array in reverse order: ";
 12
 13 print pop @copy, " " while @copy > 0;
 14 print "\n";
```

6. Modify the previous script to print the array sorted in ascending order and in descending order. Sample output:

```
Enter a list of numbers: 5
45
65
2
1
<ctrl-d>
Original list: 5 45 65 2 1
Ascending sort: 1 2 5 45 65
Descending sort: 65 45 5 2 1
```

Suggested solution:

```
lab0406.plx
  1 #!/usr/bin/perl -w
  2
  3 # Sort ascending and descending
  4 print "Enter a list of numbers: ";
  5 chomp(@input=<STDIN>);
  6
  7 @asort = sort {$a <=> $b} @input;
  8 @dsort = sort {$b <=> $a} @input;
  9
 10 print "Original list: @input\n";
 11 print "Ascending sort: @asort\n";
 12 print "Descending sort: @dsort\n";
```

7. Modify the previous script so that each element is separated by three dashes. Use `join` to produce the output.

   Sample output:

```
Enter a list of numbers: 5
45
65
2
1
<ctrl-d>
Original list: 5---45---65---2---1
Ascending sort: 1---2---5---45---65
Descending sort: 65---45---5---2---1
```

   Suggested solution:

lab0407.plx

```
 1 #!/usr/bin/perl -w
 2
 3 # Sort ascending and descending
 4 print "Enter a list of numbers: ";
 5 chomp(@input=<STDIN>);
 6
 7 @asort = sort {$a <=> $b} @input;
 8 @dsort = sort {$b <=> $a} @input;
 9
10 $istr = join "---", @input;
11 $astr = join "---", @asort;
12 $dstr = join "---", @dsort;
13
14 print "Original list: $istr\n";
15 print "Ascending sort: $astr\n";
16 print "Descending sort: $dstr\n";
```

8. Write a script that will read in a line of text. Split the line on spaces, and print each element of the resulting array.

   Sample output:

```
Enter a sentence: Live long and prosper
$arr1[0] = Live
$arr1[1] = long
$arr1[2] = and
$arr1[3] = prosper
```

   Suggested solution:

```
lab0408.plx
  1 #!/usr/bin/perl -w
  2
  3 # Split a Sentence
  4 print "Enter a sentence: ";
  5 chomp($line = <STDIN>);
  6
  7 @arr1 = split ' ', $line;
  8
  9 for ($x = 0; $x <= $#arr1; $x++) {
 10   print "\$arr1[$x] = $arr1[$x]\n";
 11 }
```

9. (Optional) Write a script that reads in a list of hosts passed on the command line and then displays the number of users logged into each host.

   Use the command `finger @host` to obtain a list of the users logged into the host. Remove the first two lines from the output so that the lines are not included in your count.

   Sample output:

```
$ lab0409.plx host1 host2
The user load for each server is:
host1 has 7 users
host2 has 2 users
```

Suggested solution:

lab0409.plx

```
 1 #!/usr/bin/perl -w
 2
 3 # Get approximate user count using finger
 4
 5 print "The user load for each server is:\n";
 6
 7 foreach $host (@ARGV){
 8
 9   @users = `finger \@$host`; # finger hostname
10
11   shift @users; # Remove hostname
12   shift @users; # Remove title line
13
14   $count = @users; # Count users
15
16   print $host, " has ", $count, " users\n";
17
18 }
```

Hashes

Objectives

Upon completion of this module, you should be able to:

- Print a hash using a `foreach` statement
- Access, add, and delete hash keys and values
- Print a hash using a `while` loop and the `each` function
- Determine the number of occurrences of a string in an array using a hash
- Determine if a key and value exist in a hash

Relevance

Discussion – The following questions is relevant to understanding hashes:

How do you find information in a book?

Introducing Hashes

Perl offers a second type of variable that holds multiple scalar values: hashes. Unlike arrays, the data of a hash is organized as pairs of keys and values. Hashes are prefixed with a percent sign: `%hash`. Like scalars and arrays, hashes have their own namespace.

Note – UNIX shell programmers might be familiar with the concept of hashes from `awk`, where they are simply called "arrays" (`awk` does not have a construct like a Perl array; all arrays in `awk` are associative.)

Initialization and Access

Hashes are initialized by assigning values to keys. The assignment is made using the following format:

```
key => value
```

Each key/value pair is separated by the => operator. The example on Line 4 uses this method.

```
e0501.plx
  1 #!/usr/bin/perl -w
  2
  3 # Define using => Operator
  4 %fflint = ( "name" => "flintstone",
  5             "fname" => "fred",
  6             "job" => "stonecutter") ;
  7
  8 # Define using array format
  9 %fflint = ( "name", "flintstone",
 10             "fname", "fred",
 11             "job", "stonecutter");
```

Line 9 demonstrates the alternative for defining a hash. Keys and values are separated by commas. However, using the => operator to define hashes is preferable because it increases readability and prevents the key from being accidentally interpolated.

Note – Hash keys should never contain white space, although it is possible to force Perl to accept them. This approach may interfere with the way hash lookups are performed, and can cause a number of unexpected problems. Also, note that quoting a key when accessing a value forces the evaluation of the key string, which slows down the lookup and is unnecessary unless you explicitly need that functionality.

Accessing Single Elements

Accessing a hash value is similar to obtaining an array element. The following example shows how the values can be printed out for the `%fflint` hash.

```
e0502.plx
  1 #!/usr/bin/perl -w
  2
  3 # Define using => Operator
  4 %fflint = ( "name" => "flintstone",
  5             "fname" => "fred",
  6             "job" => "stonecutter");
  7
  8 # Print Elements
  9 print qq/name: $fflint{name}\n/;
 10 print "fname: $fflint{fname}\n";
 11 print "job: $fflint{job}\n";

$ e0502.plx
name: flintstone
fname: fred
job: stonecutter
```

A special hash algorithm allows Perl to find the value of a key extremely quickly. Instead of searching the whole list, the memory position of the value is directly calculated from the key.

Uses for Hashes

Hashes are mainly used in the following situations:

- Database access by unequivocal keys; for example, whole address information by name as key
- Translation tables; for example, name of the month to digit of the month
- Frequency counts; for example, lists of words and statistics
- Complex data structures; for example, address databases as a list of address hashes

Hash Functions

There are a number of different functions that can be used with hashes. These functions are described in the following sections.

The keys Function

In a list context, the keys function returns the list of all the keys in a hash. The output of the keys function is either assigned to an array or directly used in a loop.

e0503.plx
```
  1 #!/usr/bin/perl -w
  2
  3 #Define %fflint
  4 %fflint = ( "name" => "flintstone",
  5             "fname" => "fred",
  6             "job" => "stonecutter");
  7
  8 @k = keys %fflint; # Return Keys
  9 $k = keys %fflint; # Return Number of Keys
 10
 11 print "Keys = @k\n";
 12 print "There are $k keys\n";
```

```
$ e0503.plx
Keys = name job fname
There are 3 keys
```

In scalar context, keys returns the number of pairs in the hash. As mentioned before, the keys are in a random order that depends on the memory positions of the values.

The values Function

The values function yields all the values of a hash as a list.

```
e0504.plx
  1 #!/usr/bin/perl -w
  2
  3 %fflint = ( "name" => "flintstone",
  4             "fname" => "fred",
  5             "job" => "stonecutter");
  6
  7 @v = values %fflint; # Return values
  8 $v = values %fflint; # Return number of values
  9
 10 print "Values = @v\n";
 11 print "There are $v values in the hash\n";
```

```
$ e0504.plx
Values = flintstone stonecutter fred
There are 3 values in the hash
```

In a scalar context, the values function returns the total number of values in the hash.

The each Hash Function

The each operator returns one key and value in every call. This pair is then assigned to a list of two variables ($k, $v).

```
e0505.plx
  1 #!/usr/bin/perl -w
  2
  3 %fflint = ( "name" => "flintstone",
  4             "fname" => "fred",
  5             "job" => "stonecutter");
  6
  7 ($k, $v) = each %fflint; # Return 1 key and value
  8
  9 print "The first key/value pair is: $k / $v.\n";
```

The output of this script varies depending on the order in which Perl stores the hash. Because of this, the each operator is useful only in an iterative function (a while loop).

Iteration

When the hash was printed out in the `e0502.plx` example in "Accessing Single Elements" on page 5-5, a statement was needed to display each key/value pair in the hash. A systematic way to iterate over all keys of the hash makes this task much easier. There are two alternatives for doing this, which are described in the next sections.

Using the `for` Loop With the `keys` Function

The `keys %hash` statement returns a list of all keys. If a `foreach` loop is put around this list, every element of the hash is accessed easily. For example, the hash is printed as follows using a `foreach` loop.

e0506a.plx

```
 1 #!/usr/bin/perl -w
 2
 3 %fflint = ( "name" => "flintstone",
 4             "fname" => "fred",
 5             "job" => "stonecutter") ;
 6
 7 foreach $k ( keys (%fflint) ) {
 8
 9     print "$k --> $fflint{$k}\n";
10
11 }
```

```
$ e0506a.plx
name --> flintstone
job --> stonecutter
fname --> fred
```

Or this could be written using $_ as:

```
e0506b.plx
  1 #!/usr/bin/perl -w
  2
  3 %fflint = ( "name" => "flintstone",
  4             "fname" => "fred",
  5             "job" => "stonecutter");
  6
  7 foreach ( keys (%fflint) ) {
  8
  9     print "$_ --> $fflint{$_}\n";
 10
 11 }
```

Using the while Loop With the each Function

The second way to iterate over the hash is to use the each operator. A while loop catches all the elements. As soon as the hash is completely iterated, each returns undef, and the loop is terminated.

```
e0507.plx
  1 #!/usr/bin/perl -w
  2
  3 %fflint = ( "name" => "flintstone",
  4             "fname" => "fred",
  5             "job" => "stonecutter");
  6
  7 while ( ($k,$v) = each %fflint ) {
  8
  9     print "$k --> $v\n";
 10
 11 }
```

```
$ e0507.plx
name --> flintstone
job --> stonecutter
fname --> fred
```

The iterator for each hash is shared by the each, keys, and values functions. Therefore, you should only use one of these within a given loop.

Hash Sorting

To sort a hash by its keys, apply the `sort` command to the `keys` list operator.

e0508.plx

```
  1 #!/usr/bin/perl -w
  2
  3 %fflint = ( "name" => "flintstone",
  4             "fname" => "fred",
  5             "job" => "stonecutter");
  6
  7 foreach $key (sort keys %fflint){
  8
  9     print "$key => $fflint{$key}\n";
 10
 11 }
```

```
$ e0508.plx
fname =>  fred
job =>  stonecutter
name =>  flintstone
```

Sorting a hash by its values is more complicated. Because values can be retrieved only with keys, to print the results sorted by values, make the values keys. If all the values are unique, reverse the complete hash with the `reverse` operator. The `reverse` operator changes the values to keys and the keys to values. Now the hash can be sorted by its new keys again.

e0509.plx

```
 1 #!/usr/bin/perl -w
 2
 3 %fflint = ( "name" => "flintstone",
 4             "fname" => "fred",
 5             "job" => "stonecutter");
 6
 7 %rev = reverse %fflint; # Swap keys and values
 8 # Now swapped hash is in %rev
 9
10 foreach $element (sort keys %rev){
11
12     print "$element <-- $rev{$value}\n";
13
14 }
```

```
$ e0509.plx
flintstone <-- name
fred <-- fname
stonecutter <-- job
```

Notes

Adding, Removing, and Testing Elements

This section reviews how elements are added and removed from a hash and how they are tested.

Adding Elements

To add a new element to a hash, simply specify a new key/value pair.

```
e0511a.plx
  1 #!/usr/bin/perl -w
  2
  3 %fflint = ( "name" => "flintstone",
  4             "fname" => "fred",
  5             "job" => "stonecutter" );
  6
  7 $fflint{friend} = "barney"; # Add Barney
  8
  9 foreach $k ( keys %fflint ) {
 10
 11     print "$k --> $fflint{$k}\n";
 12
 13 }

$ e0511a.plx
friend --> barney
name --> flintstone
job --> stonecutter
fname --> fred
```

Hashes can also be created on the fly simply by defining a key-value pair. For example, this script prompts for the value for the key "job."

e0511b.plx

```
1 #!/usr/bin/perl -w
2 $key = "job";
3
4
5 print "Enter Fred's job: ";
6 chomp($fflint{$key} = <STDIN>);
7
8 print "$key --> $fflint{$key}\n";
```

```
$ e0511b.plx
Enter Fred's job: stonecutter
job --> stonecutter
```

Deleting Elements

There are two ways to delete elements:

- Assigning `undef` to a key changes its value to `undef`, but it does not delete the key/value pair.
- The `delete` operator deletes the whole key/value pair from the hash.

The following script demonstrates the difference between the two methods.

e0512.plx

```
 1 #!/usr/bin/perl -w
 2
 3 %fflint = ( "name" => "flintstone",
 4             "fname" => "fred",
 5             "job" => "stonecutter");
 6
 7 undef $fflint{name}; # Value now undef
 8
 9 delete $fflint{job}; # Job no longer in hash
10
11 foreach $k ( keys(%fflint) ) {
12
13     print "$k --> $fflint{$k}\n";
14
15 }
```

```
$ e0512.plx
name -->
fname --> fred
```

The `undef` operator can also be used to remove the entire hash from memory:

```
undef %hash;
```

It removes scalars, arrays, and subroutines from memory as well.

Testing for Existence

To check whether a key/value pair exists in a hash, use the `exists` function. The `exists` function only checks to see if the key exists. If the value for that key is undefined, `exists` still returns true. To determine whether a value is defined, use the `defined` function. This function returns true only if the value is defined. For example:

```
e0513.plx
  1 #!/usr/bin/perl -w
  2
  3 %fflint = ( "name" => "flintstone",
  4             "fname" => "fred",
  5             "job" => "stonecutter") ;
  6
  7 $fflint{friend} = "barney"; # Add barney
  8 $fflint{animal} = undef ;
  9
 10 # Values to test for
 11 @test = qw/name fname job friend animal wife child/;
 12
 13 foreach $testkey (@test) {
 14
 15   print "$testkey: ";
 16   print "exists " if exists $fflint{$testkey};
 17   print "defined" if defined $fflint{$testkey};
 18   print "\n";
 19
 20 }
```

```
$ e0513.plx
name: exists defined
fname: exists defined
job: exists defined
friend: exists defined
animal: exists
wife:
child:
```

For each `$testkey`, the script prints whether the key exists and if it has a defined value.

Hash Slices

To access a subset of elements in a hash, use hash slices. Because a slice is a list, the prefix (sigil) has to be an @.

e0514.plx

```
 1 #!/usr/bin/perl -w
 2
 3 %fflint = ( "name" => "flintstone",
 4             "fname" => "fred",
 5             "job" => "stonecutter");
 6
 7 # Use hash slice to replace fred with wilma
 8 @fflint{"fname", "job"} = ("wilma", "secretary");
 9
10 foreach $k ( keys (%fflint) ) {
11
12     print "$k --> $fflint{$k}\n";
13
14 }
```

```
$ e0514.plx
name --> flintstone
job --> secretary
fname --> wilma
```

Program Environment: %ENV

The special hash %ENV provides access to system environment variables. The names of the environmental variables serve as hash keys and the variable values are stored as hash values. For example, the value of the variable $PATH is obtained through this statement:

```
$path = $ENV{PATH} ;
```

The following script demonstrates how to view all the environment variables.

```
e0515.plx
  1 #!/usr/bin/perl -w
  2
  3 foreach $var (sort keys %ENV) {
  4
  5   print "$var: $ENV{$var}\n" ;
  6
  7 }
```

```
$ e0515.plx
ATHENAHOME: /home/athena
ATHENA_BIN: /home/athena/bin
AUDIODEV: /tmp/SUNWut/dev/utaudio/98
CUE: /usr/dist/pkgs/cue
CUE_ARCHTYPE: sparc
CUE_HOSTNAME: sunray6
etc...
```

Frequency Counts

Frequency counts are a popular use for hashes. The example that follows performs a frequency count on the `last` command. The `last` command contains an entry for each time a user logs in to or out of a system. This script counts the total logins and logouts for each user.

e0516.plx
```
  1 #!/usr/bin/perl -w
  2
  3
  4 @logs = qx/last/; # Get output of last command
  5
  6 for (@logs) {
  7
  8   @temp = split; # The first field will be in $temp[0]
  9   next unless @temp;                    # Skip Blank lines
 10   $users{$temp[0]}++; # Add +1 for each user name found
 11
 12 }
 13
 14 for (sort keys %users) { # Print if 5 or > occurrences
 15
 16   print "$_ occurs $users{$_} times\n" if $users{$_} > 4;
 17
 18 }
```
```
$ e0516.plx
jl90a010 occurs 12 times
jny111st occurs 5 times
ksr123ka occurs 43 times
root occurs 6 times
sgd34743 occurs 5 times
sm12as33 occurs 5 times
th29df43 occurs 5 times
```

The script reads the output of `last` into `@logs`. The first field (the user name) is split from each line. If the line is not blank, the user name is used as the hash key, and the increment operator is used to increase the value for that key by 1.

After the count is complete, a `foreach` loop prints any key that has a value greater than 4. The seven keys that match this criterion are shown in the output.

Exercise: Create Scripts Using Hashes

In this exercise, you complete the following tasks:

- Print a hash using a `foreach` statement
- Access, add, and delete hash keys and values
- Print a hash using a `while` loop and the `each` function
- Determine the number of occurrences of a string in an array using a hash
- Determine if a key and value exist in a hash

Tasks

Complete the following steps:

1. In this exercise, use a grocery list to define a hash using the following definition. Then create a script to perform the following hash operations.

```
%hash1 = ("Apples" => 2,
          "Oranges" => 4,
          "Grapes" => 1,
          "Pears" => 4);
```

   - Print out the hash.
   - Add a key for Grapefruit, and assign it a value of 2. Print out the list sorted by its keys.
   - Delete Grapes, and print out the list using the each function.

Sample output:

```
The initial values in the hash are:
Grapes = 1
Pears = 4
Apples = 2
Oranges = 4
== Added new Fruit==
Apples = 2
Grapefruit = 2
Grapes = 1
Oranges = 4
Pears = 4
== Deleted Grapes ==
Grapefruit = 2
Pears = 4
Apples = 2
Oranges = 4
```

2. Using the initial hash from Step 1, create a grocery list script. Allow the user to print the list by entering "l." Allow the user to add an item by typing "a." End the program when "q" is entered.

Sample output:

```
Grocery Lister
(a)dd (l)ist or (q)uit: l
Apples = 2
Grapes = 1
Oranges = 4
Pears = 4
(a)dd (l)ist or (q)uit: a
Enter name: Cherries
Enter quantity: 3
(a)dd (l)ist or (q)uit: l
Apples = 2
Cherries = 3
Grapes = 1
Oranges = 4
Pears = 4
(a)dd (l)ist or (q)uit: q
```

3. Update the script. Allow a user to delete an item from the list by entering a "d."

   Extra credit:

   - Allow the user to modify an item.
   - If the user tries to delete an item that does not exist, display an error message.

   Sample output:

```
Grocery Lister
(a)dd (d)elete (l)ist or (q)uit: l
Apples = 2
Grapes = 1
Oranges = 4
Pears = 4
(a)dd (d)elete (l)ist or (q)uit: d
Enter name: Grapes
(a)dd (d)elete (l)ist or (q)uit: l
Apples = 2
Oranges = 4
Pears = 4
(a)dd (d)elete (l)ist or (q)uit: q
```

4. Write a script that reads in an array of words. Count the number of occurrences of each word, and print the results in a sorted order.

   Sample output:

   Please enter a list of words (Control-D to end list):**if**

   and
   or
   else
   if
   and
   or
   else
   if
   do
   while
   if
   do
   while
   and
   or
   but
   <Ctrl-D>

```
Here are the words you entered and an occurrence count:
"and" occured 3 times.
"but" occured 1 times.
"do" occured 2 times.
"else" occured 2 times.
"if" occured 4 times.
"or" occured 3 times.
"while" occured 2 times.
```

5. Write a script that identifies each user name on a system and counts the number of processes that each user owns. Use the output of `ps -ef` to get the needed data. If you split on spaces, `$array[0]` is the user name.

   Sample output:

```
User achoma owns 2 processes
User amcall owns 24 processes
User an113699 owns 6 processes
User annev owns 28 processes
etc...
```

Exercise Summary

Discussion – Take a few minutes to discuss the experiences, issues, or discoveries you had during the lab exercises.

- Experiences
- Interpretations
- Conclusions
- Applications

Exercise Solutions

1. In this exercise, use a grocery list to define a hash using the following definition. Then create a script to perform the following hash operations.

```
%hash1 = ("Apples" => 2,
          "Oranges" => 4,
          "Grapes" => 1,
          "Pears" => 4);
```

   - Print out the hash.
   - Add a key for Grapefruit, and assign it a value of 2. Print out the list sorted by its keys.
   - Delete Grapes, and print out the list using the each function.

   Sample output:

```
The initial values in the hash are:
Grapes = 1
Pears = 4
Apples = 2
Oranges = 4
== Added new Fruit==
Apples = 2
Grapefruit = 2
Grapes = 1
Oranges = 4
Pears = 4
== Deleted Grapes ==
Grapefruit = 2
Pears = 4
Apples = 2
Oranges = 4
```

Suggested solution:

lab0501.plx

```
 1 #!/usr/bin/perl -w
 2
 3 # Assign values to a Hash.
 4 %hash1 = ("Apples" => 2,
 5           "Oranges" => 4,
 6           "Grapes" => 1,
 7           "Pears" => 4);
 8
 9 #Print the Hash out
10 print "\nThe initial values in the hash are:\n";
11
12 foreach $key (keys %hash1){
13   print "$key = $hash1{$key}\n";
14 }
15
16 #Add two Grapefruites to the list and print out
17 $hash1{Grapefruit} = 2;
18 print "== Added new Fruit==\n";
19
20 foreach $key (sort keys %hash1){
21   print "$key = $hash1{$key}\n";
22 }
23
24 #Delete Grapes from the list and print the list using each
25 delete $hash1{Grapes};
26 print "== Deleted Grapes ==\n";
27
28 while (($fruit,$quan) = each %hash1){
29   print "$fruit = $quan\n";
30 }
```

2. Using the initial hash from Step 1, create a grocery list script. Allow the user to print the list by entering "l." Allow the user to add an item by typing "a." End the program when "q" is entered.

   Sample output:

```
Grocery Lister
(a)dd (l)ist or (q)uit: l
Apples = 2
Grapes = 1
Oranges = 4
Pears = 4
(a)dd (l)ist or (q)uit: a
Enter name: Cherries
Enter quantity: 3
(a)dd (l)ist or (q)uit: l
Apples = 2
Cherries = 3
Grapes = 1
Oranges = 4
Pears = 4
(a)dd (l)ist or (q)uit: q
```

Suggested solution:

lab0502.plx

```
 1 #!/usr/bin/perl -w
 2
 3 %hash1 = ("Apples"=>2,
 4           "Oranges"=>4,
 5           "Grapes"=>1,
 6           "Pears"=>4);
 7
 8 print "Grocery Lister\n";
 9
10 while ( 1 ){ # main loop
11
12       print "(a)dd (l)ist or (q)uit: ";
13       chomp($cmd = <STDIN>);
14
15       last if $cmd eq "q" ; # Exit if q
16
17       if ($cmd eq "l"){ # print hash
18
19          for (sort keys (%hash1)){
20                print "$_ = $hash1{$_}\n";
21          }
22       }
23
24       if ($cmd eq "a"){ # Add Item
25          print "Enter name: ";
26          chomp($name = <STDIN>);
27
28          print "Enter quantity: ";
29          chomp($qty = <STDIN>);
30
31          $hash1{$name} = $qty;
32
33       }
34 }
```

3. Update the script. Allow a user to delete an item from the list by entering "d."

   Extra credit:

   - Allow the user to modify an item.
   - If the user tries to delete an item that does not exist, display an error message.

   Sample output:

```
Grocery Lister
(a)dd (d)elete (l)ist or (q)uit: l
Apples = 2
Grapes = 1
Oranges = 4
Pears = 4
(a)dd (d)elete (l)ist or (q)uit: d
Enter name: Grapes
(a)dd (d)elete (l)ist or (q)uit: l
Apples = 2
Oranges = 4
Pears = 4
(a)dd (d)elete (l)ist or (q)uit: q
```

Suggested solutions:

lab0503a.plx

```
 1 #!/usr/bin/perl -w
 2
 3 %hash1 = ("Apples"=>2,
 4                   "Oranges"=>4,
 5                   "Grapes"=>1,
 6                   "Pears"=>4);
 7
 8 print "Grocery Lister\n";
 9
10 while ( 1 ){ # Main loop
11
12       print "(a)dd (d)elete (l)ist or (q)uit: ";
13       chomp($cmd = <STDIN>);
14
15       last if $cmd eq "q" ;
16
17       if ($cmd eq "l"){ # List hash
18
19          foreach (sort keys (%hash1)){
20                print "$_ = $hash1{$_}\n";
21          }
22       }
23
24       if ($cmd eq "a"){ # Add Item
25
26          print "Enter name: ";
27          chomp($name = <STDIN>);
28
29          print "Enter quantity: ";
30          chomp($qty = <STDIN>);
31
32          $hash1{$name} = $qty;
33       }
34
35       if ($cmd eq "d"){ # Delete Item
36
37          print "Enter name: ";
38          chomp($name = <STDIN>);
39
40          delete $hash1{$name};
41       }
42 }
```

lab0503b.plx

```
 1 #!/usr/bin/perl -w
 2
 3 %hash1 = ("Apples"=>2,
 4                   "Oranges"=>4,
 5                   "Grapes"=>1,
 6                   "Pears"=>4);
 7
 8 print "Grocery Lister\n";
 9
10 while ( 1 ){
11
12       print "(a)dd (d)elete (l)ist (m)odify or (q)uit: ";
13       chomp($cmd = <STDIN>);
14
15       last if $cmd eq "q" ;
16
17       if ($cmd eq "l"){ # List item in hash
18
19          foreach (sort keys (%hash1)){
20                 print "$_ = $hash1{$_}\n";
21          }
22       }
23
24       if ($cmd eq "a" || $cmd eq "m"){ # Add or modify
25          print "Enter name: ";
26          chomp($name = <STDIN>);
27
28          print "Enter quantity: ";
29          chomp($qty = <STDIN>);
30
31          $hash1{$name} = $qty;
32       }
33
34       if ($cmd eq "d"){ # Delete block
35
36          print "Enter name: ";
37          chomp($name = <STDIN>);
38
39          if (exists($hash1{name})) {
40              delete $hash1{$name};
41          }
```

```
42
43          else {
44              print "Name not in list\n";
45          }
46      }
47 }
```

4. Write a script that reads in an array of words. Count the number of occurrences of each word, and print the results in a sorted order.

   Sample output:

```
Please enter a list of words (Control-D to end list):if
and
or
else
if
and
or
else
if
do
while
if
do
while
and
or
but
<Ctrl-D>
Here are the words you entered:
"and" occured 3 times.
"but" occured 1 times.
"do" occured 2 times.
"else" occured 2 times.
"if" occured 4 times.
"or" occured 3 times.
"while" occured 2 times.
```

Suggested solution:

lab0504.plx

```
 1 #!/usr/bin/perl -w
 2
 3 print "Enter a list of words (Ctrl-D to end list):";
 4
 5 @wordlist=<STDIN>;
 6 chomp @wordlist;
 7
 8 foreach $word (@wordlist){
 9   $wordhash{$word}++; # count each instance
10 }
11
12 print "Here are the words you entered:\n";
13
14 foreach $word (sort keys %wordhash){
15   print "'$word' occured $wordhash{$word} times.\n";
16 }
```

5. Write a script that identifies each user name on a system, and counts the number of processes that each user owns. Use the output of `ps -ef` to get the needed data. If you split on spaces, `$array[0]` is the user name.

   Sample output:

```
User achoma owns 2 processes
User amcall owns 24 processes
User an113699 owns 6 processes
User annev owns 28 processes
etc...
```

Suggested solution:

lab0505.plx

```
 1 #!/usr/bin/perl -w
 2
 3 # Get a list of all processes on this system
 4 @pslist = qx/ps -ef/;
 5
 6 # Remove the header line
 7 shift @pslist;
 8
 9 foreach $line (@pslist){
10
11   @splitline = split " ", $line;
12   $count{$splitline[0]}++; # Count user names
13
14 }
15
16 foreach $user (sort keys %count){
17
18   print "User $user owns $count{$user} processes\n";
19
20 }
```

Module 6

Basic I/O and Regular Expressions

Objectives

Upon completion of this module, you should be able to:

- Use the <> operator to read a file specified on the command line, line-by-line
- Use the `printf` command to format the output of a script
- Test for a word or phrase in a file using regular expressions
- Use anchors and character classes in regular expressions
- Use alternation in regular expressions
- Use variable interpolation to define regular expressions
- Substitute one text string for another using regular expressions
- Extract parts of a string using regular expressions

Relevance

Discussion – The following questions are relevant to understanding regular expressions:

- Have you used regular expressions before? If so, what are they?
- What is the importance of regular expressions?

Introducing Basic I/O and Regular Expressions

Regular expressions create text patterns, which are used to search, extract, or replace parts of text strings. Regular expressions are very common in the UNIX world. They are used in `sed`, `awk`, `more`, `vi`, and `grep`. Why are they needed in practice?

System and network administrators often search log files for special text patterns that indicate critical situations. They also want to filter data that is the output of one program and the input for a second program. Database administrators need to convert data from one database format into another. (Data conversion is often referred to as "munging.")

Reading From the <> Filehandle

The diamond filehandle <> works just like the input mechanism in the standard Unix utilities: you can specify a file name (or a list of file names) on the command line, use a pipe to supply the data, or enter the data at the console after the script is invoked. This is an example of the Do What I Mean (DWIM) methodology of Perl.

For example, a script that reads a file specified on the command line and prints it to STDOUT line-by-line is written as follows:

```
e0601.plx
  1 #!/usr/bin/perl -w
  2
  3 while (<>){
  4   print;
  5 }

$ e0601.plx temp.plx
#!/usr/bin/perl -w
print "Welcome to our Perl Course\n";
```

This script reads one file. However, multiple files can be specified on the command line, and the <> filehandler reads them in automatically. This simplifies scripts, as opening and using explicit filehandles to perform the same function is slightly more complex.

Formatted Output

This section describes basic text formatting functions available in Perl.

The `printf` Function

Sometimes data needs to be printed in a fixed format. For example, printing a table with the `print` command would be difficult at best. With Perl, formatting text output is accomplished with the `printf` function. The function has the following form:

```
printf [FILEHANDLE] FORMAT, LIST;
```

The `printf` function has three arguments: a filehandle, a format string, and a data list. The format string contains a series of placeholders that indicate the output format of the values passed in the argument list.

The format placeholders start with a `%` and are written inside a string. The data to be printed follows the format string in a comma-separated list:

```
printf "%15s %5d %4.2f\n", "a", 5, 5.1;
```

The previous line is an example of a `printf` statement. This line prints the following:

- The "a" string in a 15-character-wide field that is right-justified
- The number 5 in a 5-character-wide field that is right-justified
- The number 5.1 in a 6-character-wide floating-point field that has two decimal places

Some of the commonly-used data types in a `printf` are:

```
d   decimal integer
f   fixed point floating-point format
s   string
-   left justified
+   right justified
```

In the following example, the script prints an ASCII table of 10 characters. The table prints the character, its ordinal value, and its ordinal value using decimal places. The first column is left justified, and the others are right justified. Right-justification is the default if nothing is specified.

```
e0602.plx
  1 #!/usr/bin/perl -w
  2
  3 #print ASCII Table
  4 printf "%-12s %8s %10s\n", "Character", "ASCII", "Dec ASCII";
  5
  6 $letnum = ord "a"; # Get ASCII number for "a"
  7
  8 for ($x = $letnum; $x <= ($letnum + 10); $x++){
  9
 10   printf "%-12s %8d %10.2f\n", chr($x), $x, $x;
 11
 12 }
```

```
$ e0602.plx
Character       ASCII  Dec ASCII
a                  97      97.00
b                  98      98.00
c                  99      99.00
d                 100     100.00
e                 101     101.00
f                 102     102.00
g                 103     103.00
h                 104     104.00
i                 105     105.00
j                 106     106.00
k                 107     107.00
```

A complete list of all format types and symbols for `printf` is found in man page `printf(3)`.

The `sprintf` Function

If you want to assign a formatted string to a variable instead of printing it out, you can use `sprintf`. The format specification is identical to the format specification for `printf`.

```
$fstring = sprintf FORMAT, LIST;
```

To demonstrate, the previous script is rewritten using `sprintf`. Instead of printing the formatted strings, the output is assigned to a scalar, which is then added to an array. The array is printed using `foreach`.

```
e0603.plx
  1 #!/usr/bin/perl -w
  2
  3 $output = sprintf "%-12s %8s %10s\n", "Character", "ASCII", "Dec
ASCII";
  4 push @out, $output;
  5
  6 $letnum = ord "a"; # Get ASCII number for "a"
  7
  8 for ($x = $letnum; $x <= ($letnum + 10); $x++){
  9
 10   $output = sprintf "%-12s %8d %10.2f\n", chr($x), $x, $x;
 11   push @out, $output;
 12
 13 }
 14
 15 for (@out){
 16   print;
 17 }
```

```
$ e0603.plx
Character       ASCII  Dec ASCII
a                  97      97.00
b                  98      98.00
c                  99      99.00
d                 100     100.00
e                 101     101.00
f                 102     102.00
g                 103     103.00
h                 104     104.00
i                 105     105.00
j                 106     106.00
k                 107     107.00
```

Page Formats

In addition to line-formatting commands, Perl offers data output in fixed and complex page formats. Such page formats are defined by the keyword `format`, and they are output with the command `write`. Appendix A, “Formats,” contains a detailed overview of page formats.

Regular Expressions

A regular expression is a way to define a pattern or template that matches some text in a search. The simplest way to create a regular expression is to search for a single word or phrase.

Binding Operator

In Perl, the binding operator (=~) specifies the variable which is to be tested against a regular expression. The following is a typical example of its usage.

e0604a.plx
```
  1 #!/usr/bin/perl -w
  2
  3 $line = "To ere is human";
  4
  5 if ($line =~ m/ere/) {
  6     print "$line\n";
  7 }
```

```
$ e0604a.plx
To ere is human
```

Line 5 is where the search takes place. The string to be searched is placed on the left of the binding operator, and the regular expression appears on the right. The text being searched for is preceded by an `m` (the matching operator) and placed between the slashes following it. The text, "ere" in this example, is the regular expression. If `ere` is found anywhere inside the string `$line`, the match is true, otherwise it is false. In this example, `ere` is found and the line is printed out.

The power of regular expressions comes when they are applied to a lot of data, a file for example. In many of the examples included in this module, the following file is searched.

```
$ cat regex.dat
ab
abc
a5e
a6f
123 a6c
a5b
a55b
a555b
a5555b
a55555b
a555555b
a5xb
1/4
3+2=5
def ghi
abc ab
```

Below, the last example is modified to use the diamond operator instead of a simple string, and then it is used on the `regex.dat` file.

```
e0604b.plx
  1 #!/usr/bin/perl -w
  2
  3 while (<>) {
  4   if ($_ =~ /ab/) { # Search $_ for ab
  5     print; # Print lines that include ab
  6   }
  7 }
```

```
$ e0604b.plx regex.dat
ab
abc
abc ab
```

The script now searches each line of the file passed on the command line. If `ab` is found anywhere in the line stored in `$_`, the line is printed out. Notice that the `m` operator is not used on Line 4. This is because the matching operator is the default when forward slashes are used as delimiters.

Just remember that the following two expressions are equivalent:

```
$_ =~ /ab/
$_ =~ m/ab/
```

Working With $_

Testing against `$_` can be simplified even further. If the text to be searched is contained in `$_`, neither the variable nor the operator has to be specified.

Therefore, `if ($_ =~ /ab/)` can be written as `if ( /ab/ )`.

Applying these simplifications to the previous script produces the following:

```
e0604c.plx
  1 #!/usr/bin/perl -w
  2
  3 while (<>) {
  4   if (/ab/) {
  5     print;
  6   }
  7 }

$ e0604c.plx regex.dat
ab
abc
abc ab
```

The same output as before is produced.

But even this script can be improved upon. For example:

```
e0604d.plx
  1 #!/usr/bin/perl -w
  2
  3 while (<>) {
  4   print if /ab/;
  5 }

$ e0604d.plx regex.dat
ab
abc
abc ab
```

This script explores patterns in the next section of this module. But, as always with Perl, there is yet one more way to write a script. This same script can be written this way:

```
perl -wne '/ab/ && print' regex.dat
```

This distills the previous script down to one line.

Patterns

This section describes how patterns build the search criteria for regular expressions. Patterns are the building blocks of regular expressions. The examples that follow all search the file `regex.dat`. You can use either of the following scripts to search the file.

```
regtest.plx
  1 #!/usr/bin/perl -w
  2
  3 while (<>) {
  4   print if /ab/;
  5 }
```

or

```
perl -wne 'print if /ab/;' regex.dat
```

Note – In the examples that follow, only the regular expression being tested is included in the text. The scripts used previously to produce the results have been left out for brevity.

Metacharacters

To build patterns that match more than simple words or phrases, a number of special characters are used to search for single characters. These characters are described in Table 6-1.

Table 6-1 Special Characters for Searching for Single-Character Patterns

Character	Description
`.`	The dot stands for any character (letter, digit, or special character) except `\n`.
`[...]`	Characters within square brackets build a character class. Any character found in the class returns a true value for the expression. A `-` indicates a range of values.
`[^..]`	Any character that is not in the list is a match. This negates the character class.
`\`	This causes the character following `\` to be taken literally. There are many characters with special meanings in regular expressions (for example, `.`, `*`, or `+`) `.` To match a `+` sign, prefix it with a backslash.
`|`	The `or` operator causes multiple patterns to be matched alternatively.

Examples

The following examples show how special characters are used in a regular expression to search for single-character patterns.

The . Character

The following expression matches all occurrences of "a" followed by any character, and then c.

```
/a.c/
```

Results:

```
abc
123 a6c
abc ab
```

The [...] Character Class

The following expression matches all three-character instances where the letter a is followed by any character which is then followed by any lowercase letter except c.

```
/a.[abd-z]/
```

Results:

```
a5e
a6f
a5b
a5xb
```

The following expression matches a string where an a is followed by a digit which is followed by a lowercase letter.

```
/a[0-9][a-z]/
```

Results:

```
a5e
a6f
123 a6c
a5b
a5xb
```

The [^..] Character Class

The following example matches any line that contains an a which is followed by two characters that are not digits. Note that the negative class means "anything except" the specified set. The newline character is a valid match, which is why the first ab (which is really ab\n) matches.

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

Results:

```
ab
abc
abc ab
```

The \ Character

The following expression matches a digit followed by + (plus sign) and another digit.

```
/[0-9]\+[0-9]/
```

Results:

```
3+2=5
```

The | Character

The following expression matches all the lines that contain bc or a6.

```
/bc|a6/
```

Results:

```
abc
a6f
123 a6c
abc ab
```

Default Character Classes

You can derive common character classes from the previous examples:

```
[012345689] # any digit
[0-9] # any digit
[a-z0-9] # any lowercase letter or digit
[a-zA-Z0-9] # any letter or digit
```

In fact, Perl includes special predefined characters as shorthand for these character classes. The default character classes are shown in Table 6-2.

Table 6-2 Default Character Classes

Predefined Character	Character Class	Negated Character	Negated Class
\d (digit)	[0-9]	\D	[^0-9]
\w (word-building char)	[a-zA-Z0-9_]	\W	[^a-zA-Z0-9_]
\s (white space)	[\r\t\n\f]	\S	[^ \r\t\n\f]

Examples

The following expression matches any line that contains a followed by two non-digits.

```
/a\D\D/
```

Results:

```
ab
abc
abc ab
```

The following expression matches any line that includes a digit followed by a non-"word-building" character.

```
/\d\W/
```

Results:

```
123 a6c
1/4
3+2=5
```

The following expression matches any pattern with at least five characters, the fourth of which is a space or a tab (white space).

.

```
/...\s./
```

Results:

```
123 a6c
def ghi
abc ab
```

Anchors: Word and String Boundaries

Anchors determine the edges of the search patterns. Patterns may be anchored to the start or end of strings or words as well as lines. The anchor characters are shown in Table 6-3.

Table 6-3 Anchors

Anchor	Description
^	The beginning of the line.
$	The end of the line.
\b	A word boundary. To delimit a word, put \b in the front and at the end of the pattern. A word is everything that consists of \w characters that ends before a\W character or newline.
\B	This is the opposite of \b, which specifies that the word does not end at this point.

Examples

The following examples demonstrate the use of anchors.

The ^ Anchor

The following expression searches for the sequence “a6” in the test file.

```
/a6/
```

Results:

```
a6f
123 a6c
```

Two strings are returned.

If the pattern is anchored to the beginning of the string, as in the following example, the number of lines returned changes.

```
/^a6/
```

Results:

```
a6f
```

Note – Do not confuse the ^ anchor with the negation character in a character class `[^...]`.

The $ Anchor

The following example searches for three-character patterns with `c` as the third character:

```
/..c/
```

Results:

```
abc
123 a6c
abc ab
```

If the pattern is anchored to the end of a string, the results change.

```
/..c$/
```

Results:

```
abc
123 a6c
```

The \b Anchor

The following expression finds any `ab` followed by a word boundary.

```
/ab\b/
```

Results:

```
ab
abc ab
```

The \B Anchor

The following expression finds any ab not on a word boundary.

```
/ab\B/
```

Results:

```
abc
abc ab
```

Quantifiers

To specify that a placeholder is repeated a number of times, a quantifier is used. Table 6-4 describes pattern quantifiers.

Table 6-4 Pattern Quantifiers

Quantifier	Description
*	The previous character is repeated zero or more times.
+	The previous character is repeated one or more times.
?	The previous character must appear exactly one time or not at all.
{n}	The previous character appears exactly *n* times.
{m,n}	The previous character appears from *m* to *n* times.
{m,}	The previous character appears *m* or more times.
(..)	This groups characters together for use in alternation.

Examples

The following examples demonstrate the use of pattern quantifiers.

The *, +, and ? Quantifiers

The following expression finds a followed by three or more 5's.

```
/a5555*/
```

Results:

```
a555b
a5555b
a55555b
a555555b
```

Why is a555b returned when there are four 5's in the expression? The * following the last 5 means that it appears zero or more times. Because the fourth 5 did indeed "appear" zero times in a555b, the match is true.

Changing the previous example to use a +, as follows, yields a different result.

```
/a5555+/
```

Results:

```
a5555b
a55555b
a555555b
```

Because at least one 5 is required, a555b is not a match with +.

The following expression matches all patterns starting with a, followed by zero or one 5's and a non-digit.

```
/a5?\D/
```

Results:

```
ab
abc
a5e
a5b
a5xb
abc ab
```

The {n}, {m,n}, and {m,} Quantifiers

The following expression matches an a followed by exactly four occurrences of 5 and a non-digit.

```
/a5{4}\D/
```

Results:

```
a5555b
```

The following expression matches an a, three to five occurrences of 5, and a non-digit.

```
/a5{3,5}\D/
```

Results:

```
a555b
a5555b
a55555b
```

The following expression matches an `a`, four or more occurrences of `5`, and a non-digit.

```
/a5{4,}\D/
```

Results:

```
a5555b
a55555b
a555555b
```

The `(..)` Quantifier

The following expression matches `55` repeated twice.

```
/(55){2}/
```

Results:

```
a5555b
a55555b
a555555b
```

Exercise: Use Regular Expressions to Search Files

In this exercise, you will write scripts that:

- Use the <> operator to read a file specified on the command line, line-by-line
- Use the `printf` command to format the output of a script
- Test for a word or phrase in a file using regular expressions
- Use anchors and character classes in regular expressions
- Use alternation in regular expressions
- Use variable interpolation to define regular expressions

Tasks

Complete the following steps:

1. Write a script that performs the same function as the `more` command. When displaying the file, pause the display every 24 lines. Let the user press Return to display the next 24 lines.

2. Modify the script you created in the previous step to display the line number of each line displayed.

   Sample output:

```
$ lab0602.plx temp.plx
  1 #!/usr/bin/perl -w
  2 print "Welcome to our Perl Course\n";
  3
```

3. Modify the previous script to use a regular expression to search a file passed on the command line. Have your script search for any word that contains `then` in the file `/usr/dict/words`. Print out all the words you find. If you do not find any matches, print out a message stating this.

   Sample output:

```
$ lab0603.plx /usr/dict/words
  1 Athena
  2 Athenian
  3 Athens
  4 authentic
  5 authenticate
  6 calisthenic
  7 earthen
  8 earthenware
  9 Eratosthenes
 10 heathen
 11 heathenish
 12 lengthen
 13 neurasthenic
 14 Parthenon
 15 ruthenium
 16 strengthen
 17 then
 18 thence
 19 thenceforth
```

4. Modify the script you created in the previous step to find any word that begins with a `b`, contains two consecutive `r`'s, and ends in a `y`.

   Sample output:

```
$ lab0604.plx /usr/dict/words
  1 baneberry
  2 barberry
  3 bayberry
  4 bearberry
  5 berry
  6 blackberry
  7 blueberry
  8 blurry
```

5. Modify the script you created in the previous step to find any word that contains the words `soft` or `ware`.

   Sample output:

```
$ lab0605.plx /usr/dict/words
  1 aware
  2 beware
  3 Delaware
  4 dinnerware
  5 earthenware
  6 failsoft
  7 firmware
  8 flatware
  9 glassware
 10 greenware
 11 hardware
 12 hollowware
 13 housewares
 14 silverware
 15 soft
 16 softball
 17 soften
 18 software
 19 softwood
 20 stoneware
 21 ware
 22 warehouse
 23 warehouseman
```

6. Modify the previous script so that you can enter the regular expression from a prompt. Test your script using the same expression shown below in the example.

   Hint: You can put a variable in a regular expression to perform a search. For example, `/$search/`. The variable is expanded to the value stored in it.

   Sample output:

```
$ lab0606.plx /usr/dict/words
What would you like to search for?  ^\wrs\w*
  1 arsenal
  2 arsenate
  3 arsenic
  4 arsenide
  5 arsine
  6 arson
  7 ersatz
  8 Erskine
  9 Mrs
 10 Ursa
 11 Ursula
 12 Ursuline
```

Capturing and Back-Referencing

In a search pattern, a substring specified in an expression often needs to be referred to. To mark a substring, surround it with parentheses. The first pattern can be referenced with the variable `$1`, the second with `$2`, and so on; these values persist until the next successful match. This technique is known as *capturing*. The following script demonstrates how capturing can be used to swap two names.

```
e0605.plx
  1 #!/usr/bin/perl -w
  2
  3 $lf = "Doe, Jane";
  4
  5 if ($lf =~ /(\w+),\s(\w+)/){
  6   print "$2 $1\n"; # Swap first and last
  7 }

$ e0605.plx
Jane Doe
```

Lines 5 and 6 show how parts of a string can be identified and then reused. In this case, the first and last names are identified, and then swapped and printed out.

When used in list context, the match operator returns the parts of the text matched by the captured patterns.

```
e0606.plx
  1 #!/usr/bin/perl -w
  2
  3 $str = "Name:Pantani,First Name:Marco,Country:Italy";
  4
  5 @arr = $str =~ /Name:(.*),First Name:(.*),Country:(.*)/;
  6
  7 print "\$1 = $1 \$2 = $2 \$3 = $3\n";
  8
  9 for ($x=0; $x <= $#arr; $x++){
 10   print "\$arr[$x] = $arr[$x] ";
 11 }
 12 print "\n";

$ e0606.plx
$1 = Pantani $2 = Marco $3 = Italy
$arr[0] = Pantani $arr[1] = Marco $arr[2] = Italy
```

The following example reads the `login` and the `comment` field from the `/etc/passwd` file.

```
e0607a.plx
  1 #!/usr/bin/perl -w
  2
  3 while (<>) {
  4   /(.*):.*:.*:.*:(.*):.*:.*/;
  5   $login = $1;
  6   $comment = $2;
  7   print "Login: $login  Comment:  $comment\n";
  8 }
```

```
$ e0607a.plx /etc/passwd
Login: adm  Comment:  Admin
Login: lp  Comment:  Line Printer Admin
Login: uucp  Comment:  uucp Admin
Login: nuucp  Comment:  uucp Admin
Login: listen  Comment:  Network Admin
Login: nobody  Comment:  Nobody
```

So far, you have looked at capturing, which is used once the regular expression has been executed. However, you can also use temporary matches known as back-references inside the regular expressions. For example, the following script uses this feature to find the word `Mississippi` in the `/usr/dict/words` file.

```
e0607b.plx
  1 #!/usr/bin/perl -w
  2
  3 # Search for "mississippi";
  4 # Use backreferencing within expression
  5
  6 while(<>) {
  7   print if /\w+i(ss)i\1/;
  8 }
```

```
$ e0607b.plx /usr/dict/words
Mississippi
Mississippian
```

Within the regular expression, back-references can be accessed with `\1`, `\2`, and so on. In this case, the repeated `ss` is referred to as `\1`.

This final example shows the power of regular expressions. The expression used in this script tests for any integer or float, but it does not consider other cases such as fractions, imaginary numbers, or numbers containing exponents.

e0607c.plx

```
 1 #! /usr/bin/perl -w
 2
 3 # A Perl script that test to see
 4 # if the user enter a number and/or float
 5
 6 while (1) {
 7   print "Enter a number: ";
 8   chomp($ans = <STDIN>);
 9
10   if ( $ans !~ /^(-|\+)?(\.?\d+$|\d+\.?\d*$)/ ) {
11     print "You did NOT enter a number...\n";
12   }
13   else {
14     print "You did enter a number...\n";
15   }
16 }
```

```
$ e0607c.plx
Enter a number: 2.5
You did enter a number...
Enter a number: blah
You did NOT enter a number...
```

Greediness

Patterns always try to match the largest possible part of a string. This property is called *greediness*. To demonstrate, examine the following script:

```
e0608a.plx
  1 #!/usr/bin/perl -w
  2
  3 $string = "Long, long ago in a galaxy far, far away";
  4
  5 if ($string =~ /(f.*r)/){
  6   print "\$1 = $1\n";
  7 }

$ e0608a.plx
$1 = far, far
```

Perl always grabs the largest string that matches the search pattern. Even though `far` matches the pattern, `far, far` also matches the pattern, and because it is larger, that is what is extracted.

To change a pattern so it matches the smallest possible match, put a `?` "generosity modifier" after the wildcard (`.*`). The script changes to this:

```
e0608b.plx
  1 #!/usr/bin/perl -w
  2
  3 $string = "Long, long ago in a galaxy far, far away";
  4
  5 if ($string =~ /(f.*?r)/){
  6   print "\$1 = $1\n";
  7 }

$ e0608b.plx
$1 = far
```

Now the smallest match is picked.

Special Variables

Table 6-5 describes the four other special variables used in match operations.

Table 6-5 Special Variables

Variable	Meaning
$&	Contains the matching part of the whole string
$`	Contains the left part of the string (before the matching position)
$´	Contains the right part (after the matching position)
$+	Contains the last match in parentheses

To demonstrate, print all values stored in these variables using the previous example as a starting point.

e0609.plx

```
  1 #! /usr/bin/perl -w
  2
  3 $var = 'quadrangle';
  4
  5 $var =~ /d(.)/;     # matches dr
  6
  7 print "\$` $`\n";  # What's before dr
  8 print "\$' $'\n";  # What's after dr
  9 print "\$& $&\n";  # What matched /d(r)/
 10 print "\$+ $+\n";  # What matched (r)
```

```
$ e0609.plx
$` qua
$' angle
$& dr
$+ r
```

Substitute Operator

The substitute operator replaces a matching text pattern with a new string. The operator is written as follows:

```
$string =~ s/match/replacement/;
```

In `$string` the pattern `match` is searched and replaced by the string or the regular expression `replacement`. This short script demonstrates how the substitute operator might be used:

```
e0610.plx
  1 #!/usr/bin/perl -w
  2
  3 $string = "Error: 236 in myscript.plx"; # Error Number
  4
  5 # Substitute Message and print
  6 $string =~ s/236/File not writable/;
  7 print $string,"\n";

$ e0610.plx
Error: File not writable in myscript.plx
```

Modifiers

Perl includes a number of modifiers that change the search expression.

The i Modifier

The i modifier makes searches case-insensitive. For example, the following substitute operator replaces any instance of the target pattern.

```
e0611.plx
  1 #!/usr/bin/perl -w
  2
  3 $search = "The stars live in hollywood, Hollywood, HollyWood!\n";
  4
  5 # Swap Hollywood for TinselTown, ignore case
  6 $search =~ s/Hollywood/TinselTown/i;
  7
  8 print $search;

$ e0611.plx
The stars live in TinselTown, Hollywood, HollyWood!
```

However, by default, the substitute operator replaces only the first occurrence. How can all occurrences be replaced?

The g Modifier

The global modifier searches a string and replaces all occurrences of the matching pattern. Applying the modifier to the previous example is shown as follows.

```
e0612.plx
  1 #!/usr/bin/perl -w
  2
  3 $search = "The stars live in hollywood, Hollywood, HollyWood!\n";
  4
  5 # Swap TinselTown for all Hollywood's, ignore case
  6 $search =~ s/Hollywood/TinselTown/ig;
  7
  8 print $search;

$ e0612.plx
The stars live in TinselTown, TinselTown, TinselTown!
```

The problem is solved. The script now replaces all occurrences in a string.

More on the Match Operator

There are several other miscellaneous topics worth mentioning.

Matches Not

The opposite of the match operator is `!~` (pronounced "matches not"). It returns 0 if the pattern matches and 1 if not. The example below shows a good occasion for the use of the "`unless`" operator, which makes the code more readable:

```
print "Pattern NOT found!\n" if $_ !~ /^abc/;
print "Pattern NOT found!\n" unless /^abc/;
```

Changing Delimiters

If another delimiter is needed in a regular expression, any other non-alphanumeric character may be used as long as the "`m`" (match) operator is present. For example, to use the colon as a delimiter, do the following:

```
if ( $line =~ m:/^abc/: ) { ... };
```

Translation Operator

The translation operator, `tr`, is a special variation of the match operator. It requires two lists of characters and replaces every character in the first list with the equivalent character in the second list.

```
e0613a.plx
  1 #!/usr/bin/perl -w
  2
  3 $str = "abcdabcd";
  4
  5 $str =~ tr/a/A/; # Make A's Uppercase
  6 print "After first tr, \$str = $str\n";
  7
  8 $str =~ tr/a-z/A-Z/ ; # Make all Upper
  9 print "After second tr, \$str = $str\n";

$ e0613a.plx
After first tr, $str = AbcdAbcd
After second tr, $str = ABCDABCD
```

Line 5 shows a single-character replacement. Each `a` is made uppercase. Line 8 shows how a range of values can be translated all at once. For those who are familiar with `sed`, Perl provides an alias of `y` for `tr`. These may be used interchangeably.

Squeezing Modifier

The `tr` operator can remove consecutive repeating characters from a string using the `s` modifier. For example:

```
e0613b.plx
  1 #!/usr/bin/perl -w
  2
  3 $str = "aaaabbbccccddddddd";
  4
  5 print "Start with: $str\n";
  6
  7 $str =~ tr/a-z/a-z/s ; #remove all repeating letters
  8 print "After tr, \$str = $str\n";

$ e0613b.plx
Start with: aaaabbbccccddddddd
After tr, $str = abcd
```

All of the repeating letters (`a`, `b`, `c`, `d`) are collapsed down to a single instance.

Complement Modifier

The complement modifier tells `tr` to replace any character not included in the first list with the characters in the second list. For example:

```
e0613c.plx
  1 #!/usr/bin/perl -w
  2
  3 $str = "jack and jill, jane and spot";
  4
  5 print "Start with: $str\n";
  6
  7 $count = ($str =~ tr/a-z//c); # Count non a-z
  8 print "There were $count non alpha characters\n";
  9
 10 $str =~ tr/a-z/_/c ; # Change to _
 11 print "After tr, \$str = $str\n";
```

```
$ e0613c.plx
Start with: jack and jill, jane and spot
There were 6 non alpha characters
After tr, $str = jack_and_jill__jane_and_spot
```

Line 7 is an example of using a modifier to count characters. By not specifying a replacement string, the characters that match are counted and stored in the scalar `$count`. In this example, there are 6 non-letter characters, five spaces, and a comma. Line 10 shows the modifier in action. All non-letters are replaced with underscores.

Truncating Second List

There is one final technique of interest. If the length of the second list does not match the first, the last character in the second list will replace any characters in the first list beyond it. For example:

```
e0613d.plx
  1 #!/usr/bin/perl -w
  2
  3 $test = "abcdabcdefg";
  4 print "Start with: $test\n";
  5
  6 $test =~ tr/a-z/a-c/ ; #anything past c is c
  7 print "After tr, \$test = $test\n";

$ e0613d.plx
Start with: abcdabcdefg
After tr, $test = abccabccccc
```

In this example, the binding operator shows you do not have to use `$_`. Notice that Line 6 causes all the characters after `c` in the string to be replaced with `c`.

Example: Processing Log Files

The following is a sample from a `last` command file. The command produces a list of user IDs and their login and logout times.

```
dhan      ftp          starship.Central Fri Mar 23 10:27 - 10:27  (00:00)
heidia    ftp          esintranet.Centr Fri Mar 23 10:23 - 10:23  (00:00)
heidia    ftp          esintranet.Centr Fri Mar 23 10:22 - 10:23  (00:00)
ap120137  dtlocal      :196             Fri Mar 23 10:22 - 10:24  (00:02)
heidia    ftp          esintranet.Centr Fri Mar 23 10:18 - 10:18  (00:00)
```

Assume you are only interested in file transfer protocol (FTP) connections. First, a script is created to isolate only the FTP entries. The output is captured in a file named `login.txt`.

```
e0614.plx
  1 #!/usr/bin/perl -w
  2
  3 while (<>){
  4   print if /\bftp\b/;
  5 }
```

```
$ e0614.plx login.txt
sgeist    ftp          finprod.Central. Wed Mar 28 14:39 - 14:39  (00:00)
weaverl   ftp          itcdev.Central.S Wed Mar 28 13:59 - 13:59  (00:00)
sw74388   ftp          discover.Central Wed Mar 28 13:09 - 13:09  (00:00)
```

The script that follows creates a simple report that counts how many FTP connections each user has made and how many FTP connections each host has made. The script uses a regular expression to extract the user and host name, and it counts the occurrences of each.

e0615.plx

```
 1  #!/usr/bin/perl -w
 2
 3  while (<>) {
 4    # Only check lines with ftp.
 5
 6      # Get UserID, and host
 7      next unless /^(\w+)\s*ftp\s+(\S+)/;
 8
 9      $uid = $1; $host = $2;
10      $users{$uid}++;              # record some characteristics
11      $hosts{$host}++;
12
13  }
14
15  print "FTP User Report:\n";
16  printf "%-10s %12s\n","User","Connections" ;
17
18  foreach (sort keys %users){
19    printf "%-10s %12d\n", $_,  $users{$_};
20  }
21
22  print "\nFTP Host Report:\n";
23  printf "%-22s %12s\n","Host","Connections";
24
25  foreach (sort keys %hosts){
26    printf "%-22s %12d\n", $_,  $hosts{$_};
27  }
```

The script produces the following output.

```
$ e0615.plx login.txt
FTP User Report:
User        Connections
an113699              4
aw128359              1
dangoe                9
dhan                  1
haasc                 1
heidia               11
jbronson              7
pquinn                1
sb110810              7
scairns               1
sgeist                4
sw74388               3
tp127462              1
wbrock                5
weaverl               5
wm123109              2

FTP Host Report:
Host                    Connections
cti01.EBay.Sun.C                  2
ctiws02                           1
dhcp-brm1-10-6                    2
dhcp-brm5-50-33                   8
discover.Central                  1
eastapp2.East.Su                  9
edueast.East.Sun                  1
eiffel                            1
esintranet.Centr                 12
esun1as-be.Centr                  7
finprod.Central.                  4
gigabyte                          1
itcdev.Central.S                  4
itctest.Central.                  1
mac103.Central.S                  4
nafo-gds                          1
starship.Central                  1
sunray7                           1
vacd01.Central.S                  1
webman.Central.S                  1
```

The Smart Match Operator

Perl 5.10 introduced a new binary operator, ~~, called the smart match operator. It compares its operands in a smart way. It can be used to compare two scalars, arrays, and hashes. It can use regular expressions and compare a scalare with an array or a hash. In addition, you do not need to put in the "use 5.010' pragma. Unllike many of the other new features of Perl 5.10, it will be included automatically.

Lets look at some examples of this new operator. The example introduces a lot of new variables and then uses ~~ to compare them.

e0616,plx

```
  1 #!/usr/bin/perl -w
  2 #use 5.010;
  3
  4 $a = "Foo";
  5 $b = "Foo";
  6 $b2 = "Bar";
  7 $c = 123;
  7 $d = 123.0;
  8
  9 print "\$a eq \$b\n" if $a ~~ $b;
 10 print "\$a ne \$b2\n" unless $a ~~ $b2;
 11 print "\$c == \$d\n" if $c ~~ $d;
 12
 13 %fflint = ( "name" => "flintstone",
 14 "fname" => "fred",
 15 "job" => "stonecutter");
 16 %fflint2 = ( "name" => "flintsone",
 17 "fname" => "wilma",
 18 "job" => "houswife");
 19 @list = qw\Foo Bar Baz\;
 20 @list2 = qw\Foo Bar Baz\;
 21
 22 use 5.010;
 23 say "equal lists" if @list ~~ @list2;
 24 say "equal key sets for hashes" if %fflint ~~ %fflint2;
 25
```

```
 26 say "\@list contains $a" if $a ~~ @list;
 27 say "\@list contains $a" if @list ~~ $a;
 28 #~~ is a commutative operator
 29 $b = "name";
 30 say "$b is a key" if ($b ~~ %fflint);
 31 say "$b is a key" if (%fflint ~~ $b);
 32
 33 #find the "ob" at the end of the key "job" in the %fflint hash
 34 say "found" if /ob$/ ~~ %fflint;
```

```
$ e0616.plx
$a eq $b
$a ne $b2
$c == $d
equal lists
equal key sets for hashes
@list contains Foo
@list contains Foo
name is a key
name is a key
found
```

First, note that "use 5.010;" is not needed to use the Smart Match operator. Second, the operator can tell, in a smart way, if two numbers are equal. If it one of the operands is recognized as a number, a numerical comparison is performed. For two arrays to be equal, it means that the elements at each index are equal. For two hashes to be equal, it means that the arrays formed by their keys are equal. The example shows that equality of values is not required.

The Smart Match operator can check if a scaler is in an array of if the scalar is a key for a hash. Since this check is commutative, it is not necessary to try to remember the correct order of the operands.

In addition, where a scalar was used, a requlr expression could have been used. It is truly a smart operator. Does this mean that "=~" is now the "Dumb Operator"?

Exercise: Using Substitution, Capturing, and Back-Referencing

In this exercise, you write scripts that:

- Substitute one text string for another using regular expressions
- Extract parts of a string using regular expressions

Tasks

Complete the following steps:

7. Given the following definition, write a script that replaces all three-character instances of x with `red`, and all four-character instances of x with `blue`.

   ```
   $a = "green xxx brown xxx yellow xxxx xxxx gray xxxx
   pink";
   ```

   Sample output:

   ```
   Starting value:
   green xxx brown xxx yellow xxxx xxxx gray xxxx pink

   After substituting red:
   green red brown red yellow xxxx xxxx gray xxxx pink

   After substitution blue:
   green red brown red yellow blue blue gray blue pink
   ```

8. Write a script that reads the `/etc/passwd` file and produces formatted output like that shown on the next page. Use regular expressions to remove the desired fields.

   Hint: Use the `printf` statement to format the output. To left-justify a string field, the statement is the following:

   ```
   printf ("%-10s", $string)
   ```

   Hint: The fields in a login file are:

   ```
   Login:passwd:uid:gid:gcos:dir:shell
   ```

Sample output:

```
$ lab0608.plx /etc/passwd

   Login Directory      Shell           Comments
    root /              /sbin/sh        Super-User
  daemon /              /usr/bin/true
     bin /usr/bin       /usr/bin/true
     sys /              /usr/bin/true
     adm /var/adm       /usr/bin/true   Admin
      lp /usr/spool/lp  /usr/bin/true   Line Printer Admin
    uucp /usr/lib/uucp  /usr/bin/true   uucp Admin
  nobody /              /usr/bin/true   Nobody
```

9. You have been asked to write a script that converts a date from a log file into a new format. Given the following definition, create a script that converts the date into the format shown in the following sample output.

   ```
   $date = "Tue Aug 13 15:35:23 MET DST 2000" ;
   ```

   Sample output:

   ```
   Date: 13.8.2000  Time: 15:35
   ```

10. (Optional) Using the example file called `e0615.plx` in "Example: Processing Log Files" on page 6-39, create a new reporting script. Your script should report connections that occur on Tuesday or Wednesday. The output should be the same as shown previously in this module.

    For extra credit, try any of the following:

    - Calculate FTP connections for Monday through Friday.
    - Calculate the time each user is logged in.
    - Create a separate report for each connection type.

11. (Optional) Write a script that searches the output of the `finger` command using a regular expression. Prompt the user to enter the regular expression, and any line matches should be printed out as shown below.

    Sample output:

```
$ lab0611.plx sunray5
Enter your search string: Mark
For host host5, the following lines meet your search criteria:
[host5]
Login       Name               TTY         Idle    When    Where
mbell Mark Bell           dtlocal          Tue 08:22  :25
mminden  Mark Minden       dtlocal          Wed 10:32  :205
mbummer Mark Bummer         dtlocal         Wed 08:19  :144
mlaga  Mark Laga - Sun Ed  dtlocal          Wed 11:02  :209
```

12. Given the following array of names, @list = qw/tom dick harry peter paul mary jane/, ask the user to enter a name and then check if the name is in the list. If the name is not in the list, ask the user if he would like to join. Then print out the list of names.

lab0612.plx

```
     Please enter a name.
     David
     David is not on the list.  Would you like this name added to the
list? (Y or N)
     yes
     Here is the list:  tom dick harry peter paul mary jane david
```

Exercise Summary

Discussion – Take a few minutes to discuss the experiences, issues, or discoveries you had during the lab exercises.

- Experiences
- Interpretations
- Conclusions
- Applications

Exercise Solutions

1. Write a script that performs the same function as the `more` command. When displaying the file, pause the display every 24 lines. Let the user press Return to display the next 24 lines.

   Suggested solution:

```
lab0601.plx
  1 #!/usr/bin/perl -w
  2
  3 $count=0;
  4
  5 while (<>){
  6   print;
  7
  8   if ($count < 23){
  9     $count++;
 10
 11   }else{
 12     $_ = <STDIN>;
 13     $count=0;
 14   }
 15 }
```

2. Modify the script you created in the previous step to display the line number of each line displayed.

   Sample output:

```
$ lab0602.plx temp.plx
  1 #!/usr/bin/perl -w
  2 print "Welcome to our Perl Course\n";
  3
```

Suggested solution:

lab0602.plx

```
 1 #!/usr/bin/perl -w
 2
 3 $lineno = 1; $count=0;
 4
 5 while (<>){
 6   printf "%3s ", $lineno;
 7   print;
 8
 9   $lineno++;
10
11   if ($count < 23){
12     $count++;
13
14   }else{
15     $_ = <STDIN>;
16     $count=0;
17   }
18 }
```

3. Modify the previous script to use a regular expression to search a file passed on the command line. Have your script search for any word that contains `then` in the file `/usr/dict/words`. Print out all the words you find. If you do not find any matches, print out a message stating this.

   Sample output:

```
$ lab0603.plx /usr/dict/words
  1 Athena
  2 Athenian
  3 Athens
  4 authentic
  5 authenticate
  6 calisthenic
  7 earthen
  8 earthenware
  9 Eratosthenes
 10 heathen
 11 heathenish
 12 lengthen
 13 neurasthenic
 14 Parthenon
 15 ruthenium
 16 strengthen
 17 then
 18 thence
 19 thenceforth
```

Suggested solution:

```
lab0603.plx
  1 #!/usr/bin/perl -w
  2
  3 $lineno = 0; $count=0;
  4
  5 while (<>){
  6
  7   if (/then/){
  8     $lineno++;
  9
 10     printf "%3s ", $lineno;
 11     print;
 12
 13     if ($count < 23){
 14       $count++;
 15
 16     }else{
 17       $_ = <STDIN>;
 18       $count = 0;
 19     }
 20
 21   }
 22 }
 23 if ($lineno == 0){
 24   print "Sorry, no matches!\n";
 25 }
```

4. Modify the script you created in the previous step to find any word that begins with a `b` contains two consecutive `r`'s, and ends in a `y`.

   Sample output:

```
$ lab0604.plx /usr/dict/words
  1 baneberry
  2 barberry
  3 bayberry
  4 bearberry
  5 berry
  6 blackberry
  7 blueberry
  8 blurry
```

Suggested solution:

lab0604.plx

```
 1 #!/usr/bin/perl -w
 2
 3 $lineno = 0; $count = 0;
 4
 5 while (<>){
 6
 7   if (/^b.*rr.*y$/){
 8     $lineno++;
 9     printf "%3s ",$lineno;
10     print;
11
12     if ($count < 23){
13       $count++;
14
15     }else{
16       $_ = <STDIN>;
17       $count = 0;
18     }
19
20   }
21 }
22 if ($lineno == 0){
23   print "Sorry, no matches!\n";
24 }
```

5. Modify the script you created in the previous step to find any word that contains the words `soft` or `ware`.

   Sample output:

```
$ lab0605.plx /usr/dict/words
  1 aware
  2 beware
  3 Delaware
  4 dinnerware
  5 earthenware
  6 failsoft
  7 firmware
  8 flatware
  9 glassware
 10 greenware
 11 hardware
 12 hollowware
 13 housewares
 14 silverware
 15 soft
 16 softball
 17 soften
 18 software
 19 softwood
 20 stoneware
 21 ware
 22 warehouse
 23 warehouseman
```

Suggested solution:

lab0605.plx

```
 1 #!/usr/bin/perl -w
 2
 3 $lineno = 0; $count = 0;
 4 while (<>){
 5
 6   if (/soft|ware/){
 7     $lineno++;
 8
 9     printf "%3s ", $lineno;
10     print;
11
12     if ($count < 23){
13       $count++;
14     }else{
15       $_ = <STDIN>;
16       $count = 0;
17    }
18   }
19 }
20 if ($lineno == 0){
21   print "Sorry, no matches!\n";
22 }
```

6. Modify the previous script so that you can enter the regular expression from a prompt. Test your script using the same expression shown below in the example.

   Hint: You can put a variable in a regular expression to perform a search. For example, `/$search/`. The variable is expanded to the value stored in it.

   Sample output:

```
$ lab0606.plx /usr/dict/words
What would you like to search for?  ^\wrs\w*
  1 arsenal
  2 arsenate
  3 arsenic
  4 arsenide
  5 arsine
  6 arson
  7 ersatz
  8 Erskine
  9 Mrs
 10 Ursa
 11 Ursula
 12 Ursuline
```

Suggested solution:

lab0606.plx

```
 1 #!/usr/bin/perl -w
 2
 3 $lineno = 0; $count = 0;
 4
 5 print "What would you like to search for?  ";
 6 chomp($search = <STDIN>);
 7
 8 while (<>){
 9
10   if (/$search/){
11     $lineno++;
12
13     printf "%3s ", $lineno;
14     print;
15
16     if ($count < 23){
17       $count++;
18
19     }else{
20       $_ = <STDIN>;
21       $count = 0;
22     }
23   }
24 }
25 if ($lineno == 0){
26   print "Sorry, no matches!\n";
27 }
```

7. Given the following definition, write a script that replaces all three-character instances of x with `red`, and all four-character instances of x with `blue`.

```
$a = "green xxx brown xxx yellow xxxx xxxx gray xxxx
pink";
```

Sample output:

```
Starting value:
green xxx brown xxx yellow xxxx xxxx gray xxxx pink

After substituting blue:
green xxx brown xxx yellow blue blue gray blue pink

After substitution red:
green red brown red yellow blue blue gray blue pink
```

Suggested solution:

lab0607.plx

```
 1 #!/usr/bin/perl -w
 2
 3 $a = "green xxx brown xxx yellow xxxx xxxx gray xxxx pink";
 4
 5 print "Starting value:\n";
 6 print "$a\n\n";
 7
 8 $a =~ s/x{4}/blue/g;
 9
10 print "After substituting blue:\n";
11 print "$a\n\n";
12
13 $a =~ s/x{3}/red/g;
14
15 print "After substitution red:\n";
16 print "$a\n\n";
```

8. Write a script that reads the `/etc/passwd` file and produces formatted output like the following. Use regular expressions to remove the desired fields.

   Hint: Use the `printf` statement to format the output. To left-justify a string field, the statement is the following:

   ```
   printf "%-s10", $string
   ```

   Hint: The fields in a login file are:

   ```
   Login:passwd:uid:gid:gcos:dir:shell
   ```

   Sample output:

```
$ lab0608.plx /etc/passwd

   Login Directory      Shell           Comments
    root /              /sbin/sh        Super-User
  daemon /              /usr/bin/true
     bin /usr/bin       /usr/bin/true
     sys /              /usr/bin/true
     adm /var/adm       /usr/bin/true   Admin
      lp /usr/spool/lp  /usr/bin/true   Line Printer Admin
    uucp /usr/lib/uucp  /usr/bin/true   uucp Admin
  nobody /              /usr/bin/true   Nobody
```

Suggested solution:

lab0608.plx

```
  1 #!/usr/bin/perl -w
  2
  3 print "\n";
  4 printf "%8s %-14s %-15s %-20s\n", "Login", "Directory", "Shell",
"Comments";
  5
  6 while (<>){
  7
  8   ($login, $gcos, $dir, $shell) = /(.*):.*:.*:.*:(.*):(.*):(.*)/;
  9   chomp $shell;
 10
 11   printf "%8s %-14s %-15s %-20s\n", $login, $dir, $shell, $gcos;
 12 }
```

9. You have been asked to write a script that converts a date from a log file into a new format. Given the following definition, create a script that converts the date into the format shown in the following sample output.

```
$date = "Tue Aug 13 15:35:23 MET DST 2000" ;
```

Sample output:

```
Date: 13.8.2000  Time: 15:35
```

Suggested solution:

lab0609.plx

```
 1 #!/usr/bin/perl -w
 2
 3 # Conversion of date format
 4 $date = "Tue Aug 13 15:35:23 MET DST 2000" ;
 5
 6 # We need something, that receives 'AUG' and returns '8'.
 7 %month = qw (Jan 1 Feb 2 Mar 3 Apr 4 May 5 Jun 6
 8              Jul 7 Aug 8 Sep 9 Oct 10 Nov 11 Dec 12) ;
 9
10 # Extracting interesting parts
11 $date =~ /(.*) (.*) (.*) (\d\d):(\d\d):\d\d (.*) (.*)/ ;
12
13 $newdate = "Date: $3.$month{$2}.$7  Time: $4:$5" ;
14 print "$newdate\n" ;
```

10. (Optional) Using the example file called `e0615.plx` in "Example: Processing Log Files" on page 6-39, create a new reporting script. Your script should report connections that occur on Tuesday or Wednesday. The output should be the same as shown previously in this module.

    For extra credit, try any of the following:

    - Calculate FTP connections for Monday through Friday.
    - Calculate the time each user is logged in.
    - Create a separate report for each connection type.

Suggested solution:

lab0610.plx

```
 1 #!/usr/bin/perl -w
 2
 3 while (<>) {
 4
 5   # Only check lines with ftp.
 6   if (/ftp/) {
 7
 8     # Get UserID and host
 9     /^(\w+)\s+ftp\s+(\S+)/;
10
11     if ($3 =~ /^(Tue|Wed)$/){
12       $uid = $1; $host = $2;
13       $users{$uid}++ ;              # record some characteristics
14       $hosts{$host}++ ;
15     }
16
17   }
18 }
19
20 print "FTP User Report:\n";
21 printf "%-10s%12s\n","User","Connections" ;
22
23 foreach (sort keys %users){
24   printf "%-10s%12d\n", $_,  $users{$_};
25 }
26
27 print "\nFTP Host Report:\n";
28 printf "%-22s%12s\n","Host","Connections" ;
29
30 foreach (sort keys %hosts){
31   printf "%-22s%12d\n", $_,  $hosts{$_};
32 }
```

11. (Optional) Write a script that searches the output of the `finger` command using a regular expression. Prompt the user to enter the regular expression, and any line matches should be printed out as shown below.

    Sample output:

```
$ lab0611.plx sunray5
Enter your search string: Mark
For host host5, the following lines meet your search criteria:
[host5]
Login       Name               TTY         Idle    When    Where
mbell Mark Bell          dtlocal          Tue 08:22  :25
mminden  Mark Minden       dtlocal          Wed 10:32  :205
mbummer Mark Bummer         dtlocal          Wed 08:19  :144
mlaga  Mark Laga - Sun Ed  dtlocal          Wed 11:02  :209
```

Suggested solution:

lab0611.plx

```
 1 #!/usr/bin/perl -w
 2
 3 print "Enter your search string: ";
 4 chomp($search=<STDIN>);
 5
 6 foreach $host (@ARGV){
 7   print "For $host, the following lines match:\n";
 8
 9   @list1 = qx/finger \@$host/;
10
11   print $list1[0];
12   print $list1[1];
13
14   foreach $line (@list1){
15
16     if ($line =~ /$search/){
17         print $line;
18     }
19   }
20 }
```

12. Given the following array of names, @list = qw/tom dick harry peter paul mary jane/, ask the user to enter a name and then check if the name is in the list. If the name is not in the list, ask the user if he would like to join. Then print out the list of names.

```
lab0612.plx
     Please enter a name.
     David
     David is not on the list.  Would you like this name added to the
list? (Y or N)
     yes
     Here is the list:  tom dick harry peter paul mary jane david
```

Suggested solution:

```
lab0612.plx
  1 #!/usr/bin/perl -w
  2 use 5.010;
  3
    4 @list = qw/tom dick harry peter paul mary jane/, ;
    5 say"Please enter a name.";
    6 chomp($name = <STDIN>);
    7 $lname = lc $name;
    8 if ($lname ~~ @list){
    9     say "$name is on the list.";
   10 } else{
   11     say "$name is not on the list";
   12      say "Would you like to add the name to the list? (Y or N)";
   13      chomp($resp = <STDIN>);
   14      if (uc($resp) ~~ /Y/){
   15         push @list, $lname;
   16     }
   17 }
   18
   19 say "Here is the list:\n@list";
```

Module 7

Filehandles and Files

Objectives

Upon completion of this module, you should be able to:

- Use filehandles to open a file
- Use `die` to display an error message if there is an error accessing a file
- Append data to the end of a file
- Use a filehandle to read the output of a program

Relevance

Discussion – The following questions are relevant to using files in Perl:

- When do you need filehandles?
- Why not use the <> instead?

Introducing Filehandles and Files

In Perl, filehandles are used for input and output operations. A filehandle is an easy-to-use interface that hides complex file-related activities. Perl performs all the reading or writing of data in the background. A few predefined filehandles, such as `STDIN` or `STDOUT`, are automatically opened by Perl. Others must be opened by you.

Filehandles have their own namespace. However, because there is no prefix like `$` or `@` for filehandles, the parser could confuse them with Perl keywords. Because all keywords are written in lowercase, typically filehandles are written in upper or mixed case, for example `MYFILE` or `Myfile`.

Opening and Reading Files

A file is opened and read in three basic steps. First, the filehandle is opened. Second, data is read from the filehandle. Then, the file is closed using the filehandle.

Opening a Filehandle

To open a file in read-mode, use the following syntax:

```
open FH, "filename" or die "Open Failed\n";
```

Choose any name for the filehandle. The file name is an absolute or relative path.

```
e0701.plx
  1 #!/usr/bin/perl -w
  2
  3 open FILE, "<temp.plx" or die "Open failed!\n";
  4
  5 print "File Opened Successfully!\n";
  6
  7 close FILE;

$ e0701.plx
File Opened Successfully!
```

This script demonstrates the basic steps needed to open and close a file. In Line 3, the file is opened using a file name `<temp.plx`. The `<` indicates the file is being opened for reading. This is the default setting and is normally left off when a file is opened only for reading. The filehandle `FILE` is created if the file is successfully opened.

However, if the file cannot be opened for some reason, that is where the `die` function comes in. It displays an error message and ends the program if the file cannot be opened. The statement can be read as "open the file or die!" The `or` is used as a flow-control operator, and has the lowest precedence of all operators (similar to `and` which is used for the same purpose) for precisely this reason. If the open is successful, the right side of the expression is never evaluated. If the open fails, then control is passed to the `die` function.

Line 7 closes the file with the `close` statement.

The following script provides an example of `die` in action. The file used on Line 3 does not exist.

```
e0702.plx
  1 #!/usr/bin/perl -w
  2
  3 open FILE,"badfile.plx" or die "Open failed!\n";
  4
  5 print "File Opened Successfully!\n";
  6
  7 close FILE;

$ e0702.plx
Open failed!
```

Because the file does not exist, the right side of the `or` statement is executed. This calls `die`, which displays the error message as shown.

Passing the Actual Error

The actual error detected by Perl can be passed to the `die` function using the `$!` variable. This variable provides more information about the actual cause of the problem. The script that follows incorporates this feature.

```
e0703.plx
  1 #!/usr/bin/perl -w
  2
  3 open FILE,"badfile.plx" or die "Open failed: $!\n";
  4
  5 print "File Opened Successfully!\n";
  6
  7 close FILE;

$ e0703.plx
Open failed: No such file or directory
```

This error message is much more informative than before.

Reading From Files

The next step in the process is to actually read lines from the file. To read in data from a filehandle, place the filehandle in angle brackets.

```
$line = <FILE>;
```

The input operator reads in the data line-by-line out of the source until the end of the file is reached. Then, the operator returns `undef`. Because `undef` is interpreted as false in a Boolean context, a loop that uses the input operator as its condition is automatically terminated at the end of a file (eof). The following script reads in a file and displays its contents.

```
e0704.plx
  1 #!/usr/bin/perl -w
  2
  3
  4 open FILE, "temp.plx" or die "Open failed: $!\n";
  5
  6 while ($line=<FILE>){
  7 print $line;
  8 }
  9
 10 print "----- EOF ----\n";
 11
 12 close FILE;
```

```
$ e0704.plx
#!/usr/bin/perl -w
print "Welcome to our Perl Course\n";

----- EOF ----
```

In scalar context, `<...>` always yields one single line, so loops are needed to read entire files.

Using $_

Reading files is a great way to use the `$_` variable. Instead of using a variable in a `while` loop, the previous scripts can be rewritten as follows:

e0705.plx

```
 1 #!/usr/bin/perl -w
 2
 3 open FILE, "temp.plx" or die "Open Failed: $!\n";
 4
 5 while (<FILE>){
 6   print;
 7 }
 8
 9 print "----- EOF ----\n";
10
11 close FILE;
```

```
$ e0705.plx
#!/usr/bin/perl -w
print "Welcome to our Perl Course\n";

----- EOF ----
```

Reading Files Into Arrays

In list context, all lines of the input source are returned at once. Each line becomes a separate element of the array. Thus, the previous example can also be written as follows.

e0706.plx

```
 1 #!/usr/bin/perl -w
 2
 3 open FILE, "temp.plx" or die "Open failed: $!\n";
 4
 5 @lines = <FILE>; # Slurp file into array
 6
 7 foreach (@lines) {
 8   print;
 9 }
10
11 print "----- EOF ----\n";
12 close FILE;
```

```
$ e0706.plx
#!/usr/bin/perl -w
print "Welcome to our Perl Course\n";

----- EOF ----
```

The advantage of this is that each line can be manipulated as an array element. For example, if multiple searches are performed on the file, having it in memory avoids having to reread the file.

Removing Newline

The newline character marks the end of a line. When reading from STDIN, trailing \n characters are returned with the line. The newlines are easily removed with chomp.

e0707a.plx

```
 1 #!/usr/bin/perl -w
 2
 3 $filename = "temp.plx";
 4
 5 open FILE,$filename or die "Open Failed: $!\n";
 6
 7 while (<FILE>){
 8   chomp; # Remove \n from $_
 9   print; # Print $_
10   print "\n";
11 }
12
13 close FILE;
```

```
$ e0707a.plx
#!/usr/bin/perl -w
print "Welcome to our Perl Course\n";
```

On Line 8, chomp removes any \n characters from the current line.

Assigning the Data

Data can easily be read into hashes as well. The following is a flat text file that contains hash data.

```
$ cat hashfile.txt
key1::Key1 data here
key2::key2 data goes here too
key3::Key3 is the third set of data
key4::Key4 is the fourth set of data
```

The following script reads this data, assigns it to a hash, and prints out the results.

```
e0707b.plx
  1 #!/usr/bin/perl -w
  2
  3 open FILE, "hashfile.txt" or die "Open failed: $!\n";
  4
  5 while (<FILE>){
  6
  7   chomp;
  8   ($key, $value) = split(/::/, $_);
  9   $hash1{$key} = $value;
 10
 11 }
 12
 13 for (keys %hash1) {
 14   print "$_ --> $hash1{$_}\n";
 15 }
 16
 17 print "----- EOF ----\n";
 18
 19 close FILE;
```

```
$ e0707b.plx
key1 --> Key1 data here
key2 --> key2 data goes here too
key3 --> Key3 is the third set of data
key4 --> Key4 is the fourth set of data
----- EOF ----
```

On Lines 7–9, each line is chomped, split, and then stored into the hash.

Reading From DATA

A predefined filehandle similar to <STDIN> is DATA. This filehandle reads data contained in the Perl script itself. The data is denoted by the marker __END__. In all other respects, lines below the __END__ mark are ignored by the Perl interpreter.

e0708.plx

```
 1 #!/usr/bin/perl -w
 2
 3 @lines = <DATA>; # Slurp <DATA>
 4
 5 print "The names stored in this file are:\n";
 6
 7 foreach $line (@lines){
 8   print $line;
 9 }
10
11 print "----- EOF ----\n";
12
13 __END__
14 Hans Schmidt
15 Rafaela Savore
16 France Roux
17 Jane Brown
```

```
$ e0708.plx
The names stored in this file are:
Hans Schmidt
Rafaela Savore
France Roux
Jane Brown
----- EOF ----
```

Writing to Files

Writing to files is similar to reading from them. First a filehandle is opened for output. Then, data is written to the file. Finally, the filehandle is closed.

Opening a Filehandle for Output

To open a filehandle for output, the `open` command is used, but the file name is prefixed by `>` or `>>`. The `>` and `>>` symbols have the same meaning as in UNIX shells. In the first case, the contents of the file is overwritten by the new data. In the second case, the new data is appended to the file.

```
open FH , ">filename";
open FH , ">>filename";
```

To open a file both for reading and writing, put `+<` in front of the file name. (A `<` opens the file only for reading, but, because that is the default, it is usually omitted.)

Writing to a Filehandle

To write data to a file, use `print`. The destination filehandle is specified as the first argument to `print`. The second argument is the data to be written. There is no comma between arguments.

```
print FILEHANDLE LIST;
```

First, the data from the last example is put in a separate file, `names1.txt`.

```
$ cat names1.txt
Hans Schmidt
Rafaela Savore
France Roux
Jane Brown
```

The following script reads in `names1.txt` and writes to a new file called `out.txt`.

e0709.plx

```
 1 #!/usr/bin/perl -w
 2
 3 # Open files
 4 open IN,"names1.txt" or die "Open failed: $!\n";
 5 open OUT,">out.txt" or die "Open failed: $!\n";
 6
 7 # Read a line and write it out
 8 while (<IN>) {
 9   print OUT "Out data: $_";
10 }
11
12 close IN;
13 close OUT;
14
15 print "File written\n";
```

```
$ e0709.plx
File written

$ cat out.txt
Out data: Hans Schmidt
Out data: Rafaela Savore
Out data: France Roux
Out data: Jane Brown
```

To append to a file, add an additional >.

e0710.plx

```
 1 #!/usr/bin/perl -w
 2
 3 # Open files, OUT to append
 4 open IN,"names1.txt" or die "Open failed: $!\n";
 5 open OUT,">>out.txt" or die "Open failed: $!\n";
 6
 7 # Read a line and write it out
 8 while (<IN>) {
 9   print OUT "New data: $_";
10 }
11
12 close IN;
13 close OUT;
14
15 print "File written\n";
```

```
$ e0710.plx
File written

$ cat out.txt
Out data: Hans Schmidt
Out data: Rafaela Savore
Out data: France Roux
Out data: Jane Brown
New data: Hans Schmidt
New data: Rafaela Savore
New data: France Roux
New data: Jane Brown
```

Modifying print

Usually print outputs its argument list sequentially with no space between elements and without a \n at the end. The behavior of print can be changed with $, and $\ special characters. The $, is the separator between elements, and $\ specifies the trailing character. The following script demonstrates how these variables change the output.

e0711.plx

```
  1 #!/usr/bin/perl -w
  2
  3 #Print without special variables changed
  4 print "Before:\n";
  5 print "aa","bb";
  6 print "cc","dd";
  7 print "\n\n";
  8
  9 # Print with $, and $\:q
q
 changed.
 10 print "After:\n";
 11 $, = "--"; # Changes separator to --
 12 $\ = "\n"; # Appends \n to each print command
 13 print "aa","bb";
 14 print "cc","dd";
```

```
$ e0711.plx
Before:
aabbccdd

After:
aa--bb
cc--dd
```

These variables are very helpful when creating flat-file databases and similar applications.

Writing to STDOUT and STDERR

STDOUT is the predefined name of the filehandle that Perl connects automatically to the standard output channel of the program. Usually, this turns out to be the screen. As shown repeatedly, if no filehandle is specified, the print command defaults to STDOUT.

```
e0712.plx
  1 #!/usr/bin/perl -w
  2
  3 print "This line is just printed\n";
  4 print STDOUT "But this line is printed to STDOUT\n";
```

```
$ e0712.plx
This line is just printed
But this line is printed to STDOUT
```

Both lines are printed to STDOUT.

The STDERR filehandle is pre-opened as well, and it is attached to the standard error channel. To print an error message, print to the STDERR filehandle as shown below.

```
e0713.plx
  1 #!/usr/bin/perl -w
  2
  3 print STDERR "It is not a good idea to divide by
zero\n";
```

```
$ e0713.plx
It is not a good idea to divide by zero
```

Normally, STDERR prints to the screen as well. Note that the warn operator prints to STDERR by default.

Processes and System Commands

Perl offers several methods for exchanging data with other processes. Depending on the situation, data is sent out to an external command or read into a script.

Starting Processes With Back Quotes or `system`

One way to start external processes is to use backquotes `` `...` `` or the `qx` operator as shown a number of times previously. You can use backquotes in a scalar or list context.

```
e0714.plx
  1 #!/usr/bin/perl -w
  2
  3 print "The date is ", qx/date/, "\n";
  4
  5 print "The following people are logged in:\n";
  6
  7 @list = qx/who | grep root/;
  8 print @list;
```

```
$ e0714.plx
The date is Thu Mar 18 14:12:31 MDT 2010

The following people are logged in:
mhosalli   dtlocal      Mar 11 15:41    (:25)
salas      pts/138      Mar 17 11:06    (:94.0)
salhadef   pts/316      Mar 11 15:49    (:73.0)
```

Essentially, a command initiated with backquotes is like a command initiated from a shell. The command is executed and the output from `STDOUT` is captured until the command completes. Also, the command must be completed before anything else can be done.

Using `system`

With `system`, a command is handed to a new shell where it is executed. However, only the command's success or failure is returned to the Perl script. If the command is successful, a 0 is returned. This is the opposite of what is normally expected.

```
e0715.plx
  1 #!/usr/bin/perl -w
  2
  3 print "Today's calendar:\n";
  4 system "cal";

$ e0715.plx
Today's calendar:
     March 2010
 S  M Tu  W Th  F  S
    1  2  3  4  5  6
 7  8  9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29 30 31
```

The command is successful and calls `cal`. Output is sent to `STDOUT`.

Scripts that use `system` to execute commands execute sequentially just like scripts that use backquotes. The only difference is that the output of a `system` command goes to `STDOUT` and not into a variable. To demonstrate, the following script executes until Control-C is pressed. However, only a 0 is returned to the script.

```
e0716.plx
  1 #!/usr/bin/perl -w
  2
  3 $host = $ARGV[0]; # Get host name
  4
  5 print "Is the system up?\n";
  6
  7 system "ping -s $host" and die "Cmd failed: $!\n";
  8
  9 system "cal 1 2010" and die "Cmd failed: $!\n";
```

```
$ e0716.plx oracle.com
Is the system up?
PING oracle.com: 56 data bytes
64 bytes from bigip-www-adc.oracle.com (141.146.9.91): icmp_seq=0. time=63.1 ms
64 bytes from bigip-www-adc.oracle.com (141.146.9.91): icmp_seq=1. time=59.7 ms
64 bytes from bigip-www-adc.oracle.com (141.146.9.91): icmp_seq=2. time=59.8 ms
^C
----oracle.com PING Statistics----
3 packets transmitted, 3 packets received, 0% packet loss
round-trip (ms)  min/avg/max/stddev = 59.7/60.9/63.1/1.9
   January 2010
 S  M Tu  W Th  F  S
                1  2
 3  4  5  6  7  8  9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
31
```

Notice the `and` flow-control operator used on Lines 7 and 9 is the opposite of what was used for files. System commands return `0` on success and non-zero on failure but `0` means "false" in Perl programming! Because of this, the sense of the test must be inverted, so we use an `and` where we would usually use an `or`.

From the output, you can see the commands execute sequentially.

Filehandles to Processes

Filehandles can be connected not only to files, but they can also be connected to processes. Using filehandles to read the output of a command is very similar to using back quotes. For example, the following is an example used earlier, rewritten to use filehandles.

e0717.plx

```
  1 #!/usr/bin/perl -w
  2
  3 $bytes = 0; $count = 0;
  4
  5 open LS,"ls -l |"; # Open ls as filehandle
  6
  7 $_ = <LS>; # Remove first line
  8
  9 while (<LS>) {
 10   print; # Print Current line
 11
 12   @line = split ' ';    # Split, remove whitespace
 13   $bytes = $bytes + $line[4]; # Get number of bytes
 14   $count++;                   # Count number of files/dirs
 15 }
 16 close LS;
 17
 18 print "$count files or dirs using $bytes bytes\n"
```

```
$ e0717.plx
-rwxr-xr-x   1 user9255 staff        120 May  8 09:00 e0701.plx
-rwxr-xr-x   1 user9255 staff        123 May  8 09:01 e0702.plx
-rwxr-xr-x   1 user9255 staff        124 May  8 09:27 e0703.plx
-rwxr-xr-x   1 user9255 staff        183 May  8 09:28 e0704.plx
... and so on
25 files or dirs using 5928 bytes
```

Line 5 attaches a filehandle to the `ls -l` command. The | at the end of the command creates a pipe that is attached to the specified filehandle and can be read from. Line 7 removes the first line from the output because it is not needed. Lines 9--15 loop through the file counting bytes and the number of files listed.

Piping

One additional feature of filehandles is the ability to write to another process. This is accomplished by putting a | at the start of a command. This opens a pipe to the input of the command that can be written to; this is often referred to as a *filter*.

```
open FH, "| command"
```

To demonstrate, the wc UNIX command can be improved by using a Perl script. In the example that follows, Perl reads a file in and sends its contents to the wc process using a filehandle. The script adds a header to the output of wc.

```
e0718.plx
  1 #!/usr/bin/perl -w
  2
  3 $filename = shift(@ARGV);
  4
  5 open INPUT, $filename or die "Open failed: $!\n";
  6
  7 open OUT,"| wc" or die "Output failed: $!\n";
  8
  9 print "   Lines   Words   Chars\n";
 10
 11 while (<INPUT>){
 12   print OUT;
 13 }
 14
 15 close INPUT;
 16 close OUT;

$ e0718.plx e0717.plx
   Lines   Words   Chars
      18      69     402
```

The output now includes three headers for the data printed from the command. The header is added on Line 9 before the looping on the file starts. Then, the file is sent to wc, which displays its normal output.

Creating a Basic Text Database

As an example, the code for the beginnings of a basic customer support application is described in the following section. This application uses several of the concepts already explained in this course. At this stage, the application performs the following functions:

- Lists all the customers in the database if `e0719list.plx` is executed
- Adds a record to the database file if `e0719add.plx` is executed

The Text File

The first example that follows is not the script code but the actual text database. Each record is indexed based on the customer's first name. This indexed name is separated from the recorded name with `===`. Each field in the record (name, department, severity, problem) is separated by `--`. This is the format that must be used to read and write data.

```
$ cat support.db
John===John--hr--4--My hard drive has fallen and it can't get up
Jane===Jane--eng--1--My server has crashed
Jill===Jill--IS--1--My pager stopped working
```

Listing Records

The script that follows prints out all the records in this database. Notice on Line 19 that the value portion of the hash is split into its various parts.

e0719list.plx

```
 1 #!/usr/bin/perl -w
 2
 3 $file = "support.db";
 4
 5 open IN, "$file" or die "Open failed: $!\n";
 6
 7 while (<IN>){
 8
 9 ($k,$v) = split(/===/, $_); # Key/data separated by ===
10   $data{$k} = $v;  # Create hash
11 }
12
13 close IN; # Close File
14
15 print "Open Problems\n";
16 print "="x65,"\n";
17
18 foreach $key (sort keys %data){ # Print DB
19   ($name, $dept, $sev, $prob) = split /--/, $data{$key};
20   print "First Name: $name\n";
21   print "Department: $dept\n";
22   print "Severity:   $sev\n";
23   print "Problem:    $prob\n" ;
24   print "-"x65,"\n";
25 }
```

```
$ e0719list.plx
Open Problems
=====================================================================
First Name: Jane
Department: eng
Severity:   1
Problem:    My server has crashed

---------------------------------------------------------------------
First Name: Jill
Department: IS
Severity:   1
Problem:    My pager stopped working

---------------------------------------------------------------------
First Name: John
Department: hr
Severity:   4
Problem:    My hard drive has fallen and it can't get up

---------------------------------------------------------------------
```

Adding a Record

With the add script, data is collected from the user, put into an appropriate format on Line 12, and appended to the data file.

e0719add.plx

```
 1 #!/usr/bin/perl -w
 2
 3 $file = "support.db";
 4
 5 print "First Name:     "; $name = <STDIN>;
 6 print "Department:     "; $dept = <STDIN>;
 7 print "Severity [1-5]: "; $sev = <STDIN>;
 8 print "Problem:        "; $prob = <STDIN>;
 9 chomp ($name, $dept, $sev, $prob);
10
11 # Join parts together and save into file
12 $v = "$name--$dept--$sev--$prob";
13
14 open OUT, ">>$file" or die "Open failed: $!\n";
15 print OUT "$name===$v\n";
16
17 close OUT;
```

```
$ e0719add.plx
First Name:     Dave
Department:     Jones
Severity [1-5]: 2
Problem:        My keyboard is broken
```

Of course, this is only the start. Scripts are needed to edit records, to delete records, and, when the database gets larger, to search for records.

IExercise: Create Scripts Using Files and Filehandles

In this exercise, you complete the following tasks:

- Use filehandles to open a file
- Use `die` to display an error message if there is an error accessing a file
- Append data to the end of a file
- Use a filehandle to read the output of a program

Tasks

Complete the following steps:

1. Use filehandles to create enhanced versions of the UNIX utility `cat`. Display an appropriate error message if a bad file name is passed.

   a. Write a script that performs the same function as the `cat` command. When displaying the file, pause the display every 24 lines. Let the user press Return to display the next 24 lines.

   b. Modify the previous script to display the contents of the file preceded by the file's name.

   c. Modify the script you created in the previous step to display the line number of each line displayed.

   Sample output:

```
$ lab0701a.plx temp.plx
#!/usr/bin/perl -w
print "Welcome to our Perl Course\n";


$ lab0701b.plx temp.plx
temp.plx: #!/usr/bin/perl -w
temp.plx: print "Welcome to our Perl Course\n";
temp.plx:

$ lab0701c.plx temp.plx
  1 #!/usr/bin/perl -w
  2 print "Welcome to our Perl Course\n";
  3
```

2. Write a script that replaces a string throughout an entire file. Pass the script an input and output file from the command line. Prompt for the search and replace strings in the script. Test your script on the files shown in the following sample output.

   Sample output:

```
$ lab0702.plx temp.html out.html
Please enter the string you wish to search for: <li>
Please enter the replacement string: <ListItem>
```

3. Modify the script you created previously to print the number of lines in the file and the number of replacements that took place.

   Hint: `$replaced = s/old/new/g; # Counts replacements`

```
$ lab0703.plx temp.html out.html
Please enter the string you wish to search for: <li>
Please enter the replacement string: <listitem>
54 lines searched.
5 replacements.
```

4. You have been given a file that contains a list of acronyms and their definitions (`acronyms.dat`). Read the file into a hash indexed on the acronym. Ask a user to enter an acronym name. If the acronym is in your database, display the definition. If `all` is entered, display all the terms in your database. Enter `quit` to exit your program.

   Sample output:

```
$ lab0704.plx
Please enter an acronym: www
WWW: World Wide Web

Please enter an acronym: slip
SLIP: Serial Line Internet Protocol

Please enter an acronym: quit
```

5. Create a simple address book database using text files and hashes. The application consists of two scripts. One script lists all the names in the database. The second adds a new record to the database.

   Every record is one line in the database. Keys are separated from other data by ---. Record fields are separated by ##.

   Here is what the database looks like.

```
$ cat address.db
Smith---Smith##Alanis##10, Covent Garden, London##004420/238958
Jones---Jones##Matt##1234 1st, New York, NY##303-123-1234
Baker---Baker##Fred##456 2nd st, Los Angeles, CA##651-123-4567
Wong---Wong##Bev##4546 8th, Miami, FL##909-324-8788
```

   Sample output:

```
$ lab0705list.plx
Name:    Baker, Fred
Address: 456 2nd st, Los Angeles, CA
Phone:   651-123-4567

Name:    Jones, Matt
Address: 1234 1st, New York, NY
Phone:   303-123-1234

Name:    Smith, Alanis
Address: 10, Covent Garden, London
Phone:   004420/238958

Name:    Wong, Bev
Address: 4546 8th, Miami, FL
Phone:   909-324-8788

$ lab0705add.plx
Last name:  Smith
First name: John
Address:    123 2nd St
Phone:      303-555-1214
Smith added
```

6. (Optional) Every month you receive sales data for a new product. The data is stored in a file (`june.dat`) with one number per line. Each line represents the number of items sold for that day.

   Write a Perl script that computes the average daily sales for the items sold. In addition, create a report that displays the number of the day, the number of items sold on that day, the difference from the average, and the percentage difference from the average. A sample of what the report should look like is listed below. If the percent difference is greater than 10 percent, put a + next to the `%diff` column. If the percent difference is greater than 20 percent, put ++ next to the `%diff` column.

   Sample output:

```
$ lab0706.plx
Statistical Report for june.dat
-------------------------------------------------
Mean: 134

     day   amount       diff      %diff

       1      112        -22       -16%
       2      166         32        23% ++
       3      115        -19       -14%
       4      145         11         8%
       5      168         34        25% ++
       6      166         32        23% ++
       7      131         -3        -2%
       8      111        -23       -17%
       9      116        -18       -13%
      10      147         13         9%
      11      160         26        19%  +
etc.....

-------------------------------------------------
```

7. (Optional) In industrial production, Perl is sometimes used as an I/O filter. Imagine two machines, one of which produces data that is used by the second one. If the data formats of both machines are not compatible, Perl converts the data to an appropriate format.

   You can find an output file of an imaginary machine in `machine1.txt`. Take a look at it. Write a Perl filter that reads the data, skips all lines beginning with #, swaps Rows 2 and 3 of the data lines, and writes the data to a new file `machine2.txt`. Print the screen as you write it to the file. Your script should immediately stop processing when the line "`# END DATA`" is found.

   Sample output:

```
$ cat machine1.txt
# ----------------------
# machine: Lago7
# date: January 10, 2000
# time: 09:56:34
# ----------------------
#
# running 67h 46m 23s
# software revision: 3.65
# last check: 06/27/99
# next check: 06/27/00
#
# BEGIN DATA
135.45 65.3 456.67
142.87 65.4 442.17
129.83 65.4 433.51
132.62 65.3 416.65
141.88 65.3 471.19
128.43 65.2 463.38
131.65 65.3 426.57
139.89 65.3 422.18
138.33 65.4 484.41
145.46 65.3 466.69
132.67 65.4 432.57
149.89 65.4 433.56
# END DATA
#
# Warranty:
# Hadrian will not give any warranty
# for the correctness of the preceding
# data, since ....
```

```
$ lab0707.plx
135.45 456.67 65.3
142.87 442.17 65.4
129.83 433.51 65.4
132.62 416.65 65.3
141.88 471.19 65.3
128.43 463.38 65.2
131.65 426.57 65.3
139.89 422.18 65.3
138.33 484.41 65.4
145.46 466.69 65.3
132.67 432.57 65.4
149.89 433.56 65.4

$ cat machine2.txt
135.45 456.67 65.3
142.87 442.17 65.4
129.83 433.51 65.4
132.62 416.65 65.3
141.88 471.19 65.3
128.43 463.38 65.2
131.65 426.57 65.3
139.89 422.18 65.3
138.33 484.41 65.4
145.46 466.69 65.3
132.67 432.57 65.4
149.89 433.56 65.4
```

8. (Optional) Write a script that reads in a list of hosts passed on the command line and then displays the number of users logged into each host.

   Use the command `finger @host` to obtain a list of the users logged into the host. Use a filehandle to process the command.

   Remove the first line or two from the command so that they are not included in your count.

   Sample output:

```
$ lab0408.plx host1 host2 host3
The servers currently have the following load.
host1 has 146 users
host2 has 82 users
host3 has 115 users
```

Exercise Summary

Discussion – Take a few minutes to discuss the experiences, issues, or discoveries you had during the lab exercises.

- Experiences
- Interpretations
- Conclusions
- Applications

Exercise Solutions

1. Use filehandles to create enhanced versions of the UNIX utility `cat`. Display an appropriate error message if a bad file name is passed.

   a. Write a script that performs the same function as the `cat` command. When displaying the file, pause the display every 24 lines. Let the user press Return to display the next 24 lines.

   b. Modify the previous script to display the contents of the file preceded by the file's name.

   c. Modify the script you created in the previous step to display the line number of each line displayed.

   Sample output:

```
$ lab0701a.plx temp.plx
#!/usr/bin/perl -w
print "Welcome to our Perl Course\n";


$ lab0701b.plx temp.plx
temp.plx: #!/usr/bin/perl -w
temp.plx: print "Welcome to our Perl Course\n";
temp.plx:

$ lab0701c.plx temp.plx
  1 #!/usr/bin/perl -w
  2 print "Welcome to our Perl Course\n";
  3
```

Suggested solutions:

lab0701a.plx

```
 1 #!/usr/bin/perl -w
 2
 3 $count=0;
 4 $filename = $ARGV[0];
 5
 6 open(IN, $filename) || die("Open failed: $!\n");
 7
 8 while (<IN>){
 9   print $_;
10
11   if ($count < 23){
12     $count++;
13
14   }else{
15     $_ = <STDIN>;
16     $count=0;
17   }
18 }
```

lab0701b.plx

```
 1 #!/usr/bin/perl -w
 2
 3 $count=0;
 4 $filename = $ARGV[0];
 5
 6 open(IN, $filename) || die("Open failed: $!\n");
 7
 8 while (<IN>){
 9   print "$filename: $_";
10
11   if ($count < 23){
12     $count++;
13
14   }else{
15     $_ = <STDIN>;
16     $count=0;
17   }
18 }
```

lab0701c.plx

```
 1 #!/usr/bin/perl -w
 2
 3 $lineno = 1; $count=0;
 4 $filename = $ARGV[0];
 5
 6 open(IN, $filename) || die("Open failed: $!\n");
 7
 8 while (<IN>){
 9   printf "%3s ", $lineno;
10   print $_;
11
12   $lineno++;
13
14   if ($count < 23){
15     $count++;
16
17   }else{
18     $_ = <STDIN>;
19     $count=0;
20   }
21 }
```

2. Write a script that replaces a string throughout an entire file. Pass the script an input and output file from the command line. Prompt for the search and replace strings in the script. Test your script on the files shown in the following sample output.

   Sample output:

```
$ lab0702.plx temp.html out.html
Please enter the string you wish to search for: <li>
Please enter the replacement string: <ListItem>
```

Suggested solution:

lab0702.plx

```
 1 #!/usr/bin/perl -w
 2
 3 #Get file parameters
 4 $fileIn = $ARGV[0];
 5 $fileOut = $ARGV[1];
 6
 7 #Get search and replace values
 8 print "Please enter the string you wish to search for: ";
 9 chomp($search=<STDIN>);
10
11 print "Please enter the replacement string: ";
12 chomp($replace=<STDIN>);
13
14 #Open file and perform replace.
15 open OUT,">$fileOut" or die "Open failed: $!\n";
16 open IN, $fileIn or die "Open failed: $!\n";
17
18 while (<IN>){
19   s/$search/$replace/g;
20   print OUT;
21 }
22
23 close IN;
24 close OUT;
```

3. Modify the script you created previously to print the number of lines in the file and the number of replacements that took place.

   Hint: `$replaced = s/old/new/g; # Counts replacements`

   ```
   $ lab0703.plx temp.html out.html
   Please enter the string you wish to search for: <li>
   Please enter the replacement string: <listitem>
   54 lines searched.
   5 replacements.
   ```

Suggested solution:

lab0703.plx

```
 1 #!/usr/bin/perl -w
 2 $count = 0;
 3 $replaced = 0;
 4
 5 #Get file parameters
 6 $fileIn = $ARGV[0];
 7 $fileOut = $ARGV[1];
 8
 9 #Get search and replace values
10 print "Please enter the string you wish to search for: ";
11 chomp($search=<STDIN>);
12
13 print "Please enter the replacement string: ";
14 chomp($replace=<STDIN>);
15
16 #Open file and perform replace.
17 open OUT,">$fileOut" or die "Open failed: $!\n";
18 open IN, $fileIn or die "Open failed: $!\n";
19
20 while (<IN>){
21
22   $temp = s/$search/$replace/g;
23   print OUT $_ ;
24   $count++;
25   $replaced += $temp;
26
27 }
28
29 close IN;
30 close OUT;
31
32 print "$count lines searched.\n";
33 print "$replaced replacements.\n";
```

4. You have been given a file that contains a list of acronyms and their definitions (`acronyms.dat`). Read the file into a hash indexed on the acronym. Ask a user to enter an acronym name. If the acronym is in your database, display the definition. If `all` is entered, display all the terms in your database. Enter `quit` to exit your program.

   Sample output:

```
$ lab0704.plx
Please enter an acronym: www
WWW: World Wide Web

Please enter an acronym: slip
SLIP: Serial Line Internet Protocol

Please enter an acronym: quit
```

Suggested solution:

lab0704.plx

```
 1 #!/usr/bin/perl -w
 2
 3 open ACRO, "acronyms.dat" or die "Open failed: $!\n";
 4
 5 # Split up each line into acronym and description
 6 while (<ACRO>){
 7   chomp;
 8   ($k, $v) = split /##/;
 9   $acro{$k} = $v;
10 }
11
12 close ACRO;
13
14 while (1){
15   print "Please enter an acronym: ";
16   chomp($input=<STDIN>);
17   $input = uc $input;
18
19   last if ($input eq "QUIT");
20
21   if ($input eq "ALL"){
22     print "All Acronyms in our db\n";
23     print "-"x30,"\n";
24     foreach $key (sort keys %acro){
25       print "$key: ";
26       print "$acro{$key}\n";
27     }
28     print "-"x30,"\n\n";
29     next;
30   }
31
32   if (exists($acro{$input})) {
33     print "$input: ";
34     print "$acro{$input}\n\n";
35   }
36   else
37   {
38     print "Acronym not found. Try again or 'quit'\n";
39   }
40 }
```

5. Create a simple address book database using text files and hashes. The application consists of two scripts. One script lists all the names in the database. The second adds a new record to the database.

   Every record is one line in the database. Keys are separated from other data by ---. Record fields are separated by ##.

   Here is what the database looks like.

```
$ cat address.db
Smith---Smith##Alanis##10, Covent Garden, London##004420/238958
Jones---Jones##Matt##1234 1st, New York, NY##303-123-1234
Baker---Baker##Fred##456 2nd st, Los Angeles, CA##651-123-4567
Wong---Wong##Bev##4546 8th, Miami, FL##909-324-8788
```

Sample output:

```
$ lab0705list.plx
Name:    Baker, Fred
Address: 456 2nd st, Los Angeles, CA
Phone:   651-123-4567

Name:    Jones, Matt
Address: 1234 1st, New York, NY
Phone:   303-123-1234

Name:    Smith, Alanis
Address: 10, Covent Garden, London
Phone:   004420/238958

Name:    Wong, Bev
Address: 4546 8th, Miami, FL
Phone:   909-324-8788

$ lab0705add.plx
Last name:  Smith
First name: John
Address:    123 2nd St
Phone:      303-555-1214
Smith added
```

Suggested solutions:

lab0705list.plx

```
 1 #!/usr/bin/perl -w
 2
 3 $file = "address.db" ;
 4
 5 # Reading it all database records.
 6 open IN, "$file" or die "Open failed: $!\n";
 7
 8 # Split up the records into key and value.
 9 while (<IN>){
10   ($k,$v) = split /---/, $_;
11   $data{$k} = $v ;
12 }
13 close IN ;
14
15 foreach (sort keys %data) {
16   ($name, $fname, $addr, $phone) = split /##/, $data{$_};
17
18   print "Name:    $name, $fname\n";
19   print "Address: $addr\n";
20   print "Phone:   $phone\n";
21 }
```

lab0705add.plx

```
 1 #!/usr/bin/perl -w
 2
 3 $file = "address.db" ;
 4
 5 print "Last name:  "; $name = <STDIN>;
 6 print "First name: "; $fname = <STDIN>;
 7 print "Address:    "; $addr = <STDIN>;
 8 print "Phone:      "; $phone = <STDIN>;
 9 chomp ($name, $fname, $addr, $phone);
10
11 # Join parts together and save into file.
12 $v = "$name##$fname##$addr##$phone";
13
14 open OUT, ">>$file" or die "Open failed: $!\n";
15 print OUT "$name---$v\n" ;
16
17 close OUT ;
18
19 print "$name added\n";
```

6. (Optional) Every month you receive sales data for a new product. The data is stored in a file (`june.dat`) with one number per line. Each line represents the number of items sold for that day.

   Write a Perl script that computes the average daily sales for the items sold. In addition, create a report that displays the number of the day, the number of items sold on that day, the difference from the average, and the percentage difference from the average. A sample of what the report should look like is listed below. If the percent difference is greater than 10 percent, put a + next to the `%diff` column. If the percent difference is greater than 20 percent, put ++ next to the `%diff` column.

   Sample output:

```
$ lab0706.plx
Statistical Report for june.dat
-------------------------------------------------
Mean: 134

     day   amount       diff      %diff

       1      112        -22       -16%
       2      166         32        23% ++
       3      115        -19       -14%
       4      145         11         8%
       5      168         34        25% ++
       6      166         32        23% ++
       7      131         -3        -2%
       8      111        -23       -17%
       9      116        -18       -13%
      10      147         13         9%
      11      160         26        19%  +
etc.....

-------------------------------------------------
```

Suggested solution:

lab0706.plx

```
  1 #!/usr/bin/perl -w
  2
  3 $statfile = "june.dat";
  4
  5 open STAT, $statfile or die "Open failed: $!\n";
  6
  7 @stat = <STAT> ;
  8 chomp @stat ;
  9 close STAT ;
 10
 11 # Calc mean
 12 foreach $day (@stat) {
 13   $sum += $day ;
 14 }
 15
 16 # Print header and mean
 17 $mean = int($sum / ($#stat +1)) ;
 18
 19 print "\nStatistical Report for $statfile\n" ;
 20 print "-" x 50 , "\n" ;
 21 print "Mean: $mean\n\n" ;
 22 printf ("%8s %8s %10s %10s","day","amount","diff","%diff");
 23 print "\n\n";
 24
 25 # Calculating statistical values and printing them out
 26 for ($i=0 ; $i<=$#stat ; $i++) {
 27   $d = $i +1 ;
 28   $diff = $stat[$i] - $mean ;
 29   $pdiff = int ($diff / $mean * 100) ;
 30   $mark = "" ;
 31
 32   if ($pdiff >= 10) {$mark = "+"} ;
 33   if ($pdiff >= 20) {$mark = "++"} ;
 34
 35   printf "%8s %8s %10s %10s %2s", $d, $stat[$i], $diff, $pdiff."%",
$mark;
 36   print "\n";
 37 }
 38
 39 print "\n" , "-" x 50 , "\n\n" ;
```

7. (Optional) In industrial production, Perl is sometimes employed as an I/O filter. Imagine two machines, one of which produces data that is used by the second one. If the data formats of both machines are not compatible, Perl converts the data to an appropriate format.

   You can find an output file of an imaginary machine in `machine1.txt`. Take a look at it. Write a Perl filter that reads the data, skips all lines beginning with #, swaps row 2 and 3 of the data lines, and writes the data to a new file `machine2.txt`. Print the screen as you write it to the file. Your script should immediately stop processing when the line "`# END DATA`" is found.

   Sample Output:

```
$ cat machine1.txt
# ---------------------
# machine: Lago7
# date: January 10, 2000
# time: 09:56:34
# ---------------------
#
# running 67h 46m 23s
# software revision: 3.65
# last check: 06/27/99
# next check: 06/27/00
#
# BEGIN DATA
135.45 65.3 456.67
142.87 65.4 442.17
129.83 65.4 433.51
132.62 65.3 416.65
141.88 65.3 471.19
128.43 65.2 463.38
131.65 65.3 426.57
139.89 65.3 422.18
138.33 65.4 484.41
145.46 65.3 466.69
132.67 65.4 432.57
149.89 65.4 433.56
# END DATA
#
# Warranty:
# Hadrian will not give any warranty
# for the correctness of the preceding
# data, since ....
```

```
$ lab0707.plx
135.45 456.67 65.3
142.87 442.17 65.4
129.83 433.51 65.4
132.62 416.65 65.3
141.88 471.19 65.3
128.43 463.38 65.2
131.65 426.57 65.3
139.89 422.18 65.3
138.33 484.41 65.4
145.46 466.69 65.3
132.67 432.57 65.4
149.89 433.56 65.4

$ cat machine2.txt
135.45 456.67 65.3
142.87 442.17 65.4
129.83 433.51 65.4
132.62 416.65 65.3
141.88 471.19 65.3
128.43 463.38 65.2
131.65 426.57 65.3
139.89 422.18 65.3
138.33 484.41 65.4
145.46 466.69 65.3
132.67 432.57 65.4
149.89 433.56 65.4
```

Suggested Solution:

lab0707.plx

```
 1 #!/usr/bin/perl -w
 2
 3 $infile = "machine1.txt";
 4 $outfile = "machine2.txt";
 5
 6 open IN, $infile or die "Open failed: $!";
 7 open (OUT, ">$outfile" or die "Open failed: $!";
 8
 9 while ($line = <IN>) {
10
11   next if $line =~ /^\s*#/; # Line is Comment
12   last if $line eq "# END DATA"; # Line is end of data
13
14   # split up each data line and swap row 2 and row 3
15   chomp $line;
16   ($x,$y,$z) = split / /, $line;
17
18   # print out the result for the next machine
19   print OUT "$x $z $y\n";
20   print "$x $z $y\n";
21 }
```

8. (Optional) Write a script that reads in a list of hosts passed on the command line and then displays the number of users logged into each host.

   Use the command: `finger @host` to obtain a list of the users logged into the host. Use a filehandle to process the command.

   Remove the first line or two from the command so that they are not included in your count.

   Sample output:

```
$ lab0708.plx host1 host2 host3
The servers currently have the following load.
host1 has 146 users
host2 has 82 users
host3 has 115 users
```

Suggested solution:

lab0708.plx

```
 1 #!/usr/bin/perl -w
 2
 3 print "The servers currently have the following load.\n";
 4
 5 foreach $host (@ARGV){
 6   open USERS, "finger \@$host |" or die "Open failed: $!\n";
 7
 8   @users = <USERS>;
 9
10   shift @users;
11   shift @users;
12
13   $count = @users;
14
15   print "$host has $count users\n";
16
17   close USERS;
18 }
```