1. **The vi editor: edit files, scripts**

2 mode: insert and input mode

Cmd: vi [filename]

Cmd switch to input mode: a, …

:wq, ZZ // save and quit

:q! quit not save

* Cmd delete, change

dd // delete line

x // delete character

u // undo

* Copy, paste

p // paste to the right of cursor

yy // copy current line

* Search

/findtext

n // repeat search in same direction

* Movement

G // go to the end of file

1. **Permissions and ownership**

ls –l // see permission

// “-” is a file, “d” is a directory, “l” is a link,

// ‘u’ user, ’ g ‘ group, ‘o’ other, ‘a’ all

// ‘+’ add, ‘-‘ remove, ‘=’ set permission

Ex: chmod u=rwx filename

Setting ownership

chgrp –c <group> <files>

chown –c <user> <files>

chown –c <user>:<group> <files>

1. **Finding files and directories**

find <path> <options> -print

find / -name abc -print

Finding in file

grep [option ] <find text> <path>

grep -c ‘abc’ ./

pwd // print working directory

cd [option] // change directory

mkdir name

rmdir name // remove if empty

rm –r dir // remove dir and everything in it

ls [option] // list file

-a // list all

-R // list all and every file in subdirectories

-l // detail

Wildcards: expanded by the shell before passed to the command

\* //

? //

[chars] //

[!chars] // ls ![b-c]\*

I/O redirection

command < file

command > file

command >> file // append to the end of file

command > file1 2 > file2 // write output to file1, error to file2

Shell and user variable

common environmental variables: …

echo $name // display variable ‘s content

variable=value

Archiving file & directories

tar –czvf name.tar \* // compress current directories

tar –xzvf name.tar \* // decompress tar file

gzip

gzip file

gunzip file

zcat file // see zip file

Job control:

bg

fg

& ampersand

Processes

ps // view processes

kill <pid> // kill process

top //

Others

cal

cat

clear

cp -rv flie1 file2

date

df // view amount of freee disk space

du // view used disk space for specified directory

echo //

id //

less //

ln

mv

netsat

nslookup

ping

rm

telnet

ssh

touch

traceroute

mount

unmounts

whoami

The shell is a special program used as an interface between the user and the heart of the UNIX operating

system, a program called the kernel,

The shell is a utility program that starts up when you log on. It allows users to interact with the kernel by

interpreting commands that are typed either at the command line or in a script file.

A shell script is much

like a batch file: It is a list of UNIX commands typed into a file, and then the file is executed

The Bourne shell is the standard UNIX shell, and is used to administer the system. Most of the system

administration scripts, such as the rc start and stop scripts and shutdown are Bourne shell scripts, and when in

single user mode, this is the shell commonly used by the administrator when running as root. This shell was

written at AT&T and is known for being concise, compact, and fast. The default Bourne shell prompt is the

dollar sign ($).

The C shell was developed at Berkeley and added a number of features, such as command line history,

aliasing, built−in arithmetic, filename completion, and job control. The C shell has been favored over the

Bourne shell by users running the shell interactively, but administrators prefer the Bourne shell for scripting,

because Bourne shell scripts are simpler and faster than the same scripts written in C shell. The default C shell prompt is the percent sign (%).

When the script starts, the .cshrc file is read first and executed, so that anything set within that file will

become part of your script. You can prevent the .cshrc from being read into your script by using the –f (fast)

option to the C shell program. This option is written as

#!/bin/csh –f

**9.2.2 Reading User Input**

The $< Variable. To make a script interactive, a special C shell variable is used to read standard input into a

variable. The $< symbol reads a line from standard input up to but not including the newline, and assigns the

line to a variable[6].

%echo –n “string “

The string is echoed to the screen. The –n option causes the echo command to suppress

the newline at the end of the string. On some versions of echo, use a \c at the end of the

string to suppress the newline; e.g., echo hello\c.

% set name = ( $name )

The string consists of one word. There are not two words, so by using a subscript of [2],

the shell complains that the Subscript is out of range. Subscripts start at one. To create a wordlist, the string is enclosed in parentheses. An array is created. The string is broken up into a list of words and assigned to the variable name.

**9.2.3 Arithmetic**

1 % @ sum = 4 + 6

echo $sum

10

2 % @ sum++

echo $sum

11

3 % @ sum += 3

echo $sum

14

4 % @ sum−−

echo $sum

13

5 % @ n = 3+4

@: Badly formed number

**Echo (–x) and Verbose (–v)**

csh −v practice

The contents of the C shell script are displayed. Variable and command substitution lines

are included so that you can see how echo and verbose differ.

1.

The –v option to the csh command causes the verbose feature to be enabled. Each line of

the script is displayed as it was typed in the script, and then the line is executed.

2.

The –x option to the csh command enables echoing. Each line of the script is displayed

after variable and command substitution are performed, and then the line is executed.

Since this feature allows you to examine what is being replaced as a result of command

and variable substitution, it is used more often than the verbose option.

**Example 9.72**

(Echo and Verbose)

1 % cat practice

#!/bin/csh

echo Hello $LOGNAME

echo The date is 'date'

set echo

echo Your home shell is $SHELL

unset echo

echo Good−bye $LOGNAME

% chmod +x practice

2 % practice

Hello ellie

The date is Sun May 26 12:25:16 PDT 2001

−−> echo Your login shell is /bin/csh

−−> Your login shell is /bin/csh

−−> unset echo

Good−bye ellie

**9.2.5 Command Line Arguments**

Positional parameters are number variables. The scriptname is assigned to $0, and any words

following the scriptname are assigned to $1, $2, $3 … ${10}, ${11}, and so on. $1 is the first command line

argument. In addition to using positional parameters, the C shell provides the argv built−in array

C shell array subscripts start at 1.$argv[1] $argv[2]…The first argument, second argument,

etc. $argv[\*] All arguments. $argvAll arguments. $#argv The number of arguments. $argv[$#argv] The last

argument.

(The Script)

#!/bin/csh –f

# The greetings script

# This script greets a user whose name is typed in at the

# command line.

1 echo $0 to you $1 $2 $3

2 echo Welcome to this day 'date | awk '{print $1, $2, $3}''

3 echo Hope you have a nice day, $argv[1]\!

4 echo Good–bye $argv[1] $argv[2] $argv[3]

**FORMAT**

if ( expression ) then

command

command

else if ( expression ) then

command

command

else

command

endif

if ( expression ) single command

(In the Script: Checking for Arguments)

1 if ( $#argv != 1 ) then

2 echo "$0 requires an argument"

3 exit 1

4 endif

**FORMAT**

switch (variable)

case constant:

commands

breaksw

case constant:

commands

breaksw

endsw

**FORMAT**

foreach variable (wordlist)

commands

end

while

command

end

**The repeat Command.** The repeat command takes two arguments, a number and a command. The command

is executed that number of times.

**Example 9.99**

% repeat 3 echo hello

hello

hello

hello

**AWK**

nawk 'pattern {action}' filename

**Using More Than One Field Separator**

% nawk –F'[ :\t]' '{print $1, $2, $3}' employees

The NR Variable. The number of each record is stored in awk's built−in variable, NR. After a record has been

processed, the value of NR is incremented by one.

**Example 5.12**

% cat employees

Tom Jones 4424 5/12/66 543354

Mary Adams 5346 11/4/63 28765

Sally Chang 1654 7/22/54 650000

Billy Black 1683 9/23/44 336500

% nawk '{print NR, $0}' employees

1 Tom Jones 4424 5/12/66 543354

2 Mary Adams 5346 11/4/63 28765

3 Sally Chang 1654 7/22/54 650000

4 Billy Black 1683 9/23/44 336500

**PERL**

Perl is an acronym, short for *P*ractical *E*xtraction and *R*eport *L*anguage. It was designed for writing programs in the UNIX environment. In short, Perl is as powerful as C but as convenient as awk, sed, and shell scripts.

1: #!/usr/local/bin/perl

2: $inputline = <STDIN>;

3: print( $inputline );

In this example, the smaller units of information are $inputline, =,

<STDIN>, and ; called a *token*

$inputline is an example of a *scalar variable*. In Perl, a scalar variable can store one piece of information.

<STDIN>, represents a line of input from the *standard input file*

**Assigning a Value to a Scalar Variable**

$var = 42;

$name = "inputdata";

the library function chop, which gets rid of the closing newline character

that is part of the input line you entered. The chop library function is described in the

following section, "The chop Library Function."

if (*expr\_1*) {

*statement\_block\_1*

} elsif (*expr\_2*) {

*statement\_block\_2*

} elsif (*expr\_3*) {

*statement\_block\_3*

...

} else {

*statement\_block\_n*

unless (*expr\_1*) {

*statement\_block\_1*

}

The while statement loops *while* its conditional expression is true.

The until statement loops *until* its conditional expression is true (that is, it loops

as long as its conditional expression is *false*).

}

**What Is a Scalar Value?**

Basically, a *scalar value* is one unit of data. This unit of data can be either a number or a

chunk of text.

**Integer Scalar Values**

integers in Perl actually are represented as floating-point

numbers. This means that an integer scalar value is actually a special kind of floatingpoint

scalar value.

**Integer Scalar Value Limitations**

**Floating-Point Scalar Values**

8e+01 = 8\*10

8e-01 = 8/10

**Using Octal and Hexadecimal Notation**

Base 8 notation, or *octal*

$result = 047; // This assigns 47 octal, or 39 decimal, to $result.

Base 16 notation, or *hexadecimal* (sometimes shortened to *hex*)

$result = 0x1f; // This assigns 1f hexadecimal, or 31 decimal, to $result.

**Character Strings**

C programmers should be advised that character strings

in Perl do not contain a hidden null character at the

end of the string. In Perl, null characters can appear

anywhere in a string.

**Using Double-Quoted Strings**

**Single-Quoted Strings**

There are two differences between double-quoted strings and single-quoted strings.

The first difference is that scalar variables are replaced by their values in double-quoted strings but not in single-quoted strings. The following is an example:

$string = "a string";

$text = "This is $string"; # becomes "This is a string"

$text = 'This is $string'; # remains 'This is $string'

The second difference is that the backslash character, \, does not have a special meaning in single-quoted strings. This means that the statement assigns the following string to $text:

$text = 'This is a string.\n';

The \ character is special in only two instances for single-quoted strings. The first is when you want to include a single-quote character ' in a string.

$text = 'This string contains \', a quote character';

The preceding line of code assigns the following string to $text:

This string contains ', a quote character

The second instance is to escape the backslash itself.

$text = 'This string ends with a backslash \\';

The preceding code line assigns the following string to $text:

This string ends with a backslash \

**Interchangeability of Strings and Numeric Values**

$string = "43";

$number = 28;

$result = $string + $number; // $result = 71

If a string contains characters that are not digits, the string is converted to 0 when used in an integer context.

For example:

$result = "hello" \* 5;

# this assigns 0 to $result, since "hello" becomes0

The arithmetic operators \*\*, and unary negation

The *exponentiation operator*, \*\*

$x = 2 \*\* 4; // $x = 16

2 \*\* -5 # this is the fraction 1/32

5 \*\* 2.5 # this is 25 \* the square root of 5

**Unary Negation**

The *unary negation operator* is a - character in front of a single value. (This distinguishes it from the subtraction operator, which appears between two values.) It is equivalent to multiplying the value by -1, as illustrated by this example:

- 5; # identical to the integer -5

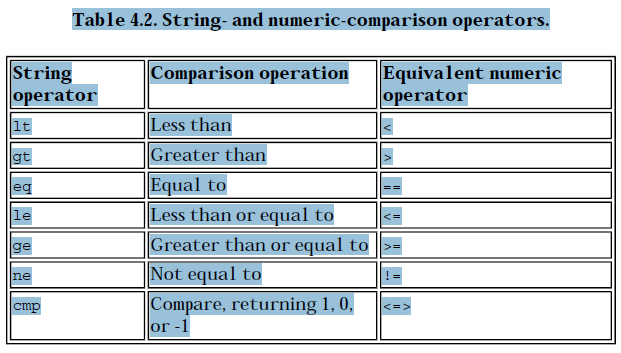
- $y; # equivalent to $y \* -1

The integer-comparison operators <=>

The <=> operator is a special case. Unlike the other integer comparison operators, <=> returns one of three values:

* 0, if the two values being compared are equal
* 1, if the first value is greater
* -1, if the second value is greater

The string-comparison operators eq, ne, lt, gt, le, ge, and cmp



Perl compares strings by determining their places in an alphabetical order. For example, the string aaa is less than the string bbb, because aaa appears before bbb when they are

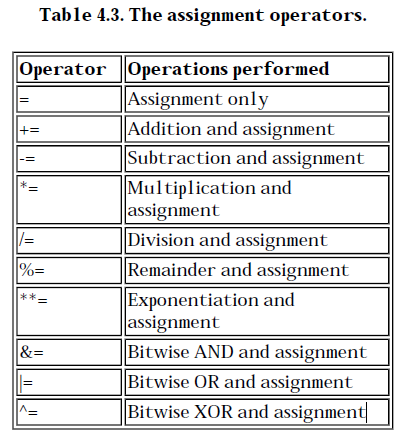
sorted alphabetically

**Comparison and Floating-Point Numbers**

Be very careful when you use floating-point numbers in comparisons, because round-off errors might affect your results.

**Logical Operators as Subexpressions: page 131**

The assignment operators \*\*= and .=



The autoincrement operator ++

The autodecrement operator –-

$result = $a++;

$result2 = ++$a;

**Using Autoincrement With Strings**

If a string value contains only alphabetic characters, the ++ operator can be used to

"add one" to a string. In other words, the operator replaces the last character of the

string with the next letter of the alphabet. The following is an example:

$stringvar = "abc";

$stringvar++;

Here, $stringvar now contains abd.

Note that this works only with ++, not –

If the last letter of the string is z or Z, ++ converts this letter to a or A, and then "adds

one" to the second-to-last character of the string:

$stringvar = "abz";

$stringvar++; # $stringvar now contains "aca"

As in numeric operations, incrementing a string that ends in 9 carries over to the next

character of the string. This works regardless of whether the next character is a digit

or alphabetic character.

$stringvar = "bc999";

$stringvar++; # $stringvar now contains "bd000"

The string-concatenation operator

* l The . operator, which *concatenates* (joins together) two strings

$newstring = "potato" . "head"; // potatohead

* l The x operator, which repeats a string

$newstring = "t" x 5; // ttttt

* l The .= operator, which combines concatenation and assignment

$a = "be";

$a .= "witched"; # $a is now "bewitched"

The string-repetition operator x

The comma operator ,: It guarantees that a particular part of an expression (the part before the ,) is evaluated first.

$var1 += 1, $var2 = $var1; // $var1 += 1; $var2 = $var1;

The conditional operator (? and : together)

$result = $var == 0 ? 14 : 7;

**The Order of Operations: Precedence, associativity, parentheses**

**Introducing Lists**

A *list* is a sequence of scalar values enclosed in parentheses. The following is a simple example of a list: (1, 5.3, "hello", 2)

()

This list also is called an *empty list*.

**Storing Lists in Array Variables**

Perl enables you to store lists in special variables designed for that purpose. These

variables are called *array variables* (or *arrays* for short)

@array = (1, 2, 3);

Because Perl uses @ and $ to distinguish array variables

from scalar variables, the same name can be used in an

array variable and in a scalar variable. For example:

$var = 1;

@var = (11, 27.1, "a string");

Normally, you won't want to use the same name in both

an array and a scalar variable, because this is confusing.

To assign the first element of the array variable @array to the scalar

variable $scalar, use the following statement:

$scalar = $array[0];

The character string

"$var[0]"

substitutes the value of the first element of @var in the string.

**Using Brackets and Substituting for Variables**

To substitute the value of $var and keep the [0] as it is, you must use one of the following:

"${var}[0]"

"$var\[0]"

"$var" . "[0]"

**Using List Ranges**

(1..5) = (1, 2, 3 ,4 ,5)

**Copying from One Array Variable to Another**

@result = @original;

**Using Array Variables in Lists**

@list1 = (2, 3, 4);

@list2 = (1, @list1, 5); // (1, 2, 3, 4, 5)

**Substituting for Array Variables in Strings**

Perl does not leave spaces if you pass an array variable to print:

@array = (1, 2, 3);

print (@array, "\n"); // 123

When the Perl interpreter sees the array variable inside the string, it substitutes the

values of the list assigned to the array variables, and leaves a space between each pair of elements. For example:

print ("@array\n"); // 1 2 3

**Assigning to Scalar Variables from Array Variables**

@array = (5, 7);

($var1, $var2) = @array;

Additional elements in an array, if they exist, are ignored. For example:

@array = (5, 7, 11);

($var1, $var2) = @array;

Here, 5 is assigned to $var1, 7 is assigned to $var2, and 11 is not assigned to anything.

If there are more scalar variables than elements in an array variable, the excess scalar variables are assigned the null string

**Retrieving the Length of a List**

@array = (1, 2, 3);

$scalar = @array;

Note that the following two statements are not equivalent:

$scalar = @array;

($scalar) = @array;

In the first statement, the length of the list in @array is assigned to $scalar. In the second statement, the first element of @array is assigned to $scalar.

It is always important to remember that $scalar and($scalar) are not the same thing. $scalar is a scalar variable, and ($scalar) is a one-element list containing $scalar.

**Using Array Slices**

@array = (1, 2, 3, 4);

@subarray = @array[1,2]; // (2, 3)

**Using List Ranges in Array-Slice Subscripts**

@subarray = @array[0..19];

This assigns the first 20 elements of @array to @subarray.

$endrange = 19;

@subarray = @array[0..$endrange];

You can also use the list stored in an array variable to define an array slice. Listing 5.8 shows how this works.

@range = (1, 2, 3);

@subarray = @array[@range];

**Assigning to Array Slices**

@array[0,1] = ("string", 46);

Here, the first two elements of the array @array become string and 46, respectively.

@array[0..$endrange] = (1, 2, 3, 4);

**Overlapping Array Slices**

@array[1,2,3] = @array[2,3,4];

The Perl interpreter has no problem with this statement because it copies the list stored

in @array[2,3,4] into a temporary location (invisible to you) before assigning it to

@array[1,2,3].

**Reading an Array from the Standard Input File**

$array = <STDIN>; // read a line

If you assign <STDIN> to an array variable instead of a scalar variable, the Perl interpreter reads in all of the data from the standard input file at once and assigns it.

@array = <STDIN>;

Ctrl+D produces a special character that indicates end of file; when the Perl interpreter sees this, it knows that there is no more input.

**Array Library Functions**

**Sorting a List or Array Variable:** The library function sort sorts the elements of an array in alphabetical order and returns the sorted list.

retlist = sort (array);

**Reversing a List or Array Variable**

The library function reverse reverses the order of the elements of a list or array variable, and returns the reversed list.

**Using chop on Array Variables**

@list = ("rabbit", "12345", "quartz");

chop (@list);

After chop is called, the list stored in @list is

("rabbi", "1234", "quart")

**Creating a Single String from a List**

$string = join (@array);

**Splitting a String into a List**

$string = "words::separated::by::colons";

@array = split(/::/, $string);

**Opening a File**

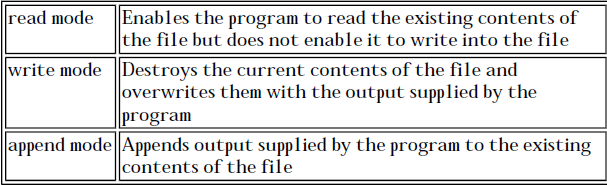
Before you can read from or write to a file, you must first open the file. This operation tells the operating system that you are currently accessing the file and that no one else can change it while you are working with it.

open (filevar, filename);

It's a good idea to use all uppercase letters for your file variable names. This makes it easier to distinguish file variable names from other variable names and from reserved words.

open(FILE1, "file1");

**The File Mode**

****

By default, open assumes that a file is to be opened in read mode. To specify write mode, put a > character in front of the filename that you pass to open, as follows:

open (OUTFILE, ">/u/jqpublic/outfile");

To specify append mode, put two > characters in front of the filename

Here are a few things to remember when opening files:

* l When you open a file for writing, any existing contents are destroyed.
* l You cannot read from and write to the same file at the same time.
* l When you open a file in append mode, the existing contents are not destroyed, but you cannot read the file while writing to it.

**Checking Whether the Open Succeeded**

* If open returns a nonzero value, the file has been opened successfully.
* If open returns 0, an error has occurred.

**Reading from a File**

To read from a file, enclose the file variable associated with the file in angle brackets (< and >), as follows:

$line = <MYFILE>; // read a line

**Terminating a Program Using die**

die (message);

When the Perl interpreter executes the die function, the program terminates immediately and prints the message passed to die.

**Reading into Array Variables**

Perl enables you to read an entire file into a single array variable. To do this, assign the file variable to the array variable, as follows:

@array = <MYFILE>;

**Writing to a File**

print OUTFILE ("Here is an output line.\n");

**The Standard Output File Variable**

By default, the print statement sends output to the standard output file, which means that it sends the output to the file associated with STDOUT. As a consequence, the following statements are equivalent:

print ("Here is a line of output.\n");

print STDOUT ("Here is a line of output.\n");

**Redirecting Standard Input and Standard Output**

#!/usr/local/bin/perl

$line = <STDIN>;

print ($line);

myperlprog <file1 >outfile

**The Standard Error File:** You can write anything you want to STDERR at any time

print STDERR ("File file1 opened successfully.\n");

**Closing a File**

When you are finished reading from or writing to a file, calling the library function close.

close (filevar);

Note that you do not have to call close when you are finished with a file: Perl automatically closes the file when the program terminates or when you open another file using a previously defined file variable

**Using Command-Line Arguments as Values**

C programmers should take note that the first element of @ARGV, unlike argv[0] in C, does not contain the name of the program. In Perl, the first element of @ARGV is the first command-line argument.

To get the name of the program, use the system variable

$0, "System Variables."

**ARGV and the <> Operator**

In Perl, the <> operator actually contains a hidden reference to the array @ARGV.

1. When the Perl interpreter sees the <> for the first time, it opens the file whose

name is stored in $ARGV[0].

2. After opening the file, the Perl interpreter executes the following library

function:

shift(@ARGV);

This library function gets rid of the first element of @ARGV and moves every other

element over one. This means that element x of @ARGV becomes element x-1.

3. The <> operator then reads all of the lines of the file opened in step 1.

4. When the <> operator exhausts an input file, the Perl interpreter goes back to

step 1 and repeats the cycle again.

1: #!/usr/local/bin/perl

2:

3: $searchword = $ARGV[0];

4: print ("Word to search for: $searchword\n");

5: shift (@ARGV);

6: $totalwordcount = $wordcount = 0;

7: $filename = $ARGV[0];

8: while ($line = <>) {

9: chop ($line);

10: @words = split(/ /, $line);

11: $w = 1;

12: while ($w <= @words) {

13: if ($words[$w-1] eq $searchword) {

14: $wordcount += 1;

15: }

16: $w++;

17: }

18: if (eof) {

19: print ("occurrences in file $filename: ");

20: print ("$wordcount\n");

21: $totalwordcount += $wordcount;

22: $wordcount = 0;

23: $filename = $ARGV[0];

24: }

25: }

26: print ("total number of occurrences: $totalwordcount\n");

$ program6\_14 single file1 file2

Word to search for: single

occurrences in file file1: 1

occurrences in file file2: 1

total number of occurrences: 2

$

**Introduction**

A *pattern* is a sequence of characters to be searched for in a character string. In Perl,

patterns are normally enclosed in slash characters:

/pattern/

**The Match Operators**

The result of the =~ operation is one of the following:

* A nonzero value, or true, if the pattern is found in the string
* 0, or false, if the pattern is not matched

The !~ operator is similar to =~, except that it checks whether a pattern is not matched.

$var =~ /abc/;