

PERL PROGRAMMING ON UNIX

Student Workbook

PERL PROGRAMMING ON UNIX

Jeff Howell

Published by ITCourseware, LLC, 7245 South Havana Street, Suite 100, Centennial, CO 80112

Contributing Authors: Brandon Caldwell, Jim McNally, and Rob Roselius

Editor: Rick Sussenbach

Assistant Editor: Jan Waleri

Special thanks to: Many instructors whose ideas and careful review have contributed to the quality of this workbook, including Jimmy Ball, Tait Cyrus, Phil Gardner, Denise Geller, Julie Johnson, Roger Jones, Roger Kobylarczyk, Michael Naseef, Evans Nash, Bill Parrette, Richard Raab, George Trujillo, and Todd Wright, and the many students who have offered comments, suggestions, criticisms, and insights.

Copyright © 2009 by ITCourseware, LLC. All rights reserved. No part of this book may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying, recording, or by an information storage retrieval system, without permission in writing from the publisher. Inquiries should be addressed to ITCourseware, LLC, 7245 South Havana Street, Suite 100, Centennial, CO 80112. (303) 302-5280.

All brand names, product names, trademarks, and registered trademarks are the property of their respective owners.

CONTENTS

Chapter 1 - Course Introduction	9
Course Objectives	10
Course Overview	12
Using the Workbook	13
Suggested References	14
Chapter 2 - Overview of Perl	17
What is Perl?	18
Running Perl Programs	20
Sample Program	22
Another Sample Program	24
Yet Another Example	26
Labs	28
Chapter 3 - Perl Variables	31
Three Data Types	32
Variable Names and Syntax	34
Variable Naming	36
Lists	38
Scalar and List Contexts	40
The Repetition Operator	42
Labs	44
Chapter 4 - Arrays and Hashes	47
Arrays	48
Array Functions	50
The foreach Loop	52
The @ARGV Array	54
The grep Function	56
Array Slices	58
Hashes	60
Hash Functions	62
Scalar and List Contexts Revisited	64
Labs	66

Chapter 5 - Quoting and Interpolation	69
String Literals	70
Interpolation	72
Array Substitution and Join	74
Backslashes and Single Quotes	76
Quotation Operators	78
Command Substitution	80
Here Documents	82
Labs	84
Chapter 6 - Operators	87
Perl Operators	88
Operators, Functions and Precedence	90
File Test Operators	92
Assignment Operator Notations	94
The Range Operator	96
Labs	98
Chapter 7 - Flow Control	101
Simple Statements	102
Simple Statement Modifiers	104
Compound Statements	106
The next, last, and redo Statements	108
The for Loop	110
The foreach Loop	112
Labs	114
Chapter 8 - I/O: Input Operations and File I/O	117
Overview of File I/O	118
The open Function	120
The Input Operator <>	122
Default Input Operator	124
The print Function	126
Reading Directories	128
Labs	130

Chapter 9 - Regular Expressions	133
Pattern Matching Overview	134
The Substitution Operator	136
Regular Expressions	138
Special Characters	140
Quantifiers (*, +, ?, { })	142
Assertions (^, \$, \b, \B)	144
Labs	146
Chapter 10 - Advanced Regular Expressions	149
Substrings	150
Substrings in List Context	152
RE Special Variables	154
RE Options	156
Multi-line REs	158
Substituting with an Expression	160
Perl RE Extensions	162
Labs	164
Chapter 11 - Subroutines	167
Overview of Subroutines	168
Passing Arguments	170
Private Variables	172
Returning Values	174
Labs	176
Chapter 12 - References	179
References	180
Creating References	182
Using References	184
Passing References as Arguments to Subroutines	186
Anonymous Composers	188
The Symbol Table	190
Labs	192

Chapter 13 - Complex Data Structures	195
Two-Dimensional Arrays in Perl	196
Anonymous Arrays and Anonymous Hashes	198
Arrays of Arrays	200
Arrays of References	202
A Hash of Arrays	204
A Hash of Hashes	206
And So On	208
Labs	210
Chapter 14 - Packages and Modules	213
Packages	214
BEGIN and END Routines	216
require vs. use	218
Modules	220
The bless Function	222
Labs	224
Chapter 15 - Introduction to Object-Oriented Programming in Perl	227
What is Object-Oriented?	228
Why Use Object-Oriented Programming?	230
Classes, Objects, and Methods in Perl	232
Inheritance, the "is-a" Relationship	234
Containment, the "has-a" Relationship	236
Overloaded Operators	238
Destructors	240
Labs	242
Chapter 16 - Binary Data Structures	245
Variable-Length (Delimited) Fields	246
Variable vs. Fixed	248
Handling Binary Data	250
The pack Function	252
The unpack Function	254
The read Function	256
C Data Structures	258
Labs	260

Chapter 17 - Multi-tasking with Perl	263
What are Single and Multi-tasking?	264
UNIX Multi-tasking Concepts	266
Process Creation with fork	268
Program Loading with exec	270
File Descriptor Inheritance	272
How UNIX Opens Files	274
One-Way Data Flow - Pipes	276
Example	278
Example (Cont'd)	280
Final Result - Page Viewing	282
Labs	284
Chapter 18 - Sockets Programming in Perl.....	287
Clients and Servers	288
Ports and Services	290
Berkeley Sockets	292
Data Structures of the Sockets API	294
Socket System Calls	296
Generic Client/Server Models	298
A Client/Server Example	300
A Little Web Server	302
Labs	304
Appendix A - The Perl Distribution	307
Where Can You Get Perl?	308
How Do You Build Perl?	310
What Gets Created and Installed?	312
Differences Between Platforms	314
Appendix B - The Perl Debugger	317
Overview of the Perl Debugger	318
Debugger Commands	320
Non-Debugger Commands	322
Listing Lines	324
Single Stepping	326

Setting and Clearing Breakpoints	328
Modifying the Debugger	330
The -w and -D Flags	332
Labs	334
 Solutions	337
 Index	401

CHAPTER 1 - COURSE INTRODUCTION

COURSE OBJECTIVES

- * Write scripts, programs, and reusable modules in Perl.
- * Make effective use of Perl modules written by others.

This course is intended for experienced programmers with user-level skills in the UNIX environment. Many programs will be written during the class. The lecture topics and lab exercises concentrate on the Perl language and its features, with less emphasis on application-specific subjects.

The particular applications that you will design, write, and work on during this class are intended to demonstrate the use of Perl's capabilities and services provided in UNIX programming environments. The example programs and solutions to the exercises will provide guidance when you get back to work and begin development on real projects.

COURSE OVERVIEW

- * **Audience:** This is a programming course designed for software development professionals. You will write many programs in this class.
- * **Prerequisites:** User-level skills in the UNIX environment are required. The ability to write programs in a high-level structured language such as C, C++, shell, or Java is strongly recommended. This course does not teach basic computer programming concepts.
- * **Classroom Environment:**
 - UNIX software development system with one workstation per student.
 - Perl 5.6 or higher installed.
 - Access to Perl documentation, such as:
 - Perl man pages readable with the **man** command.
 - Perl documentation in pod format, readable with **perldoc**.
 - Web access, to be able to read documentation at ***www.perl.com***.

USING THE WORKBOOK

This workbook design is based on a page-pair, consisting of a Topic page and a Support page. When you lay the workbook open flat, the Topic page is on the left and the Support page is on the right. The Topic page contains the points to be discussed in class. The Support page has code examples, diagrams, screen shots and additional information. **Hands On** sections provide opportunities for practical application of key concepts. **Try It** and **Investigate** sections help direct individual discovery.

In addition, there is an index for quick look-up. Printed lab solutions are in the back of the book as well as on-line if you need a little help.

The Topic page provides the main topics for classroom discussion.

The Support page has additional information, examples and suggestions.

JAVA SERVLETS

THE SERVLET LIFE CYCLE

- * The servlet container controls the life cycle of the servlet.
- When the first request is received, the container loads the servlet class.
- The container uses a separate thread to call the `init()` method.
- The container calls the `destroy()` method.

Topics are organized into first (★), second (➤) and third (■) level points.

- As with Java's `finalize()` method, don't count on this being called.
- * Override one of the `init()` methods for one-time initializations, instead of using a constructor.
- The simplest form takes no parameters.

```
public void init() { ... }
```

- If you need to know container-specific configuration information, use the other version.

```
public void init(ServletConfig config) { ... }
```

- Whenever you use the `ServletConfig` approach, always call the superclass method, which performs additional initializations.

```
super.init(config);
```

CHAPTER 2

SERVLET BASICS

The Support page has additional information, examples and suggestions.

Hands On:

Add an `init()` method to your `Today` servlet that initializes along with the current date:

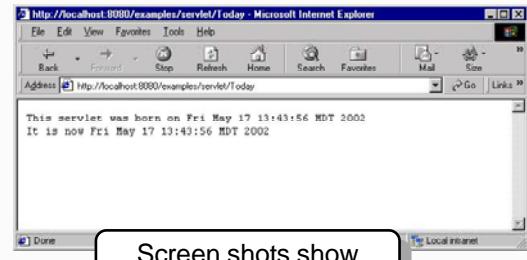
`Today.java`

```
...
public class Today extends GenericServlet {
    private Date bornOn;
    public void service(ServletRequest request,
                        ServletResponse response) throws ServletException, IOException
    {
        ...
        ...
        ...
    }
}
```

Code examples are in a fixed font and shaded. The on-line file name is listed above the shaded area.

Callout boxes point out important parts of the example code.

The `init()` method is called when the servlet is loaded into the container.



Screen shots show examples of what you should see in class.

Pages are numbered sequentially throughout the book, making lookup easy.

SUGGESTED REFERENCES

Blank-Edelman, David N. 2000. *Perl for System Administration*. O'Reilly & Associates, Sebastopol, CA. ISBN 1565926099.

Hall, Joseph N. 1998. *Effective Perl Programming: Writing Better Programs with Perl*. Addison-Wesley, Reading, MA. ISBN 0201419750.

Macdonald, John, Jon Orwant, and Jarkko Hietaniemi. 1999. *Mastering Algorithms with Perl*. O'Reilly & Associates. ISBN 1565923987.

Schwartz, Randall L., Tom Phoenix and brian d. foy. 2008. *Learning Perl, 5th Edition*. O'Reilly Media, Inc., Sebastopol, CA. ISBN 0596520107.

Srinivasan, Sriram. 1997. *Advanced Perl Programming*. O'Reilly & Associates, Sebastopol, CA. ISBN 1565922204.

Wall, Larry, Tom Christiansen, and Nathan Torkington. 1998. *Perl Cookbook*. O'Reilly & Associates, Sebastopol, CA. ISBN 1565922433.

Wall, Larry, Tom Christiansen, and Jon Orwant. 2000. *Programming Perl, 3rd Edition*. O'Reilly & Associates, Sebastopol, CA. ISBN 0596000278.

Brown, Martin. 2001. *DeBugging Perl*. Osborne/McGraw-Hill, Berkeley, CA. ISBN 0072126760.

www.perl.com

This is the central web site for the Perl community.

www.activestate.com

Originally for Win32 Perl, ActiveState has broadened its horizons.

www.oreilly.com

Books, of course, and other Perl resources.

CHAPTER 2 - OVERVIEW OF PERL

OBJECTIVES

- * Define what Perl is.
- * Run Perl programs various ways.
- * Get a feel for Perl programming by reviewing example Perl programs.

WHAT IS PERL?

- ※ Perl is an interpreted programming language.
- ※ Its strengths are:
 - Scanning and processing large amounts of text efficiently.
 - Formatting and printing reports easily.
 - Providing tremendous built-in parsing functionality to reduce the need for ancillary code in a program.
 - Great extensibility via libraries and modules.
- ※ Perl contains the following:
 - Most of the statements and operators of C.
 - Many of the library functions provided in a C environment.
 - Built-in functionality of **awk**, such as automatic record input and field splitting (but much more of this than **awk**).
 - Extended regular expressions found in **awk**, with Perl extensions.
 - The file manipulation capabilities of shell scripts.
 - Process control capabilities of UNIX.

There is a lot of confusion about whether Perl is a compiled or interpreted language. Most interpreted languages perform poorly because of the time required to interpret and then run each line of a program in turn, as in shell scripts. Perl runs interpretively; the programs you run are written and run as source Perl code. Perl also has the capability to modify itself programmatically as it runs. Perl runs much faster than most other interpreters, however, due to its capability to precompile and tokenize major portions of a program before it actually runs the script.

Since much of the functionality of Perl maps directly onto C function calls, it makes sense for Perl to simply call the C library routines as it needs them. This compilation of Perl code dramatically speeds up Perl programs, making them almost as fast as compiled C programs.

There is not, however, a “Perl compiler” which can create an actual binary executable image (file) which can then be loaded and run like **cat** or **grep**; there is a way to fake Perl into doing something like this, but it’s not efficient or easy (we will talk about this later in the course). Perl programs always run under the **perl** executable on your host computer. There’s no way around that.

This means that while Perl programs are fast to execute and very fast to develop (in comparison to C, for example), always remember that when you distribute Perl programs, you are distributing clear text source code which the user can read, modify, and easily break.

RUNNING PERL PROGRAMS

- ※ Perl programs can be run several ways.
- ※ The simplest way is to put the program in a file, say **hello.pl**, and type:

```
$ perl hello.pl
```

(The dollar sign here is the shell prompt.)

- ※ This will invoke the Perl interpreter, **perl**, which will interpret the program in the file **hello.pl**.
 - The **.pl** extension is not required; it's just a convention that some Perl programmers follow.
- ※ Some other ways to run Perl programs are described on the facing page.

Several ways to run Perl programs:

1. For short one-line programs, you can use the **-e** command line option to write and execute the program on the command line (we show the shell prompt as a \$):

```
$ perl -e 'print "Hello, world!\n";'
```

2. Use an editor to type the program into a file, say **hello.pl**. Then type the following command at the shell prompt :

```
$ perl hello.pl
```

The one-liner above might look like this inside the file **hello.pl**:

```
print "Hello, world!\n";
```

3. Use an editor to type a Perl program into a file, but insert a first line into the file so it looks like this:

```
#!/usr/bin/perl  
print "Hello, world!\n";
```

Then, make the program executable:

```
$ chmod +x hello.pl
```

Now you can run the program by just typing its name.

```
$ ./hello.pl
```

This method of running Perl programs (or any interpreted programs, such as **awk** and shell programs) works on systems that support the concept of “interpreter files.” The FIRST TWO characters in the file must be **#!**, and the pathname of the interpreter must follow on the same line.

SAMPLE PROGRAM

```
dirld.pl
#!/usr/bin/perl
# File: dirld.pl
foreach (@ARGV)
{
    system ("ls -ld $_[0]") if -d;
}
```

- * Run the above program *dirld.pl* with file names as command line arguments.

```
$ dirld.pl *
```

- * Note some Perl syntax basics in the code:
 - Semicolon at the end of the line.
 - Braces around the **foreach** loop body.
 - Comments start with a #.

Consider the program *dirld.pl*.

The program is intended to be run with command-line arguments that are filenames. For each file that is a directory, do an **ls -ld** command on that file.

1. The second line begins a **foreach** loop that comprises the entire program. For each iteration of the loop, the special variable **\$_** will be set (as a string) to the next command-line argument. The command-line arguments are stored in the special array called **@ARGV**.
2. The loop consists of a single statement, the third line of the program. The statement is a conditional, with the if syntactically FOLLOWING a call to the Perl **system** function. The **system** function, if called, will pass the string **ls -ld \$_** to the operating system, which will execute the string as a command.

Under what condition will the **system** function be called? Well, consider the pseudo-code for the program:

```
for each command line argument:  
    set the variable $_ to the string value of the argument  
    if the file named in $_ is a directory  
        call the system() function to execute "ls -ld $_"
```

Or alternatively:

```
for each command line argument:  
    set the variable $_ to the string value of the argument  
    call the system() function to execute "ls -ld $_"  IF the file named in $_  
    is a directory
```

Try It:

A more verbose version of this program, *dirldv.pl*, follows. It takes less advantage of Perl's automatic actions, and is more explicit:

```
dirldv.pl  
#!/usr/bin/perl  
#File: dirldv.pl  
foreach $file (@ARGV) {  
    if (-d $file) {  
        system ("ls -ld $file");  
    }  
}
```

The second line uses **foreach \$file**; it specifies the variable name that the foreach loop is to assign the argument strings to, instead of implicitly assigning them to the special variable **\$_**. The variable name is **\$file**. The dollar sign in the **\$file** variable name is part of the variable name. It is not a dereference operator as used in shell programming and in **awk**. The test for “directoryness” on the third line is similar to that of the shell's test command. If the **-d** test operator is not given an argument variable containing a file name, it will test the contents of the **\$_** variable, assuming that the contents are the name of a file that might be a directory.

ANOTHER SAMPLE PROGRAM

```
menu.pl
#!/usr/bin/perl
#File: menu.pl

for (;;)
{
    system("clear");    # Clear the screen
    print "\n\n";
    print "l. List Files\n";
    print "p. Print Working Directory Name\n";
    print "w. List Users Logged On\n";
    print "e. Exit Menu\n";
    print "\n\n";
    print "Enter choice: ";
    $choice = <STDIN>; chomp $choice;
    last if $choice eq "e";

    if      ($choice eq "l") { $command = "ls";   }
    elsif   ($choice eq "p") { $command = "pwd";  }
    elsif   ($choice eq "w") { $command = "who";  }
    else { next; }

    print "\n\n";
    system($command);
    print "\n\n";

    print 'Hit <Enter> to continue:';

    <STDIN>;      # Wait for the enter key
}
```

This program, **menu.pl**, displays a menu of user-selectable actions. The program executes an infinite loop using the C-style **for(;;)** construct, waiting for the user to type a choice. After performing the action chosen by the user, the program clears the screen and redisplays the menu. The program terminates when the user selects the exit menu choice.

The **system("clear")** function tells the operating system to execute the **clear** command, just as if it had been typed on the keyboard.

The line:

```
$choice = <STDIN>; chomp $choice;
```

reads one line into the variable **\$choice** from the file identified by the file handle **STDIN**, using the input operator **<>**. The **chomp** function chops off the newline character that was read from **STDIN** and stored into **\$choice**.

The line:

```
last if $choice eq "e" ;
```

exits the **for** loop using the Perl control flow statement **last** if the user chose **e**. If we had not chomped the newline we would have said:

```
last if $choice eq "e\n" ;
```

After setting the variable **\$command** based on **\$choice**, the **system** function is called once again to execute the command.

Finally, the line:

```
<STDIN>;
```

simply reads a line from the standard input. The intent is to wait for the user to hit the **<Enter>** key before clearing the screen and redisplaying the menu. The stand-alone **<STDIN>** actually reads one line and discards it.

Note that a semicolon, **;**, must terminate each simple statement. A compound statement (a sequence of simple statements, enclosed in braces) is not terminated with a semicolon.

YET ANOTHER EXAMPLE

```
dgrep.pl
#!/usr/bin/perl
# File: dgrep.pl

# A descending grep program:
# 1) open an input pipe
# 2) find all files in the current directory
# 3) and subdirectories
# 4) print the names of text files
# 5) that contain words
# 6) specified on the command line

open(PIPE, "find . -print|") or die "Pipe open failed\n";
@files = <PIPE>;
close(PIPE);

foreach $word (@ARGV)
{
    foreach $filename (@files)
    {
        chomp $filename;
        if ( -T $filename )
        {
            open(THE, $filename) || die "File open failed\n";
            while ($line = <THE>)
            {
                if ($line =~ /$word/)
                {
                    print "$filename: $line\n";
                }
            }
            close(THE);
        }
    }
}
```

The line:

```
open(PIPE, "find . -print |") or die "Pipe open failed" ;
```

creates the **PIPE** filehandle, connected to output of the UNIX **find** command, and each time that the input operator `<>` operates on the filehandle, output is read from the command. The line:

```
@files = <PIPE>;
```

reads the entire output of the find command into the array **@files**, with one line of output assigned to each element of the array. If the line had said:

```
$files = <PIPE>;
```

then only one line would have been read from the pipe into the scalar variable **\$files**. This is because of the difference between an *array context* and a *scalar context* which we will discuss later in great detail.

The **@ARGV** array holds the command-line arguments. The **foreach** loop assigns the value of the arguments to **\$word**, one argument for each iteration of the loop.

The next **foreach** loop assigns the array elements of **@files** to the scalar variable **\$filename**, one element at a time. This is doing exactly what the outer loop is doing with the **@ARGV** array. The **\$filename** variable will thus be set equal to a pathname (one line of output from the find command) for each iteration of the loop.

The **chomp** operator chops off the newline character at the end of the string in **\$filename**, and then the **-T** test checks to see if the file named in **\$filename** is a text file. If so, then a file handle named **THIS** is created, the file is opened and **THIS** is associated with the open file.

Next the **while** loop sets the value of the scalar variable **\$line** equal to each line of the open file by using the input operator `<>` on **THIS**, which reads a line from the file each time it is executed. Note that **\$line** is a scalar, so the context of the `<THIS>` operation is a scalar context.

The line:

```
if ($line =~ /$word/)
```

checks the **\$line** variable for the occurrence of the pattern stored in **\$word**, which is the current command line argument. The operator `=~` is a pattern bind operator, and the the `/$/word/` is a regular expression.

Finally, note that in the **print** statement, even though the **\$filename** and **\$line** variables are enclosed in double quotes, their values are still used. Perl calls this *interpolation*. A shell would call it *weak-quoting*.

A few words about filehandles. In Perl, the **open** function opens a file and associates it with a filehandle. After a file is opened, all file operations on the file are done by referencing the filehandle. For example:

```
open (BOOKFILE, "books") ; # Open file "books", associate with file handle BOOKFILE
$book = <BOOKFILE>; # Use input operator <> to read a line
...
close(BOOKFILE);
```

LABS

- ① What is the significance of the special variable `$_`?
(Solution: *underscore*)
- ② Write a program to echo its command line arguments.
(Solution: *args.pl*)
- ③ What would happen in the program *dgrep.pl* if we attempted to use the automatically set special variable `$_` in each of the **foreach** loops instead of the explicit variables `$word` and `$filename`?
(Solutions: *underscore2*)
- ④ Modify the program *menu.pl* so that it will prompt the user for a command and then execute as a command whatever line the user types in.
(Solution: *cmd.pl*)
- ⑤ Write a program that opens a file then uses a **while** loop to read each line from the file into a scalar variable and print the variable to the screen, prepending each line with its line number. For the line numbers, initialize a variable to 1 before the loop, then add one to it each iteration.
(Solution: *liner.pl*)

CHAPTER 7 - FLOW CONTROL

OBJECTIVES

- * Create and use simple statements with modifiers.
- * Create and use compound statements.
- * Use the four loops and their control statement.

SIMPLE STATEMENTS

- ※ Expressions are constructed from variables, literals, functions, and operators.
- ※ Every expression has a value.
- ※ A simple statement is an expression terminated by a semicolon.
- ※ Expressions result in side-effects which are what get the work done!

```
$svar = -1;  
  
@avar = ('x') x 10;  
  
($s,$m,$h,$md,$M,$Y,$wd,$j,$dst) = localtime(time);  
  
open (DA_NUMBERS, "da_books") ||  
      die "Can't open da_books\n";
```


SIMPLE STATEMENT MODIFIERS

- * An expression can be followed by a modifier:

```
E1 if E2;  
E1 unless E2;  
E1 while E2;  
E1 until E2;
```

- The modifier expression **E2** is evaluated before **E1**.

```
die "Need args\n" unless @ARGV;  
print "Got args\n";  
  
print "Made it to 19\n" if $debugging;  
  
print HANDLE $count % 9 while ++$count <= 80;
```

- * Under what condition would the last example not print?

The modifier is always evaluated before the expression that it is modifying, so Perl provides a do-block construct that allows you to execute a block of statements once before the modifier is evaluated. A *block* is a sequence of statements enclosed in curly braces.

```
do { E1;
     E2;
     E3;
     ...
} [ while E; | until E; ]
```

This construct is still considered to be a simple expression because there are things that you can do in compound statements that you can't do with a do-block construct.

So, the differences between

```
E1 while E;      and      do { E1;
                               E2;
                               E3;
                               ...
} while E;
```

are:

1. You can have multiple statements in a block.
2. The block is executed before E is evaluated, thus ensuring that the statement (or statements) executes at least once.

```
$count = 5;
do { print $count-- } while $count;
```

is almost equivalent to:

```
$count = 6;
print $count while $count--;
```

(How do they differ?)

```
do {
    print STDOUT "Important information\n";
    print KEEPIT "Important information\n";
} if $logging;
```

COMPOUND STATEMENTS

- ※ Compound statements control the flow of execution of a program.
- ※ The facing page shows all of the forms of compound statements.
- ※ Things to note:
 - The statements controlled by a compound statement must be in a block, even if there is only one statement. The curly braces are mandatory.
 - Several styles can be used

```
if ($flag) { $flag = 0; }
```

```
if ($flag) {
    $flag = 0;
}
```

```
if ($flag)
{
    $flag = 0;
}
```

```
if ($flag)      # Nope, not in Perl!
    $flag = 0;   # if() must be followed by a block.
```

But:

```
$flag = 0 if $flag; # A simple statement
                    # with a modifier.
```

Compound Statements (items shown in [] are optional):

```
if (E) block [ elseif (E) block ] ... [ else block ]  
unless (E) block [ else block ]  
[LABEL:] while (E) block [ continue block ]  
[LABEL:] until (E) block [ continue block ]  
[LABEL:] for (E1; E2; E3) block          # Any of E1,E2,E3 optional  
[LABEL:] foreach [ var ] (array) block  # Use $_ if var is omitted  
[LABEL:] block continue block
```

An expression is false if its value is **0** or an empty string.

THE NEXT, LAST, AND REDO STATEMENTS

- ※ The looping behavior of compound loops can be changed.
- ※ Loops can be partially skipped, exited prematurely, or restarted.
- ※ Within the block of a compound loop, you can use any of the following loop control statements:

```
next [LABEL]
last [LABEL]
redo [LABEL]
```

- ※ They respectively:
 - Start the next iteration of the loop, skipping the remaining statements in the block.
 - Terminate the loop immediately, and resume execution at the next statement after the loop block.
 - Redo the block without re-evaluating the loop conditional.
- ※ If the **LABEL** is omitted from a loop control statement, then the innermost enclosing loop will be affected.

Often, the **next** loop control statement is used when the loop is complicated, and another level of nesting would be too deep.

```
open (NUMBERS, "numfile");
while (<NUMBERS>) {
    next if ($_ < 0);    # Skip negative numbers
    ...    # Process positive numbers
}
```

The most commonly used loop control statement is **last**. This breaks out of a loop and resumes execution at the next statement after the loop. Assume we are to process a file till we reach the first negative number.

```
open (NUMBERS, "numfile");
while (<NUMBERS>) {
    last if ($_ < 0);    # Bail if we see a negative
    ...    # Process positive numbers
}
# <- resume execution here
```

An example of the use of a *LABEL* follows. This example opens a file named **control** which contains names of files, one per line, that contain numbers, one per line. The outer loop, labeled **CONTROL**, iterates through the numbers files, while the inner loop processes numbers (just prints them) from each numbers file until it finds a negative number in a file, in which case the inner loop closes the file and quits to the outer loop to open another numbers file.

```
control.pl
#!/usr/bin/perl
#File: control.pl

open (CONTROL, "control") || die "Control file open failed\n";

CONTROL:
while ($numfile = <CONTROL>) {    # Get name of next number file
    print $numfile;
    chomp $numfile;
    open (NUMBERS, $numfile) || die "File open failed\n";
    while (<NUMBERS>) {
        if ($_ < 0) { # Next file if negative
            close (NUMBERS);
            next CONTROL;
        }
        print $_;    # Process positive numbers
    }
    close (NUMBERS);
}
close (CONTROL);
```

THE FOR LOOP

- * The Perl **for** loop is much like the **for** loop in C, C++, Java, etc..

```
for (E1; E2; E3) block
```

- *E1* is evaluated only once, before the loop begins.
- *E2* is evaluated before each iteration: If true, execute *block*, else terminate loop.
- *E3* is evaluated after each iteration.

```
for ($i = 0; $i < $num_loops; $i++)  
{  
    # do something $num_loops times  
    ...  
}
```


THE FOREACH LOOP

- * The **foreach** loop iterates through the items in a list.

```
foreach [$var] (list) block
```

- The individual values of *list* are assigned to **\$var**, one for each loop iteration.
- If **\$var** is omitted, **\$_** is used.
- **\$var** is local and is set back to its value after the loop.
- If *list* is an actual array or a list of assignable scalars (and not a list or a list value resulting from an expression), then modifying **\$var** inside the loop changes the corresponding array value.

```
@num = (1,2,3,4,5,6,7,8,9,10);  
foreach $n (@num)  
{  
    $n = "odd" if $n % 2;  
}  
print "@num\n";
```

- * **foreach** is a synonym of **for**, and so their syntax is interchangeable.

Try It:

```
candy.pl
#!/usr/bin/perl
# File: candy.pl

@candy = ('hershey', 'jelly bean', 'licorice', 'red hot');

$_ = "\$ value before loop\n";
print;    # Print $_ before loop
print "@candy\n";
foreach (@candy) {
    if ($_ eq "licorice") {
        $_ = "mint";    # I don't like licorice
    }
}
print "@candy\n";
print;    # Print $_ after loop
```

Output:

```
$_ value before loop
hershey jelly bean licorice red hot
hershey jelly bean mint red hot
$_ value before loop
```

Note that for a list-driven loop, the keywords **for** and **foreach** are syntactically equivalent. In the above example, we could say

```
for (@candy) {
```

or we could use an intermediate variable like

```
for $var (@array) {
```

or,

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

It's up to you which one to use!

LABS

- 1** Modify the program *control.pl* so that negative numbers are merely skipped over, instead of causing processing on that file to stop.
(Solution: *skipnegr.pl*)

- 2** Rewrite the following as simple statements with modifiers. Identify E1, E2, etc.
(Solution: *mods.pl*)

a. `if ($x == $y) { print "Equal\n"; }`

b. `while (<>) {
 print int, "\n";
}`

c. `unless (open(FT, "bigfile")) {
 warn "Can't open 'bigfile'";
}`

- 3** Write a program that creates an array of strings of varying lengths. Using a **foreach** loop on the array, append the string **short** to each element whose length is less than a number passed in on the command line.

(Solution: *lesser.pl*)

- 4** Print the number of lines in each file in the current directory, and the total number of lines of all files. Verify your output with the UNIX **wc(1)** command. For those of you who know **printf**, make your output look exactly like '**wc -l***'.

(Solution: *lentext.pl*)

CHAPTER 16 - BINARY DATA STRUCTURES

OBJECTIVES

- * Compare variable-length and fixed-length record processing in Perl.
- * Understand the uses of the **pack** and **unpack** functions.
- * Write programs that use the **read**, **pack**, and **unpack** functions.

VARIABLE-LENGTH (DELIMITED) FIELDS

- ✳ Many applications operate on data records that contain variable-length delimited fields.
- ✳ Examples
 - Our *cars* file: whitespace delimited.
 - */etc/passwd*: colon delimited.
- ✳ The length of any given field can vary from record to record.
- ✳ The **join** and **split** functions handle data of this type.
- ✳ **join** joins together the elements of a list into a string, putting a specified delimiter string between each element in the joined string.
- ✳ **split** breaks a string into a list of elements; the string must contain a delimiter between each desired list element.

Try It:

```
splitter
#!/usr/bin/perl
# File: splitter

# Typical passwd line:
# bhi:x:200:1:bhi:/home/bhi:/bin/ksh

@ARGV != 1 && die "Usage: splitter id\n";
open (PASSWD, "/etc/passwd");
while (<PASSWD>) {
    if (/^$ARGV[0]/) {
        ($name, $passwd, $uid, $gid, $comment, $dir, $shell) = split(':');
        print "$name $uid $dir\n";
    }
}
```

```
joiner
#!/usr/bin/perl
# File: joiner

# Join together the command line arguments, separated by double dashes (-).
# Put the result into a file called "cmdcode".

$all = join('-', @ARGV);
open(CODE, ">cmdcode") || die " Can't open cmdcode: $!\n";
print CODE $all, "\n";
close(CODE);
```

VARIABLE VS. FIXED

- ※ Advantages of variable-length data:
 - No need to worry about fields overflowing.
 - Flexible format can be easier for users and programmers.
- ※ Disadvantages of variable-length data:
 - Delimiters take storage and must be scanned for.
 - Binary data could be mistaken for a delimiter.
 - Direct (random) access is not possible.
- ※ Advantages of fixed-length data:
 - No time or storage wasted on delimiters.
 - Records can be located by calculation and seeking.
 - Binary data presents no problem.
- ※ Disadvantages of fixed-length data:
 - Maximum size of fields must be allocated, so pad bytes waste space.
 - More meticulous code required to process fixed fields.

HANDLING BINARY DATA

- ✳ **join** and **split** have analogous fixed-length data functions called **pack** and **unpack**.
 - **unpack** creates a list from a scalar, as does **split**. The elements to be extracted out of the scalar are in known fixed positions in the scalar.
 - **pack** creates a scalar from a list, where the elements of the list are packed into the scalar in fixed positions.
- ✳ **pack** and **unpack** primary features are:
 - Fixed-position, fixed-length strings are created (packed) or broken out (unpacked).
 - ASCII-to-binary (pack) and binary-to-ASCII (unpack) conversions can take place.
- ✳ **pack** and **unpack** have many uses:
 - Creating records that are formatted as fixed-length fields.
 - Extracting fields from records that are formatted as fixed-length fields.
 - Writing and reading binary data files (ON SAME MACHINE!!!).
 - Sending/receiving binary structures to/from Perl function calls, such as **socket**.

THE PACK FUNCTION

```
packex
#!/bin/perl
# File: packex (Pack Examples)

# The hash %persons uses last names as keys, and the values
# are first name (then space) then salary.
# Pack as salary, first name, last name in: a long integer
# and two 15 character fields.

# The C program scans.c will read the file we create.

%persons = ('smith', 'slim 50000',
            'jones', 'joe 52000',
            'curtis', 'curt 49000',
            'morgan', 'mike 51000');

open(SALARYS, '>salarys') || die "Can't open salarys:$!\n";

while ( ($last, $value) = each %persons ) {
    $record = pack("L A15 A15",
                  reverse(split(' ', $value)), $last);
    print SALARYS $record;
}
close(SALARYS);
```

Synopsis of the **pack** function:

```
pack (TEMPLATE, LIST)
```

pack takes the array or list passed as **LIST** and packs its values into a binary structure which is returned in a scalar. **TEMPLATE** is a sequence of characters that specifies the order and type of values to be packed into the structure. The characters allowable for use in **TEMPLATE** are:

- A** An ascii string, will be space padded
- a** An ascii string, will be null padded
- c** A signed char value
- C** An unsigned char value
- s** A signed short value
- S** An unsigned short value
- i** A signed integer value
- I** An unsigned integer value
- l** A signed long value
- L** An unsigned long value
- n** A short in "network" order
- N** A long in "network" order
- f** A single-precision float in the native format
- d** A double-precision float in the native format
- p** A pointer to a string
- P** A pointer to a structure (fixed-length string)
- v** A short in "VAX" (little-endian) order
- V** A long in "VAX" (little-endian) order
- x** A null byte
- X** Back up a byte
- @** Null fill to absolute position
- u** A uuencoded string
- b** A bit string (ascending bit order, like **vec()**)
- B** A bit string (descending bit order)
- h** A hex string (low nybble first)
- H** A hex string (high nybble first)

A letter may optionally be followed by a number which provides a repeat count for the letter. The letter and repeat count together form a field specifier. Field specifiers may be separated by white space, which will be ignored.

A count causes **pack** to consume that many values from **LIST**, unless the letter is **a** or **A**. In those cases, only one value (string) is taken from the list, but it is full of space padded up to the count if necessary. If the repeat count is an *****, then the remaining values in the list will be packed according to the letter type being repeated.

THE UNPACK FUNCTION

- * **unpack** takes a string that is packed with contiguous bytes representing binary values, and unpacks the values into an array.
- * It understands the same list of **TEMPLATE** letters used by **pack**.

unpackex

```
#!/bin/perl
# File: unpackex

# Unpack the records in the 'salarys' file.

open(SALARYS, 'salarys') || die "Can't open
salarys:$!\n";

while ($record = <SALARYS>) {
    ($salary, $first, $last) = unpack("L A15 A15",
        $record);
    print "$first $last makes $salary dollars a
year!\n";
}
close(SALARYS);
```


THE READ FUNCTION

- * The program on the previous page doesn't work correctly because the <> operator needs to know what a *line* is.
- * The **read** function will read a specified number of bytes from a file.

readex

```
#!/bin/perl
# File: readex

# Unpack the records in the 'salarys' file.

open(SALARYS, 'salarys') || die "Can't open salarys:$!\\n";

while (read(SALARYS, $record, 34)) { # ONLY CHANGE
    ($salary, $first, $last) = unpack("L A15 A15",
        $record);
    print "$first $last makes $salary dollars a
          year!\\n";
}
close(SALARYS);
```


C DATA STRUCTURES

- ⌘ Another use for **pack** and **unpack** is to translate to the C language.
- ⌘ The C structures on the facing page map onto Perl's needs as a packed structure containing the same data.
- ⌘ Perl would create a **sockaddr_in** structure as follows:

```
$sockaddr_in = pack('S n a4 x8', $AF_INET, $port, $IP_Addr);
```

- ⌘ This maps directly onto the C structure needed by the sockets libraries.
- ⌘ This is used routinely when Perl must make a system call into the kernel (or some other UNIX process) which needs to see native machine data types.

The basic data structure for a socket descriptor contains the following information:

- Address Family
- Socket Type
- Local IP Address
- Remote IP Address
- Local Port
- Remote Port

The generic address structure used by the sockets toolkit is the C **sockaddr** structure, shown below (using 4.3BSD notation).

```
struct sockaddr {  
    u_short sa_family;  
    char    sa_data[14];  
};
```

The address structure for a specific protocol family is based on the generic **sockaddr**. The AF_INET family implementation is shown below.

```
struct sockaddr_in {  
    u_short sin_family; /* type of address */  
    u_short sin_port; /* port number */  
    u_long  sin_addr; /* ip address */  
    char    s_a_zero[8]; /* unused */  
};
```

Note that the **sockaddr_in** structure fully defines an endpoint, as discussed previously.

LABS

- 1** Write a program that reads the file *numbers* and prints out the message contained in the file. The file is packed with a 5-character string, a signed short, a 7-character string and a single precision floating-point number.
(Solution: *luck.pl*)

- 2** Write a program that packs the length of a string in front of the string itself. Use an unsigned short integer to store the binary integral length. This is the way that many languages store strings, unlike C, which does not store the lengths of strings, but rather requires that strings are terminated by a null byte. Write several such records to a file.
(Solution: *stringlen.pl*)

- 3** Using *stringlen*, write a program that uses the **read** function to read the strings from the file into variables in your program. You may wish to read them into an array. Note that your program will need to do two reads to get each string.
(Solution: *readlen.pl*)

- 4** Write a program that verifies that the ‘A’ specifier in the template for the **unpack** operator causes any padded spaces to be stripped, but the ‘a’ (lowercase) specifier does not.
(Solution: *padder.pl*)

