UNIX AND SHELL PROGRAMMING

UNIX AND SHELL PROGRAMMING

[As per Choice Based Credit System (CBCS) scheme]
(Effective from the academic year 2015 -2016)
SEMESTER – III

Subject Code 15CS35	**IA Marks 20**
Number of Lecture Hours/Week 04	**Exam Marks 80**
Total Number of Lecture Hours 50	**Exam Hours 03**

CREDITS – 04

Module -1 **10 Hours**

Introduction, Brief history. Unix Components/Architecture. Features of Unix. The UNIX Environment and UNIX Structure, Posix and Single Unix specification. The login prompt. General features of Unix commands/ command structure. Command arguments and options. Understanding of some basic commands such as echo, printf, ls, who, date, passwd, cal, Combining commands. Meaning of Internal and external commands. The type command: knowing the type of a command and locating it. The man command knowing more about Unix commands and using Unix online manual pages. The man with keyword option and whatis. The more command and using it with other commands. Knowing the user terminal, displaying its characteristics and setting characteristics. Managing the nonuniform behaviour of terminals and keyboards. The root login. Becoming the super user: su command. The /etc/passwd and /etc/shadow files. Commands to add, modify and delete users.

Module -2 **10 Hours**

Unix files. Naming files. Basic file types/categories. Organization of files. Hidden files. Standard directories. Parent child relationship. The home directory and the HOME variable. Reaching required files- the PATH variable, manipulating the PATH, Relative and absolute pathnames. Directory commands – pwd, cd, mkdir, rmdir commands. The dot (.) and double dots (..) notations to represent present and parent directories and their usage in relative path names. File related commands – cat, mv, rm, cp, wc and od commands. File attributes and permissions and knowing them. The ls command with options. Changing file permissions: the relative and absolute permissions changing methods. Recursively changing file permissions. Directory permissions.

Module -3 **10 Hours**

The vi editor. Basics. The .exrc file. Different ways of invoking and quitting vi. Different modes of vi. Input mode commands. Command mode commands. The ex mode commands. Illustrative examples Navigation commands. Repeat command. Pattern searching. The search and replace command. The set, map and abbr commands. Simple examples using these commands. The shells interpretive cycle. Wild cards and file name generation. Removing the special meanings of wild cards. Three standard files and redirection. Connecting commands: Pipe. Splitting the output: tee. Command substitution. Basic and Extended regular expressions. The grep, egrep. Typical examples involving different regular expressions.

Module -4 **10 Hours**

Shell programming. Ordinary and environment variables. The .profile. Read and readonly commands. Command line arguments. exit and exit status of a command. Logical operators for conditional execution. The test command and its shortcut. The if, while, for and case control statements. The set and shift commands and handling positional parameters. The here (<<) document and trap command. Simple shell program examples. File inodes and the inode structure. File links – hard and soft links. Filters. Head and tail commands. Cut and paste commands. The sort command and its usage with different options. The umask and default file permissions. Two special files /dev/null and /dev/tty.

Module -5 **10 Hours**

Meaning of a process. Mechanism of process creation. Parent and child process. The ps command with its options. Executing a command at a specified point of time: at command. Executing a command periodically: cron command and the crontab file.. Signals. The nice and nohup commands. Background processes. The bg and fg command. The kill command. The find command with illustrative example. Structure of a perl script. Running a perl script. Variables and operators. String handling functions. Default variables - $_ and $. – representing the current line and current line number. The range operator. Chop() and chomp() functions. Lists and arrays. The @- variable. The splice operator, push(), pop(), split() and join(). File handles and handling file – using open(), close() and die () functions.. Associative arrays – keys and value functions. Overview of decision making loop control structures – the foreach. Regular expressions – simple and multiple search patterns. The match and substitute operators. Defining and using subroutines.

MODULE 1: UNIX ARCHITECTURE AND COMMAND USAGE GENERAL PURPOSE UTILITIES ESSENTIAL SYSTEM ADMINISTRATION

MODULE 1: UNIX ARCHITECTURE AND COMMAND USAGE

1.1 Introduction

- UNIX is an operating system.
- An operating system is a set of programs that act as a link between the computer and the user.
- The operating system (OS) manages the resources of a computer.
- Examples of computer resources are: CPU, RAM, disk memory, printers, displays, keyboard, etc.

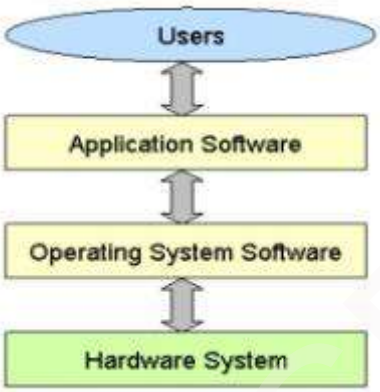


Figure 1.1 : Operating System

- UNIX OS allows complex tasks to be performed with a few keystrokes.
- UNIX OS doesn't tell or warn the user about the consequences of the command.
- Unix was originally developed in 1969 by a group of AT&T employees Ken Thompson, Dennis Ritchie, Douglas McIlroy, and Joe Ossanna at Bell Labs.
- There are various Unix variants available in the market.
- Solaris Unix, AIX, HP Unix and BSD are a few examples.
- Linux is also a flavor of Unix which is freely available.

1.2 Unix Architecture (Components)

• The UNIX operating system (OS) consists of a kernel layer, a shell layer , an application layer & files.

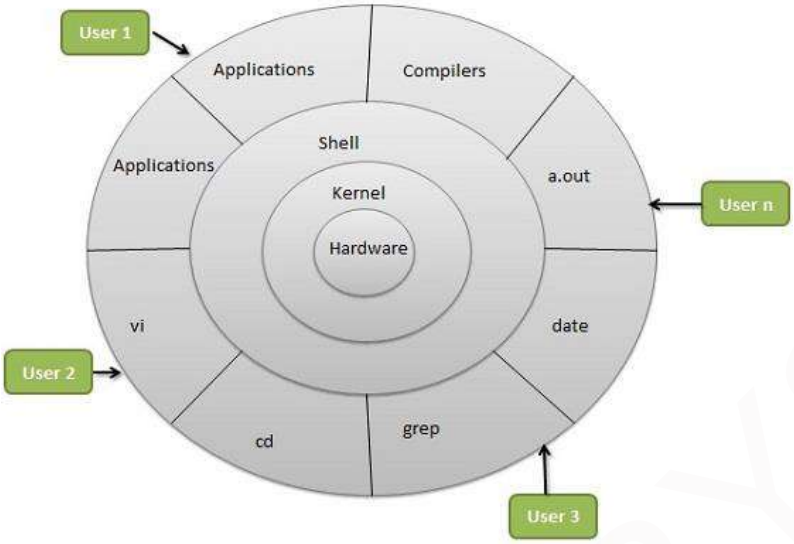


Figure 1.2 : Unix Architecture

Kernel

• The kernel is the heart of the operating system.
• It interacts with the machine's hardware.
• It is a collection of routines written in C.
• It is loaded into memory when the system is booted.
• Main responsibilities:
    1) Memory management
    2) Process management (using commands: kill, ps, nohup)
    3) File management (using commands: rm, cat, ls, rmdir, mkdir)
• To access the hardware, user programs use the services of the kernel via system calls.

System Call

➢ A system call is a request for the operating system to do something on behalf of the user's program.
➢ The system calls are functions used in the kernel itself.
➢ To the programmer, the system call appears as a normal C function call.
➢ UNIX system calls are used to manage the file system and control processes.
➢ Example: read(), open(), close(), fork(), exec(), exit()

• There can be only one kernel running on the system.

Shell

• The shell interacts with the user.
• The shell is a command line interpreter (CLI).
• Main responsibilities:
    1) interprets the commands the user types in and
    2) dispatches the command to the kernel for execution
• There can be several shells in action, one for each user who's logged in.
• There are multiple shells that are used by the UNIX OS.
• For example: Bourne shell (sh), the C shell (csh), the Korn shell (ksh) and Bourne Again shell (bash).

Application

• This layer includes the commands, word processors, graphic programs and database management programs.

File

• A file is an array of bytes that stores information.
• All the data of Unix is organized into files.
• All files are then organized into directories.
• These directories are further organized into a tree-like structure called the filesystem.

1.3 Features of Unix

1) Multiuser

• UNIX is a multiprogramming system.
• Multiple users can access the system by connecting to points known as terminals.
• Several users can run multiple programs simultaneously on one system.

2) Multitasking

• UNIX is a multitasking system.
• A single user can also run multiple tasks concurrently.
• For example:
At the same time, a user can
→ edit a file
→ print another file one on the printer
→ send email to a friend and
→ browse www
• The kernel is designed to handle a user's multiple needs.
• In a multitasking environment, a user sees one job running in the foreground; the rest run in the background.
• User can switch jobs between background and foreground, suspend, or even terminate them.

3) Pattern Matching

• UNIX has very sophisticated pattern matching features.
• Regular Expressions are a feature of UNIX.
• Regular Expressions describe a pattern to match, a sequence of characters, not words, within a line of text.

4) Portable

• UNIX can be installed on many hardware platforms.
• Unix operating system is written in C language, hence it is more portable than other operating systems.

5) UNIX Toolkit

• UNIX offers facility to add and remove many applications as and when required.
• Tools include
→ general purpose tools
→ text manipulation tools
→ compilers/interpreters
→ networked applications and
→ system administration tools

6) Programming Facility

• The UNIX shell is also a programming language; it was designed for programmer, not for end user.
• It has all the necessary ingredients, like control structures, loops and variables, that establish powerful programming language.
• This features are used to design shell scripts – programs that can also invoke UNIX commands.
• Many of the system's functions can be controlled and automated by using these shell scripts.

7) Documentation

• The principal on-line help facility available is the man command, which remains the most important references for commands and their configuration files.
• Apart from the man documentation, there's a vast ocean of UNIX resources available on the Internet.

1.4 POSIX and Single Unix Specification

• The Portable Operating System Interface (POSIX) is a family of standards specified by IEEE for maintaining compatibility between operating systems.
• Two of the most important standards from POSIX are:
    1) POSIX.1 – Specifies the C application program interface – the system calls (Kernel)
    2) POSIX.2 – Deals with the Shell and utilities
• In 2001, a joint initiative of X/Open and IEEE resulted in the unification of two standards.
• This is the Single UNIX Specification, Version 3 (SUSV3).
• The "Write once, adopt everywhere" approach to this development means that once software has been developed on any POSIX machine it can be easily ported to another POSIX compliant machine with minimum or no modification.

1.5 The Login Prompt

How to log in

• Have your userid (user identification) and password ready.
• Type your userid at the login prompt, then press ENTER. Your userid is case sensitive, so be sure you type it exactly as your system administrator has instructed.
• Type your password at the password prompt, then press ENTER. Your password is also case sensitive.
• If you provide the correct userid and password, then you will be allowed to enter into the system.
• While the prompt($) is displayed, you can type a command (say date).

```
login: kumar
kumar's password: *****
Last login: Sun Jun 14 09:32:32 2017 from 162.61.164.73
$ date
Thu Jun 25 08:30:19 MST 2017
```

1.6 Command Structure

• UNIX commands take the following general
< form: verb [options] [arguments] >
where verb is the command name that can take a set of optional options and one or more optional arguments.
• Commands, options and arguments have to be separated by spaces or tabs to enable the shell to interpret them as words.
• A contiguous string of spaces and tabs together is called a whitespace.
• The shell compresses multiple occurrences of whitespace into a single whitespace.

1.6.1 Options

• An option is preceded by a minus sign (-) to distinguish it from filenames.
• Example:
$ ls –l // -l option list all the attributes of the file note
• There must not be any whitespaces between – and l.
• Options are also arguments, but given a special name because they are predetermined.
• Options can be normally combined with only one – sign.
thus $ ls –l –a –t is same as $ ls –lat
• Because UNIX was developed by people who had their own ideas as to what options should look like, there will be variations in the options.
• Some commands use + as an option prefix instead of -.

1.6.2 Filename Arguments

• Many UNIX commands use a filename as argument so that the command can take input from the file.
• If a command uses a filename as argument, it will usually be the last argument, after all options.
• Example:

```
ls -lat chap01 chap02 chap03 # Multiple filenames as arguments
cp file1 file2 file3 dest_dir
rm file1 file2 file3
```

• The command with its arguments and options is known as the command line.
• This line can be considered complete only after the user has hit [Enter].
• The complete line is then fed to the shell as its input for interpretation and execution.

1.6.3 Exceptions

• There are some commands that don't accept any arguments.
• There are also some commands that may or may not be specified with arguments.
• For Example:
The ls command can run
→ without arguments (ls)
→ with only options (ls –l)
→ with only filenames (ls f1 f2), or
→ using a combination of both (ls –l f1 f2).
• There are some commands compulsorily take options (cut).
• There are some commands which can take an expression as an argument, or a set of instructions as argument. Ex: grep, sed

1.7 Understanding of Some Basic Commands

1.7.1 echo

- This command can be used to display a message on the terminal.
- Example:

```
$ echo "Welcome to UNIX \n"
Welcome to UNIX
```

- This command can be used with following escape characters:

"\a"	Audible Alert (Bell)
"\b“	Back Space
"\f“	Form Feed
"\n“	New Line
"\r“	Carriage Return
"\t“	Horizontal Tab
"\v“	Vertical Tab
"\\“	Backslash

1.7.2 printf

- This command can be used to display a message on the terminal.
- This command can be used with following escape characters:

"\a"	Audible Alert (Bell)
"\b“	Back Space
"\f“	Form Feed
"\n“	New Line
"\r“	Carriage Return
"\t“	Horizontal Tab
"\v“	Vertical Tab
"\\“	Backslash

- Example:

```
$ printf "Welcome to UNIX \n"
Welcome to UNIX
```

- Similar to C language, this command can be used with following format specifiers:

%d	Decimal integer
%f	Floating point number
%s	String
%o	Octal integer (base 8)
%x	Hexadecimal integer (base 16)

- Syntax:

```
printf("format-string", variable-list);
        where format-string contains one or more format-specifiers
              variable-list contains names of variables
```

- Example:

```
$ printf "My current shell is %s\n" $SHELL
My current shell is /bin/ksh
```

1.7.3 ls

- ls command can be used to obtain a list of all filenames in the current directory.
- -l option can be used to obtain a detailed list of attributes of all files in the current directory.
- Example:

```
$ ls -l
Type & Perm  Link  Owner    Group   Size  Date & Time     File Name
-rwxr-xr--   1     kumar    metal   195   may 10 13:45    chap01
drwxr-xr-x   2     kumar    metal   512   may 09 12:55    helpdir
```

1.7.4 who

• This command can be used to display the details of all the users logged-in into the unix system at the same time.

• Example:

```
$ who
kumar       tty0    Oct 8 14:10
rama        tty2    Oct 4 09:08
```

• –H option can be used to display the header information.

• Example:

```
$ who
NAME        LINE    TIME          COMMENT
kumar       tty0    Oct 8 14:10
rama        tty2    Oct 4 09:08
```

• -u option can be used to display detailed information of users.

• Example:

```
$ who -Hu
NAME        LINE    TIME                IDLE        PID     COMMENT
kumar       tty0    Oct 8 14:10         00:18       185
rama        tty2    Oct 4 09:08         00:23       123
```

1.7.5 date

• This command can be used to display the current date and time.

• Example:

```
$ date
Mon Sep 4 16:40:02 IST 2017
```

• This command can also be used with suitable format specifiers as arguments.

• Following are some format specifiers:

d – day of month (1 - 31) m - Month (01-12) y – last two digits of the year.
H– hour (00-24) M – minute (00-59) S – second (00-59)
D – date in the format mm/dd/yy
T – time in the format hh:mm:ss

• Syntax:

```
$ date +%format_specifier
```

• Example:

```
$ date +"%d-%m-%y"
04-09-17
$ date +%m                          // displays month number
09
```

1.7.6 passwd

• This command can be used to change user password.

• All Unix systems require passwords to help ensure that your files and data remain secure from hackers.

• Following are the steps to change your password –

1) To start, type password at the command prompt as shown below.
2) Enter your old password, the one you're currently using.
3) Type in your new password.
4) You must verify the password by typing it again.

```
$ passwd
Changing password for kumar
(current) Unix password: *****
New Unix password: *****
Retype new Unix password: *****
passwd: all authentication tokens updated successfully
$
```

1.7.7 cal

- This command can be used to display the calendar of the current month.
- Syntax:

  cal [[month] year]
- Example:

```
$ cal
September 2017
Su      Mo      Tu      We      Th      Fr      Sa
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
```

- This command can also be used to display the calendar of any specific month or a complete year.
- Example:

```
$ cal 9 2017
September 2017
Su      Mo      Tu      We      Th      Fr      Sa
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
```

- You can't hold the calendar of a year in a single screen page; it scrolls off rapidly to end of the year.
- To pause at each screen page, a pipe "|" can be connected to more pager:
- Example:

  $cal 2017 | more

1.8 Flexibility of Command Usage

- UNIX provides flexibility in using the commands.
- A command can often be entered in more than one way.
- Shell allow the following type of command usage:
    1) Combining Commands.
    2) A command line can overflow or Be split into multiple lines.
    3) Entering a command before previous command has finished.

1) Combining Commands

- UNIX allows you to specify more than one command in the single command line.
- Example:

```
$ wc sample.txt ; ls –l sample.txt          //Two commands separated by ; (semicolon).
2 3 16 sample.txt
-rw-rw-r-- 1 kumar group 16 Jan 30 09:35 sample.txt
$ ls | wc                  //Two commands combined here using filter
```

- You can even group several commands together so that their combined output is redirected to a file.
  (wc sample.txt ; ls –l sample.txt) > newfile
- When a command line contains a semicolon, the shell understands that the command on each side of it needs to be processed separately. Here ; is known as a metacharacter.

2) A Command line can be split into multiple lines

- UNIX terminal width is restricted to maximum 80 characters.
- Shell allows command line to overflow or be split into multiple lines.
- Example:

```
$ echo "This is                  // $ first prompt
> a three-line                  // > is a secondary prompt
> text message"                 // Command line ends here
This is a three-line text message
```

3) Entering a Command before previous command has finished

- You need not have to wait for the previous command to finish before you can enter the next command.
- Subsequent commands can be entered at the keyboard without waiting for prompt.
- The input remains stored in a buffer maintained by kernel for all keyboard input.
- The next command is passed on to the shell for interpretation after the previous program has completed.

1.9 Internal and External Commands

- Some commands are implemented as part of the shell itself rather than separate executable files. Such commands that are built-in are called internal commands.
- If the command (file) has an independence existence in the /bin directory, it is called external command.
- Examples:

```
$ type echo                  // echo is an internal command
echo is shell built-in
$ type ls                    // ls is an external command
ls is /bin/ls
```

- If the command exists both as an internal and external one, shell execute internal command only.
- Internal commands will have top priority compare to external command of same name.

1.10 type

- The UNIX is command based system i.e.,- things happens because the user enters commands in.
- Usually, UNIX commands are less than 5 characters long.
- All UNIX commands are single words like – cat, ls, pwd, date, mkdir, rmdir, cd, grep etc.
- The command names are all in lowercase.
- Example:

  $ LS bash:
  LS: command not found
- All UNIX commands are files containing programs, mainly written in C.
- All UNIX commands(files) are stored in directories(folders).
- If you want to know the location of executable program (or command), use type command.
- Example:

  $ type ls
  ls is /bin/ls
- When you execute ls command, the shell locates this file in the /bin directory and makes arrangements to execute it.

1.11 man

1.11.1 man: Browsing The Manual Pages Online

- This command can be used to display manual (documentation) of a specified command.
- Syntax:
  man command
- A pager is a program that
  → displays one screenful information and
  → pauses for the user to view the contents.
- The man command is configured to work with a pager.
- Following two commands can be used for navigation:
  1) Spacebar or f – moves forward one screen
  2) b – moves back one screen
- Finally, to quit the pager, press q. You'll be returned to the shell's prompt ($).

1.11.2 Understanding The man Documentation

- The man documentation is organized in eight (08) sections.
- Example:
  $ man wc //Help on the wc command User Commands wc(1)

NAME
wc – display a count of lines, words and characters in a file

SYNOPSIS
wc [-c | -m | -C] [-lw] [files....]

DESCRIPTION
The wc utility reads one or more input files and, by default, writes the number of newline characters, words and bytes contained in each input file to the standard output.

OPTIONS
The following options are supported:
-c Count Bytes
-m Count Characters
-l Count Lines
-w Count Words

OPERANDS
The following operands are supported:
File - a path name of an input file. If no file operand are specified, the standard input will be used.

USAGE
See largefiles(5) for the behavious of wc when encountering files >= 2 Gbytes.

EXIT STATUS
0 Successful Completion
> 0 An Error Occurred

SEE ALSO
space(3C), iswalpha(3C), iswspace(3C), setlocale(3C), attributes(5), environ(5), largefile(5)

- A man is divided into a number of compulsory and optional sections.
- Every command doesn't need all sections, but the first three (NAME, SYNOPSIS and DESCRIPTION) are generally seen in all man pages.
  1) NAME presents a one-line introduction of the command.
  2) SYNOPSIS shows the syntax used by the command.
  3) DESCRIPTION provides a detailed description.
- The SYNOPSIS follows certain conventions and rules:
  1) If a command argument is enclosed in rectangular brackets, then it is optional; otherwise, the argument is required.
  2) The ellipsis (a set if three dots) implies that there can be more instances of the preceding word.
  3) The | means that only one of the options shows on either side of the pipe can be used.
- All the options used by the command are listed in OPTIONS section.
- There is a separate section named EXIT STATUS which lists possible error conditions and their numeric representation.

1.11.3 Using man to Understand man

- You can use man command to view its own documentation.
- Example:

```
$ man man                    //Viewing man pages with man
```

- You can also set the pager to use with man ($ PAGER=less ; export PAGER).
- To understand which pager is being used by man, use $ echo $PAGER.

1.11.4 Further Help with man –k, apropos and whatis

- man –k: Searches a summary database and prints one-line description of the command.
- Example:

```
$ man –k awk                 //To know what awk does
awk (1) - pattern scanning and text processing language m
awk (1) - pattern scanning and text processing language n
awk (1) - pattern scanning and text processing language
```

- apropos can be used to list the commands and files associated with a keyword.
- Example:

```
$ apropos awk                //Same as $ man -k awk
awk (1) - pattern scanning and text processing language m
awk (1) - pattern scanning and text processing language n
awk (1) - pattern scanning and text processing language
```

- Example:

```
$ whatis awk  // Lists one-line description of command and same as $man -f awk awk
(1) - pattern scanning and text processing language
```

1.12 more

- more command can be used to display the content of a file on the screen, one page at a time.
- After each page, it stops and waits for you to tell it what to do.
- If the file contents is more, it will show the filename and percentage of the file that has been viewed:
  ----More--- (15%)
- Syntax:
  more FILENAME

Navigation

- Following two commands can be used for navigation:
  1) Spacebar or f – moves forward one screen
  2) b – moves back one screen
- Finally, to quit the pager, press q.
- You'll be returned to the shell's prompt($).

Repeat Factor

- The repeat factor used as a command prefix to repeat the command as many times as the prefix.
- For example:
  → 10f moves forward 10 screen. Here, 10 is acting as a repeat factor.

Searching for a pattern

- A pattern can be searched in a file using following 2 commands.

Command	Function
/Pattern	searches forward for pattern in the file.

- Syntax:
  /Pattern [Enter]

Repeating the Last Pattern Search

- The previous search command can be repeated using following 2 commands:

Command	Function
n	repeats the previous search command in the same direction

- n repeats search in same direction of original search.

Using more in pipeline

- Example:
  ls | more
- Here, pipe (|) is used where the output of one command is used as the input of the other command.

Table: Internal commands of more and less

more	less	action
spacebar or f	spacebar or f	one page forward
10f	-	20 pages forward
b	b	one page backward
10b	-	20 pages backward
[Enter]	[Enter]	one line forward
-	k	one line back
-	p or 1G	beginning of file
-	G	end of file
/Pattern	/Pattern	searches forward for pattern
n	n	repeat search forward
-	?Pattern	searches backward for pattern
.(dot)	-	repeat last command
v	v	start vi editor
!cmd	!cmd	executes unix command
q	q	quit
h	h	help

MODULE 1(CONT.): GENERAL PURPOSE UTILITIES

1.13 Knowing the User Terminal, Displaying its Characteristics and Setting Characteristics

1.13.1 uname : Knowing your Machines Characteristics

- This command can be used to display certain features of the OS running on your machine.
- Example:

```
$ uname -s
SunOS                    //name of OS used by sun solaris
$ uname -R
5.8                      //version of OS
```

1.13.2 tty: knowing your terminal

- This command can be used to know device name of the terminal.
- Example:

```
$ tty
/dev/pts/xyz        //here treminal filename is xyz resident in pts directory
                    //which is in turn in /dev directory
```

1.13.3 stty: Displaying and Setting Terminal Characteristics

- This command can be used to display and set various terminal attributes.
- –a option can be used to display the current settings.

```
$ stty -a
speed 38400 baud; rows 24; columns 116;
intr = ^C; quit = ^\; erase = ^?; kill = ^U;
```

Changing the Settings

- To remove the backspacing, we can use the following command:
  $stty –echoe
- To remove echo command to work, we can use the following command:
  $stty –echo

Changing the Interrupt Key (intr)

- To change the interrupt setting, we can use the following command:
  $stty intr \^c

Changing the end-of-File Key (eof)

- To change the end-of-File key setting, we can use the following command:
  $stty eof \^a
- [ctrl-a] will now terminate input for those commands that expects input from the keyboard.

When everything Else fail (sane)

- stty also provides another argument to set the terminal characteristics to value that will work on most terminals.
- Use the word sane as a single argument to the command:
  $stty sane //restores sanity to the terminal

1.14 Managing the Non-uniform Behaviour of Terminals and Keyboards

- Terminals and keyboards have no uniform behavioral pattern.
- Terminal settings directly impact the keyboard operation.
- The following table lists keyboard commands to try when things go wrong.

Keystroke or Command	**Function**
[Ctrl-h]	Erases text
[Ctrl-c] or Delete	Interrupts a command
[Ctrl-d]	Terminates login session or a program that expects its input from keyboard
[Ctrl-s]	Stops scrolling of screen output and locks keyboard
[Ctrl-q]	Resumes scrolling of screen output and unlocks keyboard
[Ctrl-u]	Kills command line without executing it
[Ctrl-\]	Kills running program but creates a core file containing the memory image of the program
[Ctrl-z]	Suspends process and returns shell prompt; use fg to resume job
[Ctrl-j]	Alternative to [Enter]
[Ctrl-m]	Alternative to [Enter]
stty sane	Restores terminal to normal status

MODULE 1(CONT.): ESSENTIAL SYSTEM ADMINISTRATION

1.15 Types of Accounts
There are 2 types of accounts on a Unix system:
1) Root Account
• The system administrator is known as superuser or root user.
• The superuser has complete control of the system.
2) User Accounts
• User accounts provide interactive access to the system for users and groups of users.
• General users are typically assigned to these accounts and usually have limited access to critical system files and directories.

1.15.1 The root Login
• The system administrator is known as superuser or root user.
• The superuser has complete control of the system.
• The superuser can run any commands without any restriction.
• The job of superuser includes:
→ maintaining user accounts
→ providing security and
→ managing disk space
→ performing backups
• The root account doesn't need to be separately created but comes with every system.
• The root account's password is generally set at the time of installation of the system and has to be used on logging in:

```
Login: root
Password: ********* [Enter]
# -
```

• The command prompt of root is hash (#).
• Once you login, you are placed in root's home directory "/".
• /sbin and /usr/sbin contains administrative commands of the system.

1.15.2 su: Becoming Super User
• Any user can acquire superuser status with the su command if they know the root password.
• For example, the user "kumar" becomes a superuser in this way:

```
$ su
Password: *****
# pwd
/home/kumar
```

• Though the current directory doesn't change, the # prompt indicates that kumar now has powers of a superuser.
• To be in root's home directory on superuser login, use

```
su –l
```

Creating a User's Environment
• User's often rush to the administrator with the complaint that a program has stopped running.
• The administrator first tries running it in a simulated environment.
• su command with a – (hyphen) can be used to recreate the user's environment without the login-password.

```
su –kumar
```

• This sequence executes kumar's .profile and temporarily creates kumar's environment.
• su runs in a separate sub-shell, so this mode is terminated by hitting [ctrl-d] or using exit.

1.16 /etc/passwd and /etc/shadow Files

• There are four main user administration files:

1) /etc/passwd: Keeps the user account and password information. This file holds the majority of information about accounts on the Unix system.
2) /etc/shadow: Holds the encrypted password of the corresponding account.
3) /etc/group: This file contains the group information for each account.
4) /etc/gshadow: This file contains secure group account information.

1) /etc/passwd

• This file contains following user account information:

1) Username
- This is the name used for logging into a UNIX system.

2) Password
- This stores the encrypted password which looks like ********.**

3) UID
- This is user's numerical identification.
- No two users can have the same UID.

4) GID
- This is user's numerical group identification.

5) Comments or GCOS
- This is contains user details or name address.
- This name is used at the front of the email address for this user.

6) Home Directory
- The directory where the user ends up on logging in.
- The login program reads this field to set the variable HOME.

7) Login shell
- This is the first program executed after logging in.
- This is usually the shell (/bin/ksh).
- The login program reads this field to set the variable SHELL.

2) /etc/shadow

• This file contains encrypted password of the corresponding account.
• This is the control file used by passwd to verify the correctness of a user's password.
• For every line in /etc/passwd, there's a corresponding entry in /etc/shadow.
• As a regular user, you do not have **read** or **write** access to this file for security reasons.
• Only superuser can access this file.

1.17 Managing Users and Groups

1.17.1 groupadd

- This command can be used to create a new group.
- We need to create groups before creating any account.
- Syntax:

```
groupadd -g gid groupname
        where -g GID → The numerical value of the group's ID
              groupname → Actual group name to be created
```

- Example:

```
groupadd -g 192 ISE                    //192 is the GID for ISE group
```

- /etc/group: This file contains the group information for each account.
- Group usually has more than one member with a different set of privileges.
- Creating a group involves defining the following parameters:
  1) A User identification number (UID) and username
  2) A group identification number (GID) and groupname
  3) The home directory 4) The login shell 5) The mailbox in /var/mail
- /etc/passwd: This file contains above 5 user account information.

1.17.2 Commands to Add, Modify and Delete Users

- Following commands can be used to create and manage user accounts:
  1) useradd 2) usermod & 3) userdel

1.17.2.1 useradd

- useradd command can be used to create a new user.
- Syntax:

```
useradd -u userid -g groupname -d homedir -s shell -m accountname
        where -u userid → You can specify a user id for this account
              -g groupname → Specifies a group account for this account
              -d homedir → Specifies home directory for the account
              -s shell → Specifies the default shell for this account
              -m → Creates the home directory if it doesn't exist
              accountname → Actual account name to be created
```

- Example:

```
# useradd -u 999 -g ISE -d /home/USP -s /bin/ksh  -m kumar
```

- This creates the user "kumar" with UID(userid)=999 group name= ISE
  home directory = /home/kumar shell = Korn shell MAIL = /var/mail
- This command
  → modifies the /etc/passwd, /etc/shadow, and /etc/group files and
  → creates a home directory.
- The .profile file is created and copied to user's home directory.
- Then, passwd command can be used to set new user password.

1.17.2.2 usermod

- The usermod command can be used to make changes to an existing account.
- This command uses the same options as the useradd command.

```
# usermod -s /bin/bash kumar          // to set current shell to bash shell
# usermod -g CSE kumar                // to set current group to CSE
```

1.17.2.3 userdel

- The userdel command can be used to delete an existing user & his account.

```
# userdel kumar          // to delete user "kumar"
```

- Example: To illustrate the usage of useradd, usermod & userdel.

```
# useradd -u 999 -g ISE -d /home/USP -s /bin/ksh  -m kumar
# usermod -s /bin/bash kumar          // to set current shell to bash shell
# usermod -g CSE kumar                // to set current group to CSE
# userdel kumar                       // to delete user "kumar"
```

MODULE 2: THE FILE SYSTEM
HANDLING ORDINARY FILES
BASIC FILE ATTRIBUTES

MODULE 2: THE FILE SYSTEM

2.1 UNIX Files

- A file is the container for storing information.
- The file doesn't store 1) file-size and 2) file-name.
- Some attributes of file are file-type, permissions, links, owner, group-owner etc.
- These file attributes are stored in inode table which is accessible only to kernel.
- A UNIX system makes no difference between a file and a directory, since a directory is just a file containing names of other files.
- Programs, services, texts, images etc are considered to be files.
- Generally, all devices including I/O devices are also considered to be files.

2.2 Naming Files

- A filename can consist up to 255 characters.
- Files may or may not have extensions.
- Files consist of any ASCII character expect the "/" and NULL character.
- Following is recommended for filenames:
  - → Alphabetic characters and numerals
  - → period(.), hyphen(-) and underscore(_)
- However, users are permitted to use control characters or other unprintable characters in a filename.
  Example: .last_time list. @ # $ % * abcd a.b.c.d.e
- UNIX is case sensitive; chap01, Chap01 and CHAP01 are three different filenames.

2.3 Basic File Types

• Three categories of files are:

1) Ordinary file (or regular file) – It contains only data as a stream of characters.
2) Directory file – it contains files and other sub-directories.
3) Device file – all devices and peripherals are represented by files.

2.3.1 Ordinary (Regular) File

• An ordinary file is a file on the system that contains data, text, or program instructions.
• An ordinary file can be either a text file or a binary file.

1) A text file contains only printable characters and you can view and edit them.
➢ Examples: All C and Java program sources, shell scripts are text files.
➢ Every line of a text file is terminated with the newline character.
2) A binary file contains both printable and non printable characters that cover the entire ASCII range.
➢ Examples: Most Unix commands, executable files, pictures, sound and video files are binary files.

Hidden Files

➢ An invisible file is one, the first character of which is the dot or the period character (.).
➢ Unix programs (including the shell) use most of these files to store configuration information.
➢ Some common examples of the hidden files include the files:
.profile – The Bourne shell (sh) initialization script.
.kshrc – The Korn shell (ksh) initialization script.
.cshrc – The C shell (csh) initialization script.
.rhosts – The remote shell configuration file.

2.3.2 Directory File

• A directory contains no data, but keeps details of the files and subdirectories that it contains.
• A directory file contains one entry for every file and subdirectory that it houses.
• Each entry has two components namely,
1) filename and
2) unique identification number of the file or directory (called the inode number).
• When you create or remove a file, the kernel automatically updates its corresponding directory by adding or removing the entry (filename and inode number) associated with the file.
• Unix directories are equivalent to windows folders.

2.3.3 Device File

• All the operations on the devices are performed by reading or writing the file representing the device.
• Advantage of device file is that some of the commands used to access an ordinary file also work with device file.
• Device filenames are generally found in a single directory structure, /dev.
• A device file is not really a stream of characters.
• It is the attributes of the file that entirely govern the operation of the device.
• The kernel identifies a de ice from its attributes and uses them to operate the device.

2.4 Parent Child Relationship

• All data in Unix is organized into files.
• All files are organized into directories.
• These directories are organized into a tree-like structure called the filesystem.

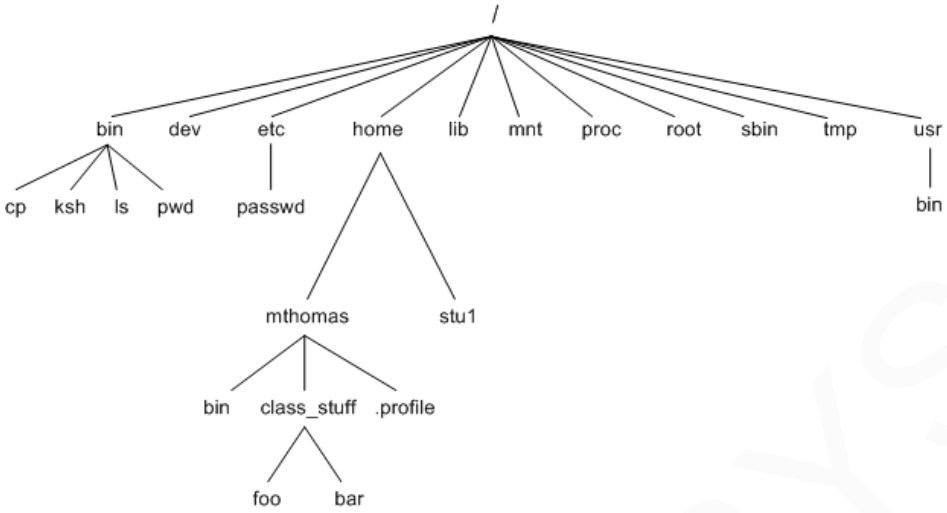


Figure 2.4: Parent Child Relationship

• The feature of UNIX file system is that there is a top, which serves as the reference point for all files.
• This top is called root (represented by a "/").
• The root is actually a directory.
• The root directory (/) has a number of subdirectories under it.
• The subdirectories in turn have more subdirectories and other files under them.
• Every file apart from root, must have a parent, and it should be possible to trace the ultimate parentage of a file to root.
• In parent-child relationship, the parent is always a directory.

2.5 Standard Directories (UNIX Filesystem)

• Following are the directories that exist on the major versions of Unix

1) / (root directory)
• This contains only the directories needed at the top level of the file structure.
2) /bin
• This contains common programs, shared by the system, the system administrator and the users.
3) /dev
• This contains references to all the CPU peripheral hardware, which are represented as files with special properties. These are device drivers.
4) /etc
• This contains system configuration files.
5) /home
• This contains home directories of the common users.
6) /lib
• This contains library files, including files for all kinds of programs needed by system and users.
7) /sbin
• This contains programs for use by the system and the system administrator.
8) /tmp
• This contains temporary space for use by the system. This space is cleaned upon reboot. So, don't use this for saving any work.
9) /usr
• This contains programs, libraries, documentation etc. for all user-related programs.
10) /var
• This contains storage for all variable files and temporary files created by users, such as
→ log files
→ mail queue or
→ print spooler area

2.6 HOME Variable

- When a user logs into the system, the user will be placed in a directory called home directory.
- Home directory is created by the system when the user account is created.
- The shell-variable HOME indicates the home directory of the current user.
- This variable is set for a user by the system admin in /etc/passwd.
- Example:

```
$ echo $HOME
/home/kumar
```

- Here, absolute pathname is shown.
- Absolute pathname is a sequence of directory names starting from root (/).
- The subsequent slashes are used to separate the directories.

2.7 PATH Variable

- This variable specifies the locations in which the shell should look for commands.
- Example:

```
$ echo $PATH
/bin: /usr/bin:
```

- When you specify a command like date, the system will locate the associated file from a list of directories specified in the PATH variable and then executes it.
- The PATH variable also includes the current directory.
- Whenever you enter any UNIX command, you are actually specifying the name of an executable file located somewhere on the system.
- The system goes through the following steps in order to determine which program to execute:

  1) Built in commands (such as cd and history) are executed within the shell.
  2) If an absolute path name (such as /bin/ls) or a relative path name (such as ./myprog), the system executes the program from the specified directory.
  3) Otherwise, the PATH variable is used.

2.8 Relative and Absolute Pathname

• A pathname is a text string made up of one or more names separated by a "/".

• A pathname specifies how to traverse (navigate) the hierarchical directory names in the file system to reach some destination object.

2.8.1 Absolute Pathname

• An absolute pathname begins with a slash (/).

• The Absolute path defines the location of a Directory or a file from the root file system (/).

• The absolute path contains the full path to the directory or file.

2.8.2 Relative Pathname

• The relative pathname do not begin with "/".

• It specify the location relative to your current working directory.

1) . (a single dot) - This represents the current directory.
2) .. (two dots) - This represents the parent directory.

• Example:

• 'date' command can executed in two ways as follows:

Using Absolute Pathname	**Using Relative Pathname**
$ /bin/date Thu Sep 7 10:20:29 IST 2017	$ date Thu Sep 7 10:20:29 IST 2017

• Example:

• P1.java can be copied to kumar under home directory in two ways as follows:

Using Absolute Pathname	**Using Relative Pathname**

$ pwd
/home/kumar
$ cp /home/sharma/P1.java /home/kumar
	$ pwd
	/home/kumar
	$ cp /home/sharma/progs .

• Example:

• cd & mkdir can be used in two ways as follows:

Using Absolute Pathname	**Using Relative Pathname**

$ pwd
/home/kumar
$ mkdir /home/kumar/progs
$ cd /home/kumar/progs
$ pwd
/home/kumar/progs
$ cd /home/kumar
$ pwd
/home/kumar
	$ pwd
	/home/kumar
	$ mkdir progs
	$ cd progs
	$ pwd
	/home/kumar/progs
	$ cd ..
	$ pwd
	/home/kumar

2.9 Directory Commands

2.9.1 pwd (print working directory)

- This command can be used to display the current working directory.
- Example:

```
$ pwd
/home/kumar
```

- Like HOME, it displays the absolute path.

2.9.2 cd

- This command can be used to change the current working directory.
- Syntax

```
cd PATHNAME
```

Case 1:

- This command can be used with the argument.
- This command can work with both absolute and relative path names.
- Example:

Using relative pathname	**Using absolute pathname**

$ pwd
/home/kumar
$ mkdir progs
$ cd progs
$ pwd
/home/kumar/progs
	$ pwd
	/home/kumar
	$ mkdir /home/kumar/progs
	$ cd /home/kumar/progs
	$ pwd
	/home/kumar/progs

Case 2:

- This command can also be used without the argument.
- When used without argument, this command changes the working directory to home directory.
- Example:

```
$ pwd
/home/kumar/progs
$ cd
$ pwd
/home/kumar
```

Case 3:

- This command can also be used with short hand notations.
- Example:

```
$ cd /        // changes the working directory to root directory (/)
$ cd ..       // changes the working directory to the one level up parent directory (..)
$ cd ../..    // changes the working directory to the two level up parent directory (../..)
```

2.9.3 mkdir

- This command can be used to create a new directory.
- Syntax:

  mkdir DIRECTORY_NAME
- This command can work with both absolute and relative path names.
- Example:

Using relative pathname	**Using absolute pathname**

$ pwd
/home/kumar
$ mkdir progs
$ cd progs
$ pwd
/home/kumar/progs
	$ pwd
	/home/kumar
	$ mkdir /home/kumar/progs
	$ cd /home/kumar/progs
	$ pwd
	/home/kumar/progs

Case 2:

- This command can also accept more than one directory name as arguments.
- Example:

```
$mkdir usp ade dms                  // creates 3 directories under current directory
$mkdir sem3 sem3/usp sem3/ade       // creates 3 subdirectories – sem3, usp, ade
```

- The order of specifying arguments is important. You cannot create subdirectories before creation of parent directory.
- System refuses to create a directory due to following reasons:
  1) User doesn't have permission to create directory. (i.e. directory write protected).
  2) The directory already exists.
  3) There may be ordinary file by that name in the current directory.

2.9.4 rmdir

- This command can be used to delete a directory.
- Syntax:

  rmdir DIRECTORY_NAME

Case 1:

- Example:

```
$ rmdir usp                         // delete usp directory under current directory
```

Case 2:

- This command can also accept more than one directory name as arguments.
- Example:

```
$ mkdir sem3 sem3/usp sem3/ade      // creates 3 subdirectories – sem3, usp, ade
$ rmdir sem3/ade sem3/usp sem3      // deletes 3 subdirectories – sem3, usp, ade
```

- Here, rmdir expects the arguments to be reverse of mkdir's arguments.
- The order of specifying arguments is important. You cannot delete parent directories before deletion of subdirectories.
- System refuses to delete a directory due to following reasons:
  1) User doesn't have permission to delete directory. (i.e. write protected directory).
  2) The directory doesn't exist in system.
  3) The directory is your present working directory.

MODULE 2(CONT.): HANDLING ORDINARY FILES

2.10 File Related Commands

2.10.1 cat

• This command can be used to display the content of a file on the terminal.
• Syntax:

```
cat FILENAME
```

Case 1:

• Example:

```
$ cat P1.c
WELCOME TO UNIX          // contents of P1.c
$ cat P2.c
WELCOME TO PERL          // contents of P2.c
```

Case 2:

• This command can also accept more than one filename as arguments.
• Example:

```
$ cat P1.c P2.c
WELCOME TO UNIX          // contents of P1.c
WELCOME TO PERL          // contents of P2.c
```

• Here, the content of the second file is shown immediately after the first file.
• So, this command concatenates two files- hence its name (cat).

Case 3:

• This command can also be used to create a new file.
• Syntax:

```
cat > FILENAME
```

• Example:

```
$ cat > P3.c
WELCOME TO SHELL                // contents of P3.c
[ctrl-d]                        // Terminates P3.c
$ cat P3.c
WELCOME TO SHELL                // contents of P3.c
```

cat Options

1) Displaying Non-printing Characters (-v)

• By default, without any option, this command displays only printing ASCII characters of the file.
• -v option can be used to display even non-printing ASCII characters of the file.

2) Numbering Lines (-n)

• -n option can be used to number the lines of the file.
• This option helps the programmer in debugging programs.

2.10.2 wc

• This command can be used to get a count of the total number of lines, words, and characters contained in a file.

• Syntax:

wc FILENAME

Case 1:

• Example:

```
$ cat P1.c
WELCOME              // contents of P1.c
TO
UNIX
$ wc P1.c
LINE   WORD  CHARATCTER  FILENAME
3      3     15          P1.c
```

• The header includes the following attributes:

1) LINE

• This represents the total number of lines in the file.

2) WORD

• This represents the total number of words in the file (excluding space, tab and newline).

3) CHARATCTER

• This represents the total number of characters in the file (including space, tab and newline).

4) FILENAME

• This represents the name of the file.

Case 2:

• This command can also accept more than one filename as arguments.

• Example:

```
$ wc  P1.c P2.c
3      3      15          P1.c
3      3      15          P2.c
```

wc Options

–l option can be used to count only number of lines

–w option can be used to count only number of words

–c option can be used to count only number of characters

• Example:

```
$ wc -l P1.c
3       P1.c
$ wc -c P1.c
15      P1.c
```

2.10.3 cp

• This command is used to copy a file(s) from one location to another location.
• It creates an exact image of the file on the disk with a different name.
• Syntax:

```
cp SOURCE_FILE DESTINATION_FILE
```

Case 1:

• This command can be used to copy a file within current working directory.
• Example:

```
$ cp FILE1 FILE2      // copies contents of FILE1 to FILE2 in current working directory
```

• Here,
1) If the destination file doesn't exist, it will first be created before copying takes place.
2) If the destination file exists, it will be overwritten without any warning from the system.

Case 2:

• This command can also be used to copy a file to the another directory.
• Example:

```
$ cp FILE1 part2/FILE2
```

• Here, this copies the file FILE1 from your current working directory to the file FILE2 in the subdirectory "part2".

Case 3:

• This command can also be used with .(dot) to signify the current directory as the destination.
• Example:

```
$ cp part2/FILE2 .          same as      $ cp part2/FILE2 FILE2
```

• Here, this copies the file FILE2 in the subdirectory "part2" to your current working directory.

Case 4:

• This command can also accept more than 2 filenames as arguments.
• In this case, the last filename must be a directory.
• Example:

```
$ cp FILE1 FILE2 FILE3 module      // copies 3 files to "module" directory
```

• Here, "module" directory should already exist because cp cannot create a directory.

Case 5:

• This command can also be used with metacharcters (* or ?).
• Example:

```
cp *.pdf DIR1          //copies all the files with extensions .pdf to directory DIR1
```

cp Options

1) Interactive Copying (-i)

• –i option can be used to warn the user before overwriting the destination file.
• Example:

```
$ cp –i FILE1 FILE2                    // copies contents of FILE1 to FILE2 if "Y" is entered
$ cp: overwrite FILE2 (yes/no)? Y
```

2) Recursive Copying (-R)

• –R option can be used to recursively copy an entire directory structure from one location to another location.
• Entire directory including all files in its subdirectories will be copied.
• Example:

```
cp –R DIR1 DIR2
```

• If directory DIR2 doesn't exist, cp creates it along with the associated subdirectories.

2.10.4 mv

• This command renames or moves files.

Case 1:

• This command can be used to rename a file in the current directory.
• Syntax:

```
mv OLDFILENAME NEWFILENAME
```

• It doesn't create a copy of the file; it merely renames it.
• No additional space is consumed on disk during renaming.
• Example:

```
$ mv OLDFILE NEWFILE              // renames the OLDFILE by NEWFILE
```

Case 2:

• This command can also be used to move a group of files to a directory.
• In this case, the last filename must be a directory.
• Example:

```
$ mv FILE1 FILE2 FILE3 module     // moves 3 files to "module" directory
```

2.10.5 rm

• This command can be used to delete a file.
• Syntax:

```
rm FILENAME
```

• Example:

```
$ rm FILE1                  // deletes FILE1
$ rm FILE1 FILE2 FILE3      // deletes three files
$ rm *.pdf                  // deletes all files with extensions .pdf in the currant directory
```

rm Options

1) Interactive Deletion (-i)

• –i option can be used to warn the user before deleting the file.
• Example:

```
$ rm –i FILE1 FILE2                    // delete FILE1 & FILE2 if "Y" is entered
rm: remove FILE1 (yes/no)? ? Y
rm: remove FILE2 (yes/no)? ? Y
```

2) Recursive Deletion (-r or -R)

• –R option can be used to recursively delete an entire directory structure.
• Entire directory including all files in its subdirectories will be deleted.
• Example:

```
$ rm –r DIR1          // delete DIR1 and all its subdirectories & files
$ rm -r *             // deletes all files in the current directory and all its subdirectories.
```

3) Forceful Deletion (-f)

• By default, this command cannot delete a file which is write-protected.
• –f option can be used to delete even the write-protected file.
• Example:

```
rm -rf *              // deletes all files in the current directory and all its subdirectories.
```

2.10.6 od

- This command can be used to display the content of executable file in a ASCII octal form.
- Example:

```
$ cat P1.obj
abcd efgh                    // content of file P1.obj
abcd efgh
```

od Options

1) byte (-b)

- -b option can be used to display octal value of each printable character.
- Each line displays 16 bytes of data in octal, preceded by the offset of the first byte in the line.
- Example:

```
$ od –b file
  offset   <-----------------  16 bytes of data in octal  ------------------------>
0000000 141 142 143 144 040 145 146 147 148 040 040 040 040 040 040 012
0000020 141 142 143 144 040 145 146 147 148 040 040 040 040 040 040 012
```

2) character (-c)

- -c option can be used to display the printable characters and its corresponding octal value.

```
$ od –bc file
od -bc ofile
0000000 141 142 143 144 040 145 146 147 148 040 040 040 040 040 040 012
          a   b   c   d       e   f   g   f                           \n
0000020 141 142 143 144 040 145 146 147 148 040 040 040 040 040 040 012
          a   b   c   d       e   f   g   f                           \n
```

MODULE 2(CONT.): BASIC FILE ATTRIBUTES

2.11 File Attributes and Permissions and Knowing them

- ls command can be used to obtain a list of all filenames in the current directory.
- -l option can be used to obtain a detailed list of attributes of all files in the current directory.
- For example:

```
$ ls -l
Type & Perm  Link  Owner   Group   Size  Date & Time     File Name
-rwxr-xr--   1     kumar   metal   195   may 10 13:45    chap01
drwxr-xr-x   2     kumar   metal   512   may 09 12:55    helpdir
```

- The header includes the following attributes:

1) Type & Perm

- This represents the file-type and the permission given on the file.

File-Type

- The first character indicates type of the file as shown below:
  i) hyphen (-) for regular file ii) d for Directory file
  iii) l for Symbolic link file iv) b for block special file v) c for character special file

File Permission

- The remaining 9 characters indicates permission of the file.
- There are 3 permissions: read (r), write (w), execute (x).
  i) Read: Grants the capability to read, i.e., view the contents of the file.
  ii) Write: Grants the capability to modify, or remove the content of the file.
  iii) Execute: User with execute permissions can run a file as a program.
- There are 3 types of users: owner, groups and others.
- The permission is broken into group of 3 characters:
  i) The first 3 characters (2-4) represent the permissions for the file's owner.
  ii) The middle 3 characters (5-7) represent the permissions for the group to which the file belongs.
  iii) The last 3 characters (8-10) represents the permissions for everyone else.
- Consider an example -rwxr-xr--
  i) Owner has read (r), write (w) and execute (x) permission.
  ii) Group members have read (r) and execute (x) permission, but no write permission.
  iii) Others have read (r) only permission.

2) Link

- This indicates the number of file names maintained by the system.
- This does not mean that there are so many copies of the file. (Link similar to pointer).

3) Owner

- This represents the owner of the file. This is the user who created this file.

4) Group

- This represents the group of the owner.
- Each group member can access the file depending on the permission assigned.
- The privileges of the group are set by the owner of the file and not by the group members.
- When the system admin creates a user account, he has to assign these parameters to the user:
  i) The user-id (UID) and ii) The group-id (GID)

5) Size

- This represents the file size in bytes.
- It is the number of character in the file rather than the actual size occupied on disk.

6) Date & Time

- This represents the last modification date and the time of the file.
- If you change only the permissions /ownership of the file, the modification time remains unchanged.
- If at least one character is added/removed from the file then this field will be updated.

7) File name

- This represents the file or the directory name.

2.12 ls

- This command can be used to list the files and directories stored in the current directory.
- Syntax:

  ls [options] [argument]
- For example:

  $ ls

  bin lib users work
- With options it can provide information about the size, type of file, permissions, dates of file creation, change and access.

2.12.1 ls Options

- -l option can be used to get more information about the listed files.
- When no argument is used, the listing will be of the current directory.
- There are many very useful options for the ls command.
- A listing of many of them follows.
- When using the command, string the desired options together preceded by "-".

a	Lists all files, including those beginning with a dot (.).
d	Lists only names of directories, not the files in the directory
F	Indicates type of entry with a trailing symbol: i) executables with * ii) directories with / and iii) symbolic links ith @
R	Recursive list
u	Sorts filenames by last access time
t	Sorts filenames by last modification time
i	Displays inode number
l	Long listing: lists the mode, link information, owner, size, last modification (time). If the file is a symbolic link, an arrow (→) precedes the pathname of the linked to file.

- The mode field is given by the -l option and consists of 10 characters.
- The first character is one of the following:

Character	**If entry is a**
d	directory
-	plain file
b	block-type special file
c	character-type special file
l	symbolic link
s	socket

- The next 9 characters are in 3 sets of 3 characters each.
- They indicate the file access permissions:

  1) the first 3 characters refer to the permissions for the user
  2) the next three for the users in the Unix group assigned to the file, and
  3) the last 3 to the permissions for other users on the system.
- Designations are as follows:

  r read permission
  w write permission
  x execute permission
  - no permission
- Example:

```
$ ls -al          // To get a long listing of all files in a directory
total 24
drwxr-sr-x 5 workshop acs 512 J  n 7 11:12 .
drwxr-xr-x 6 root sys 512 May 29 09:59 ..
-rwxr-xr-x 1 workshop acs 532 May 20 15:31 .cshrc
-rwxr-xr-x 1 workshop acs 238 May 14 09:44 .login
-rw-r--r-- 1 workshop acs 273 May 22 23:53 .plan
```

2.12.2 Listing Directory Attributes

- $ls -d : can be used to list only names of directories, not the files in the directory.
- However, this command will not list all subdirectories in the current directory.
- For example:

```
$ls –ld helpdir progs
drwxr-xr-x 2 kumar metal 512 may 9 10:31 helpdir
drwxr-xr-x 2 kumar metal 512 may 9 09:57 progs
```

- Directories are easily identified in the listing by the first character(d) of the first column.

2.13 Changing File Permissions

- A file is created with a default set of permission.
- chmod command can be used to change permission of a file.
- This command can be used in two ways: 1) Relative mode and 2) Absolute mode.

2.13.1 Relative Permissions

- This command can be used to add/delete permission for specific type of user (owner, group or others).
- This command can be used to
  → change only those permissions specified in the command line and
  → leave the other permissions unchanged.
- Syntax:
  chmod category operation permission filename
- This command takes 4 arguments:
  1) category can be
     u → user (owner)
     g → group o → others a → all (ugo)
  2) operation can be
     + → assign
     - → remove
     = → absolute
  3) permission can be
     r → read
     w → write
     x → execute
  4) Filename whose permission has to changed
- Example:

```
$ ls -l xstart
-rw-r—r-- 1 kumar metal 1906 sep 23:38 xstart
$ chmod u+x xstart          // user (u) is added(+) an execute(x) permission
$ ls -l xstart
-rwxr—r-- 1 kumar metal 1906 sep 23:38 xstart
$ chmod ugo+x xstart  OR  chmod a+x xstart  // all(a) are added(+) an execute(x) permission
$ ls –l xstart
-rwxr-xr-x 1 kumar metal 1906 sep 23:38 xstart
$ chmod go-r xstart         // group(g) & others(o) are removed(-) a read(r) permission
$ls –l xstart
-rwx—x--x 1 kumar metal 1906 sep 23:38 xstart
```

- This command can also accept multiple file names.
- Example:
  $ chmod u+x note1 note2 note3

Recursively Changing File Permissions

- chmod command can be used to change recursively permission of all the files and subdirectories found in the current directory.
- Example:
  chmod -R a+x c_progs // current directory c_progs
- Here, all files and subdirectories are made executable for all users in current directory c_progs.

2.13.2 Absolute Permissions

- This command can be used to add/delete permission for all type of users (owner, group or others).
- This command can be used to change all permissions specified in the command line.
- Syntax:

  chmod octal_value filename
- This command takes 2 arguments:

  1) octal_value contains 3 octal digits to represent 3 type of users (owner, group or others).
  - 1) First digit is for user
  - 2) Second digit is for group and
  - 3) Third digit is for others

  Each digit represents a permission as shown below:
  - 4 (100) – read only
  - 2 (010) – write only
  - 1 (001) - execute only
  - 6 (110) – read & write only

  For ex: octal_value of 644(110 100 100) means
  - → user can read & write only
  - → group can read only
  - → others can read only

  2) Filename whose permission has to changed.
- Example:

```
$ ls -l xstart
-rw-r—r-- 1 kumar metal 1906 sep 23:38 xstart                    // current permission 644
$ chmod 744 xstart          same as      $ chmod u+x xstart
                                         // user(u) is added(+) an execute(x) permission
$ ls –l xstart
-rwxr—r-- 1 kumar metal 1906 sep 23:38 xstart
$ chmod 755 xstart          same as      $ chmod a+x xstart
                                         // all(a) are added(+) an execute(x) permission
$ ls –l xstart
-rwxr-xr-x 1 kumar metal 1906 sep 23:38 xstart
$ chmod 711 xstart          same as      $ chmod go-r xstart
                                         // group(g) & others(o) are removed(-) a read(r) permission
$ls –l xstart
-rwx—x--x 1 kumar metal 1906 sep 23:38 xstart
```

- It is the directory permissions that determine whether a file can be deleted or not.

  i) 777 signify all permissions for all categories, but still we can prevent a file from being deleted.

  ii) 000 signifies absence of all permissions for all categories, but still we can delete a file.
- The system administrator can change the file permissions of every user.

  i) Only owner can change the file permissions.

  ii) User cannot change other user's file's permissions.

Relative Permissions	**Absolute Permissions**

$ ls -l xstart
-rw-r—r-- 1 kumar metal 1906 sep 23:38 xstart
$ chmod u+x xstart
$ ls -l xstart
-rwxr—r-- 1 kumar metal 1906 sep 23:38 xstart
$ chmod ugo+x xstart OR chmod a+x xstart
$ ls –l xstart
-rwxr-xr-x 1 kumar metal 1906 sep 23:38 xstart
$ chmod go-r xstart
$ls –l xstart
-rwx—x--x 1 kumar metal 1906 sep 23:38 xstart
	$ ls -l xstart
	-rw-r—r-- 1 kumar metal 1906 sep 23:38 xstart
	$ chmod 744 xstart
	$ ls -l xstart
	-rwxr—r-- 1 kumar metal 1906 sep 23:38 xstart
	$ chmod 755 xstart OR chmod a+x xstart
	$ ls –l xstart
	-rwxr-xr-x 1 kumar metal 1906 sep 23:38 xstart
	$ chmod 711 xstart
	$ls –l xstart
	-rwx—x--x 1 kumar metal 1906 sep 23:38 xstart

2.13.3 Changing File Ownership

- While creating an account on Unix, it assigns a owner ID and a group ID to each user.
- The permissions are assigned to each user based on the Owner and the Groups.
- Two commands are available to change the owner and the group of files:
  1) chown (change owner) is used to change the owner of a file.
  2) chgrp (change group) is used to change the group of a file.

2.13.3.1 chown

- This command can be used to change the ownership of a file.
- Syntax:

  chown USERNAME FILENAME
- The value of the user can be the name or uid(user id) of a user on the system.
- Example:

```
$ ls -l note
-rwxr----x 1 kumar ISE 347 may 10 20:30 note
$ chown sharma note                    //now sharma becomes new owner of file "note"
ls -l note
-rwxr----x 1 sharma ISE 347 may 10 20:30 note
```

- Now, new owner will have same file permissions as that of old owner.
- Old owner cannot edit "note" since there is no write privilege for group and others.
- Old owner cannot get back the ownership.
- Only super user can change the ownership of any file.
- But normal users can change the ownership of only those files that they own.

2.13.3.2 chgrp

- This command can be used to change the group-ownership of a file.
- Syntax:

  chgrp GROUPNAME FILENAME
- The value of the group can be the name or GID(group ID) of a group on the system.
- Example:

```
$ ls -l note
-rwxr----x 1 kumar ISE 347 may 10 20:30 note
$ chgrp CSE note                       //now CSE becomes new group-owner of file "note"
ls -l note
-rwxr----x 1 kumar CSE 347 may 10 20:30 note
```

- No super user permission is required.

2.13.4 Directory Permissions

- The default permissions of a directory are: rwxr-xr-x (755).
  1) Read directory permission grants the ability to view a file.
  2) Write directory permission grants the ability to add, change or remove files from the directory.
  3) Execute directory permission grants the ability to list (ls) the directory content or search (find) for files in the directory.
- Example:

```
$mkdir c_progs
$ls –ld c_progs
drwxr-xr-x 2 kumar metal 512 may 9 09:57 c_progs
```

- A directory must never be writable by group and others .

MODULE 3: THE VI EDITOR
THE SHELL
FILTERS USING REGULAR EXPRESSION

MODULE 3: THE VI EDITOR

3.1 vi Editor

- vi is a text editor used to edit files in Unix.
- vi is generally considered the de facto standard in Unix editors because:
  1) It's usually available on all the flavors of Unix system.
  2) Its implementations are very similar across the board.
  3) It requires very few resources.
  4) It is more user-friendly than other editors such as the ed or the ex.
- vi editor can be used
  → to edit an existing file
  → to create a new file from scratch and
  → to read a text file.

3.1.1 vi Basics

- Syntax for opening vi editor:
  vi filename
- Here, if the file already exists, the file will be opened.
  If the file doesn't exist, a new file will be created and then opened.
- For example:
  vi testfile
- The above command will generate the following full screen:

```
|
~
~
~
~
"testfile" [New File]
```

Figure 7.1a : vi editor in full screen mode

- On the screen,
  → cursor is positioned at the top
  → tilde (~) is shown in all remaining lines. A tilde represents an unused line.
  → filename is shown at the bottom.

Last Line

- The last line is reserved for commands that you can enter.
- This line is also used to display messages by the system.

Operation Modes

- vi editor can be used in following 3 modes:

1) Command-Mode

- By default, the file will be opened in a command-mode.
- In this mode, whatever you type is interpreted as a command.

2) Insert Mode

- To enter text into the file, you must be in the insert mode.
- To switch from command-mode to insert mode, press the key "i".
- In this mode, whatever you type is interpreted as input and placed in the file.
- To switch from insert mode to command-mode, press the key "Esc".

```
WELCOME TO UNIX SHELL PROGRAMMING
~
~
~
~
"testfile" [New File]
```

Figure 7.1b : inserting some text

3) Execute Mode (Last Line Mode)

- To save the entered text, you must switch to the execute mode.
- In this mode, whatever you type is interpreted as a ex-command.
- To switch from command-mode to execute mode, press the keys ":".
- After inserting ex command, press [Enter] to execute the command.
- To switch from execute mode to command-mode, press the key "[Enter]".

3.1.2 Different ways of Invoking and Quitting vi

• The following table lists out the basic commands to invoke the vi editor

Command	Function
vi filename	Creates a new file if it already does not exist, otherwise opens an existing file
vi -R filename	Opens an existing file in the read-only mode
view filename	Opens an existing file in the read-only mode

• vi always starts in the command-mode.
• To enter text, you must be in the insert mode for which simply type "i".
• To come out of the insert mode, press the Esc key, which will take you back to the command-mode.

3.1.3 Different Modes of vi

• Three modes of vi editor are: 1) Command-mode 2) Input-Mode and 3) Ex-Mode

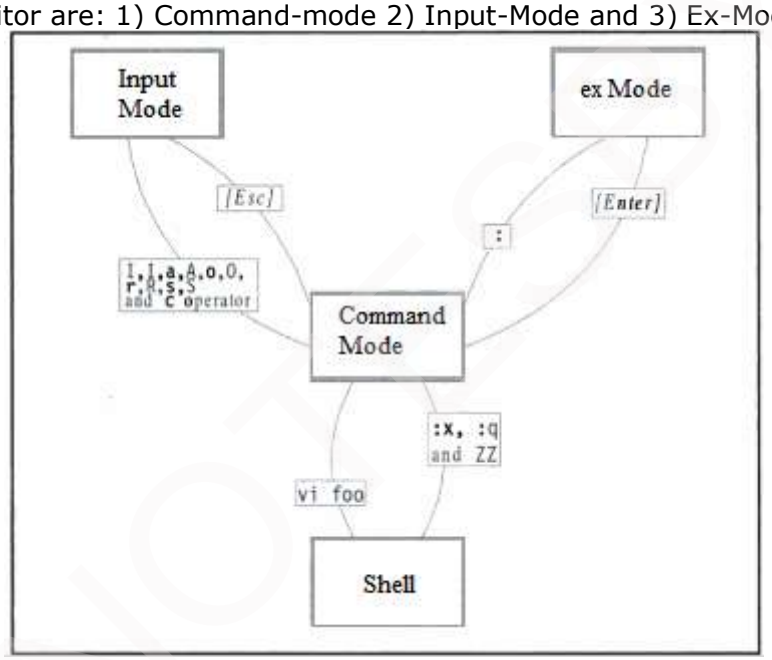


Figure 7.2: Three modes of vi editor

1) Command-Mode

• By default, the file will be opened in a command-mode.
• In this mode,
  → whatever you type is interpreted as a command
  → you can pass commands to act on text
  → you can copy(or yank) and delete text
  → you can move to other modes i.e. Input-Mode or Ex-Mode
  → you can move the cursor in the four directions (h, j, k and l)
• Some of the command-mode commands are:

Command	Function
x	to delete a text command
dd	to delete entire line
P or p	to put the character
y	to yank the text
yy	to copy entire line
u	to undo the recent actions

2) Insert Mode

- To enter text into the file, you must be in the insert mode.
- To switch from command-mode to insert mode, press the key "i".
- In this mode,
  → whatever you type is interpreted as input and placed in the file.
  → you can insert, append & replace text
- Some of the Input-Mode commands are:

Command	Function
i	inserts text
a	appends text
I	inserts at beginning of line
A	appends text at end of line
o	opens line below
O	opens line above
r	replaces a single character
s	replaces with a text
S	replaces entire line

- After a text is typed, the cursor will be positioned on the last character of the last line.
- This last line is known as current line.
- The character where the cursor is positioned is known as current cursor position.
- Input-Mode is used to handle files and perform substitution.
- To switch from insert mode to command-mode, press the key "Esc".

3) Execute Mode (Last Line Mode)

- Actually, the text entered has not been saved on disk but exists in a buffer (a temporary storage).
- To save the entered text, you must switch to the execute mode.
- In this mode,
  → whatever you type is interpreted as a ex-command
  → you can save file
  → you can quit editing mode
- Some of the sample ex commands are:

Command	Function
:W	saves file and remains in editing mode
:x	saves and quits editing mode
:wq	saves and quits editing mode
:w <filename>	save as
:w! <filename>	save as, but overwrites existing file
:q	quits editing mode
:q!	quits editing mode by rejecting changes made
:sh	escapes to UNIX shell
:recover	recovers file from a crash

- To switch from command-mode to execute mode, press the keys ":".
- Anything entered in front of the colon ":" prompt it is taken as an ex command.
- After inserting ex command, press [Enter] to execute the command.
- To switch from execute mode to command-mode, press the key "[Enter]".

3.1.4 Repeat Factor

- The repeat factor used as a command prefix to repeat the command as many times as the prefix.
- For example:
  In command-mode,
  → k moves the cursor one line up.
  → 10k moves cursor 10 lines up. Here, 10 is acting as a repeat factor.

3.2 Input Mode Commands

3.2.1 Editing Files

- To edit the file, you need to be in the insert mode.
- There are many ways to enter the insert mode from the command mode.
- Some of the Input-Mode commands are:

Command	Function
i	inserts text
a	appends text
I	inserts at beginning of line
A	appends text at end of line
o	opens line below
O	opens line above

3.2.1.1 Insertion of Text (i and a)

- To insert text before the current cursor location, press the key "i".
- To append(or insert) text after the current cursor location, press the key "a".
- Then, whatever you type is entered and displayed on the screen.
- Also, the existing text will be shifted right without being overwritten.
- To switch from insert mode to command-mode, press the key "Esc".

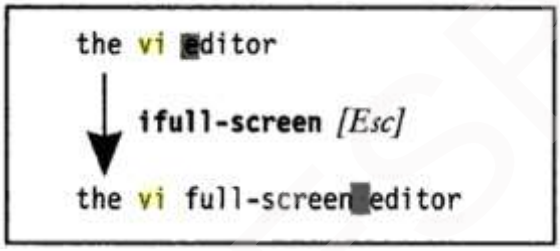


Figure 7.2 : text insertion with i

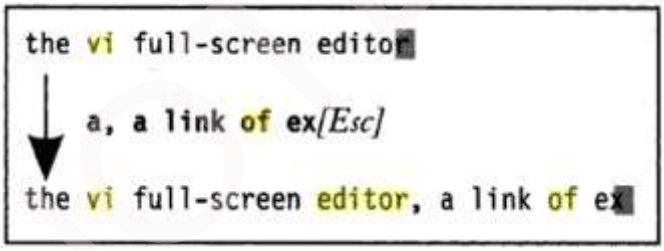


Figure 7.3 : text appending with a

3.2.1.2 Insertion of Text at Line Extremes (I and A)

- To insert text at the beginning of' the current line, press the key "i".
- To append(or insert) text at the end of the current line, press the key "A".

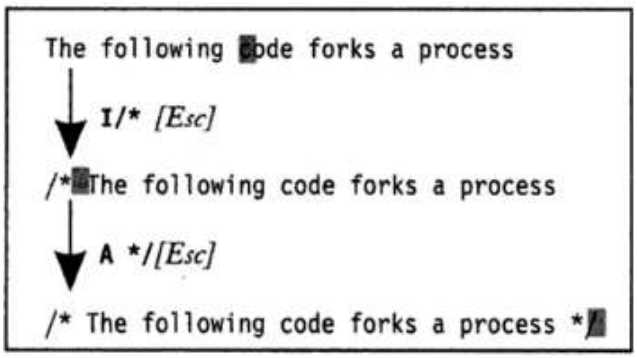


Figure 7.5 : using I & A to create a comment line in C program

3.2.1.3 Opening a New Line (o and O)

- To creates a new line for text entry below the cursor location, press the key "o".
- To creates a new line for text entry above the cursor location press the key "O".

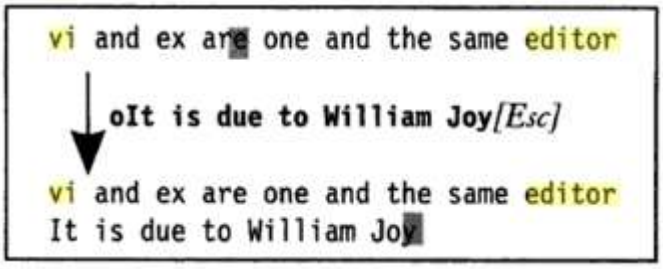


Figure 7.6 : opening a new line with o

3.2.2 Change Commands

- You also have the capability to change characters, words, or lines in vi without deleting them.
- Some of the Input-Mode commands are:

Command	Function
r	replace with a single character
s	replace with a multiple characters
R	replace with a text
S	replace entire line

3.2.2.1 Replacing Text (r, s, R and S)

- There are 4 commands to change existing text: r, s, R and S.
- To replace a single character with another character, press the key "r".
- The text under the cursor will be replaced by the new text entered.

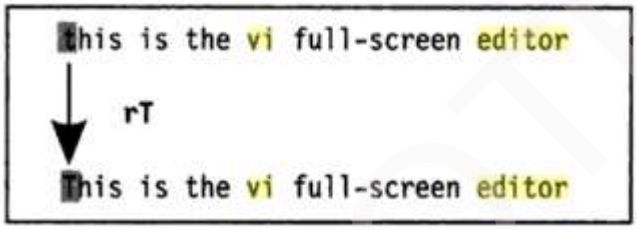


Figure 7.7 : replacing a single character with r

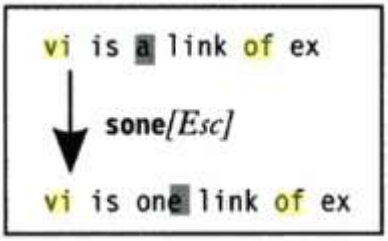


Figure 7.8: replacing a text with s

- To replace a single character with multiple characters, press the key "s".
- The text under the cursor will be deleted.
- Then command-mode is changed to Input-Mode.
- A repeat factor can be used to replace multiple characters.
- For example:

  3s replaces three characters with new text.

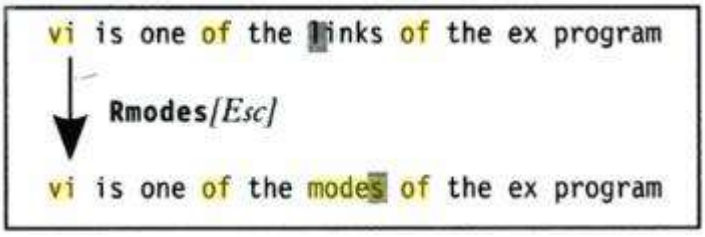


Figure 7.9: replacing a text with R

- R and S command act similar to lowercase 'r' and 's' except that they act on a larger group of characters.
- To replace all text on the right of the cursor position, press the key "R".
- To replace the entire line irrespective of cursor position, press the key "S".

3.3 Execute Mode Commands

• When you edit a file using vi, the original file is not saved on disk but exists in a buffer (a temporary storage).
• To save the file, you must save this buffer.
• Some of the sample ex commands are:

Command	Function
:W	saves file and remains in editing mode
:x or :wq	saves and quits editing mode
:w <filename>	save as
:w! <filename>	save as, but overwrites existing file
:q	quits editing mode
:q!	quits editing mode by rejecting changes made
:sh	escapes to UNIX shell
:recover	recovers file from a crash

Saving Your Work (:w)

• To save the contents of the editor, press the keys ":w[Enter]". This command will write the buffer to disk.
• To save contents of editor in another filename called "testfile2", press the keys ":w testfile2 [Enter]".

Aborting Editing(:q)

• To quit out of vi, press the keys ":q[Enter]".
• Problem: If your file has been modified in any way, the editor will warn you of this, and not let you quit.
• Solution: To quit out of vi without saving, press the keys ":q![Enter]".

Saving and Quitting (:x and :wq)

• To save the contents & quit the editor, press the keys ":wq[Enter]" or ":x[Enter]".

Escape to the UNIX Shell (:sh and [Ctrl-z])

• To switch from Ex-Mode to shell prompt, press the keys ":sh[Enter]".
• In shell prompt, any UNIX commands can be executed.
• To switch from shell prompt to Ex-Mode, press the keys "[ctrl-d]" or " exit "

Recovering from a Crash (: recover)

• The vi stores most of its buffer information in a hidden swap file.
• This file will remain on disk.
• To save the file contents, press the keys ":recover [Enter]".

Writing Selected Lines

• We can save certain lines in editor to another file.
• Example:

Command	Function
:6w newpgm.c	saves 6th line to another file
:10,50w newpgm.c	saves lines 10 through 50 to the file newpgm2.c.

• The symbols dot "." and dollar "$" have special significance:
  1) "." represents the current line and
  2) "$" represents the last line of the file.
• Example:

Command	Function
:.w tempfile	Saves current line(where cursor is positioned)
:$w tempfile	Saves last line
:.,$w tempfile	Saves current line through end

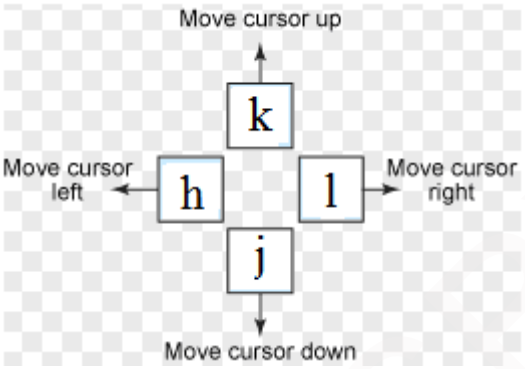
3.4.1 Navigation Commands

- To move around within a file without affecting your text, you must be in the command-mode.
- In command-mode, commands doesn't show up on screen but simply performs a function.

3.4.1.1 Movement in the Four Direction (h, j ,k and l)

- There are 4 character navigation commands:

Command	Function
k	moves cursor up
j	moves cursor down
h	moves cursor left
l	moves cursor right



- Without a repeat factor, these keys move the cursor by one position.
- With a repeat factor, these keys can move the cursor by multiple positions.
- For example:

  10k moves the cursor 10 lines up. Here, 10 is acting as a repeat factor.
  20h moves the cursor 20 characters to the left.

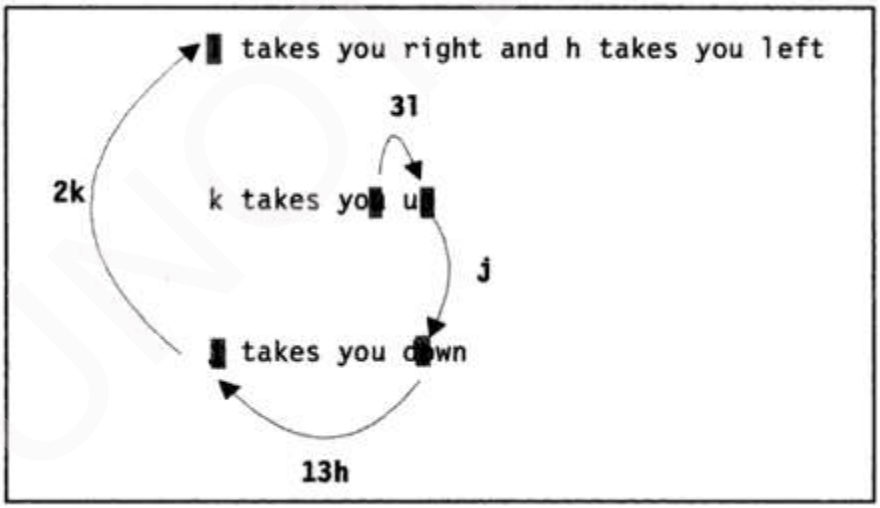


Figure 7.10: Relative navigation with h j k l

3.4.1.2 Word Navigation

- There are 3 word navigation commands:

Command	Function
b	moves back to beginning of word
e	moves forward to end of word
w	moves forward to beginning word

- With a repeat factor, these keys can speed up cursor movement along a line.
- Example:

  5b takes the cursor 5 words backward
  3w takes the cursor 3 words forward

3.4.1.3 Moving to Line Extremes

- Moving to the beginning or end of a line is a common requirement.
- This can be done by 3 commands: 0, | and $.

Command	Function
0 or |	moves cursor to the first character of a line
$	moves to the end of the current line
30|	moves cursor to column 30

- The use of these commands along with b, e, and w is allowed

3.4.1.4 Scrolling

- The two commands for scrolling a page at a time are 1) ctrl-f and 2) ctrl-b.
- Faster movement can be achieved by scrolling the pages of screen.

Command	Function
Ctrl+f	scrolls forward
Ctrl+b	scrolls backward
10ctrl+f	scroll 10 pages and navigate faster
Ctrl+d	scrolls half page forward
Ctrl+u	scrolls half page backward

- With a repeat factor, these keys can speed up scrolling pages.

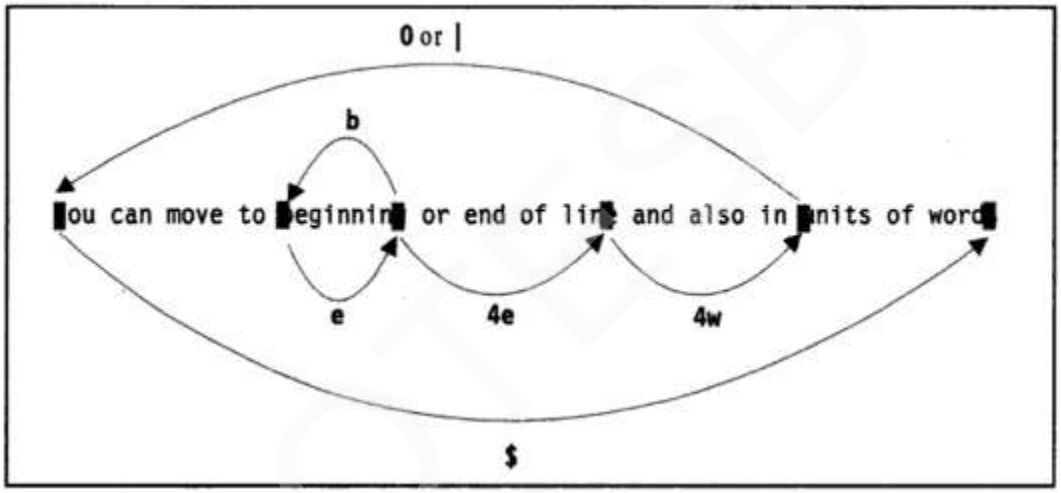


Figure 7.11: navigation along a line

3.4.1.5 Absolute Movement

- The editor displays the total number of lines in the last line.

Command	Function
Ctrl-g	to know the current line number
40G	goes to line number 40
1G	goes to line number 1
G	goes to end of file

3.4.2 Editing Text

3.4.2.1 Deleting Commands (x and dd)

- Following commands can be used to delete characters and lines in an open file.

Command	Function
x	delete a single character under the cursor
dd	delete entire line
4x	deletes the current character and 3 characters from the right.
10dd	deletes the current line and 9 lines below

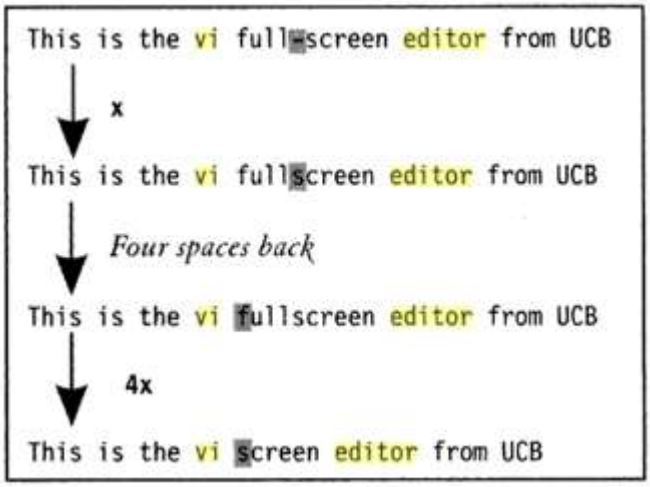


Figure 7.12: deleting text with x

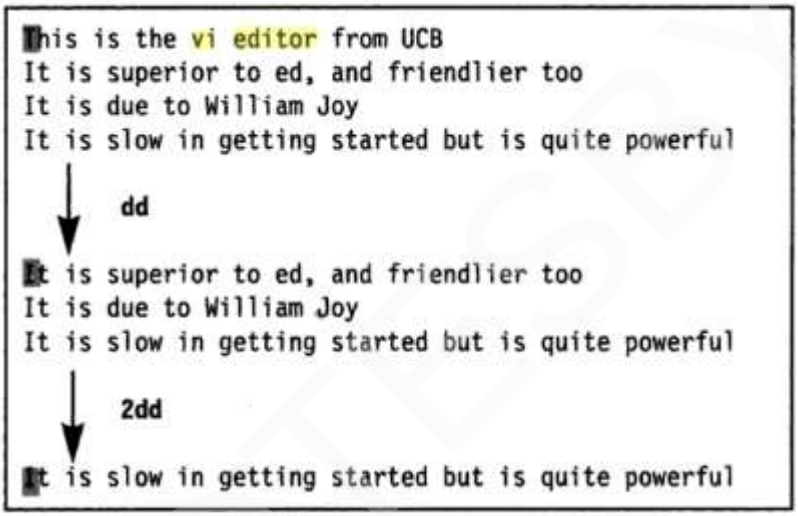


Figure 7.13: deleting line with dd

3.4.2.2 Copy and Paste Commands

Moving Text (p and P)

- Text movement requires you to perform an additional tasks: Put the text at the new cursor location.
- Normally, "put" operation follow delete or copy operation.
- When used with text:
  1) To put the copied text after the cursor, press the key "p".
  2) To put the copied text before the cursor, press the key "P".
- When used with entire line:
  1) To put entire line below the current line, press the key "p".
  2) To put entire line above the current line, press the key "P".

Copying Text (y)

- The term yanking means copying.
- Extracting a copy of the required text is known as yanking.
- This copied text has to be placed at the new location using the put commands: p and P.
- Some of the command-mode commands are:

Command	Function
y	copies the single character
yy	copies the current line
10yy	copies the current line & below 9 line.

3.4.2.3 Undoing Last Editing Instructions

Command	Function
u	To undo the last change made
U	To discard all changes made to the current line
10u	To reverse your last 10 editing actions
ctrl-r	to redo your undone actions

• The undoing limit is set by the execute mode command:
set undolevels=n,
where n is set to 1000 by default

3.4.3 Repeat Commands

• The . (dot) command can be used for repeating the last instruction in both editing and command-mode commands.
• Use the actual command only once, and then repeat the same command at other places with the dot command.
• For example:
2dd deletes 2 lines from current line and to repeat this operation, type. (dot)
• The dot command can be used to repeat only the most recent editing operation- insertion, deletion or any action modifies buffer.
• The dot command is not applicable to navigation, and searching options.
• Search commands (/ and ?) cannot be repeated using the (.) command as these commands do not make changes to the editor buffer.

3.4.4 Search Commands

• A pattern can be searched in a file using following 2 commands.

Command	Function
/Pattern	searches forward for pattern in the file.
?Pattern	searches backward for pattern in the file.

1) Forward Search

• Syntax:
/ Pattern [Enter]
• The search begins forward to position the cursor on the first instance of the pattern.
• Entire file will be searched for the pattern.
• If the pattern can't be located until the end of file, the search continues from the beginning of the file.
• If the search still fails, the message "Pattern not found" will be displayed.

2) Backward Search

• Syntax:
? Pattern [Enter]
• The search begins backward to position the cursor on the first instance of the pattern.
• If the pattern can't be located until the beginning of file, the search continues from the end of the file.
• If the search still fails, the message "Pattern not found" will be displayed.

3.4.4.1 Repeating the Last Pattern Search

• The previous search command can be repeated using following 2 commands:

Command	Function
n	repeats the previous search command in the same direction
N	repeats the previous search command in the opposite direction

• n repeats search in same direction of original search.
• n doesn't necessarily repeat a search in the forward direction. The direction depends on the search command used.
• If you used "?Pattern [Enter]" to search in the reverse direction in the first place, then n also follows the same direction.
• In this case, N will repeat the search in the forward direction.
• N can be used to retrace the search path of n.

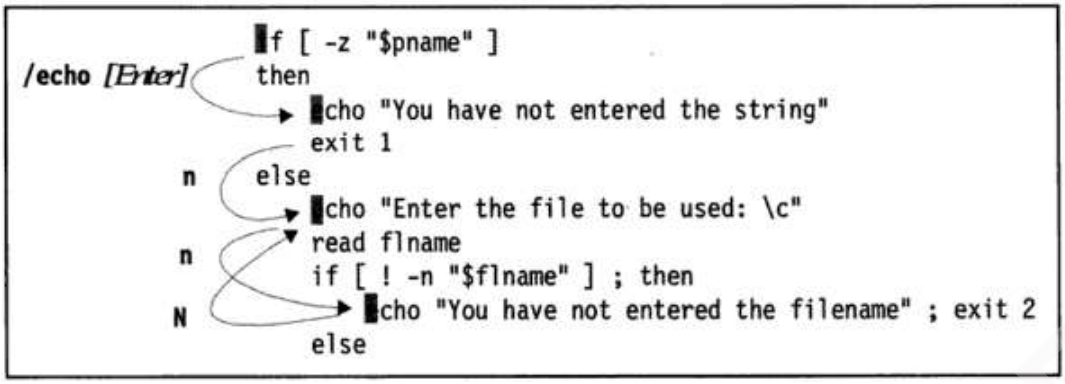


Figure 7.14: search and repeat with / and n

3.4.5 Substitution Commands

- The substitution command (:s/) can be used to replace words or groups of words within a files.
- Syntax:

  : address s/source_pattern/target_pattern/g
- Here, i) The source_pattern is replaced with target_pattern in all lines specified by address.

  ii) Address can be one or a pair of numbers.

  eg: 1,10 -> addresses all lines in a file i.e. from 1st line to 10th line.

  iii) g(global) is a flag which carries out the substitution for all occurrences of the pattern in line. If we omit the "g" flag , the substitution will be carried out only for the first occurrence in each addressed lines.

  iv) The target pattern is optional. If target pattern is not mentioned then it will delete all instances of the source pattern.
- The symbols dot "." and dollar "$" have special significance:

  1) "." represents the current line and

  2) "$" represents the last line of the file

  3) "%" represents the entire file.
- Example:

Command	Function
:1,10s/director/member/g	substitute "director" by "member" in lines from 1 to 10
:.s/director/member/g	substitute "director" by "member" only in the current line
:11,$s/director/member/g	substitute "director" by "member" in lines from 11 to end of file
:%s/director/member/g	substitute "director" by "member" in entire file
:$s/director/member/g	substitute "director" by "member" only in last line
:1,50s/unsigned/g	deletes "unsigned" in lines 1 to 50

3.4.5.1 Interactive Substitution

- c(confirmatory) is a flag which selectively replaces a string.
- Example:

  :1,10s/director/member/gc
- Each line is selected in turn, followed by a sequence of carets in the next line, just below the pattern that requires substitution.
- The cursor is positioned at the end of this caret sequence, waiting for your response.
- The user has 2 choices:

  i) Yes (y or Y) to make the replacement

  ii) No answer (n or N) to not to make the replacement and the search continues.
- Example:

```
9876|jai sharma       |director |production|03/12/50|7000
                       ^^^^^^^y
2365|barun sengupta   |director |personnel |05/11/47|7800
                       ^^^^^^^n
1006|chanchal singhvi |director |sales     |09/03/38|6700
                       ^^^^^^^y
6521|lalit chowdury   |director |marketing |09/26/45|8200
                       ^^^^^^^n
```

3.5 File .exrc

• .exrc is a file located in $HOME/
• .exrc holds many Ex-Mode commands and a series of set-commands.
• Execution of these commands provides a suitable working environment with the vi.
• The vi reads file .exrc on startup.
• This file helps to create abbreviations, redefine the keys to behave differently and also make variable settings.
• Typical entries of a .exrc file is shown below:

```
$cat .exrc
set number
ab      p printf
ab      s scanf
map  x :wq^M
```

• When file is opened in vi editor, the .exrc file will be read and all the commands present in it will be executed, thus building up the required editing environment.
• Settings commonly maintained in the .exrc file are:

1) Set commands : Set commands used in vi like set list, set number, etc.
2) Abbreviations: Frequently used words can be abbreviated. For example,say, a user uses the word 'include' lot many times. The user can create an abbreviation 'inc' for the word include.
3) Mapping keys: Some key combinations can be mapped or made into a hot-key combination

• These settings remain permanent and does not change from one session to another session.

3.6 set map and ab Commands

3.6.1 set command

• This command can be used to customize the vi environment. i.e. we can change characteristics of your editing environment.
• This command can be used to change the look and feel of vi.
• For example,
1) line number can be made to appear automatically
2) current mode of the vi editor can be made to be displayed automatically
3) case sensitivity can be removed during pattern matching.

Command	Function
:set ic	ignores case when searching
:set ai	Sets autoindent
:set noai	To unset autoindent.
:set nu	Displays lines with line numbers on the left side.
:set ro	Changes file type to "read only"

• :set number can be used to set the line number option.
• When this option is set, line numbers appear automatically.
• By default, no line number appears in vi editor.
• :set nonumber can be used to remove the line number option.
• :set showmode can be used to display the mode in which the editor is present currently.
• :set noshowmode can be used to cancel automatic displaying of the modes.
• :set autoindent can be used to automatically indent every new line keyed in by the user with one or several tabs.
• :set noautoindent can be used to cancel auto indenting.
• Options that are set using set commands are applicable only to those sessions in which they are given.
• If the user wants the settings to be applicable permanently, relevant settings must be set using corresponding commands in the .exrc file.

3.6.2 map

• This command can be used to
→ assign existing vi commands to a custom key or
→ define own custom commands.
• This command can be used to connect one or more commands to a single key. This process is known as mapping.
• This command allows the user to perform complex editing tasks with a single keystroke.
• Syntax:

```
map x sequence                  //define x as a sequence of editing commands
```

• For example:

```
map Q :q!                       // this command maps the Q key with the quit operation.
```

• So, whenever the user wants to quit the file, instead of typing ':q!', the user can simply type 'Q' from the escape mode.

3.6.3 ab

• This command is also an Ex-Mode command.
• This command is used to have short hand abbreviations for commonly used expressions or texts.
• Syntax:

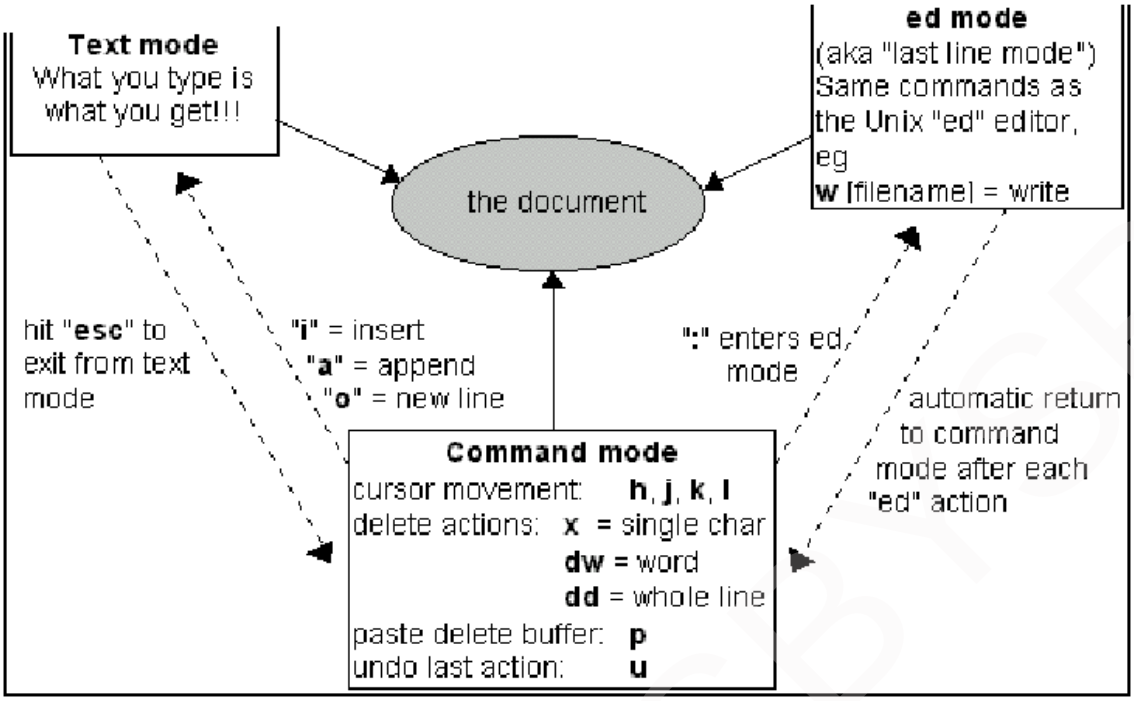
```
:ab abbr phrase
```

• Here, abbr is an abbreviation for the specified phrase.
• For example:

```
:ab p printf                    // 'p' is an abbreviation for printf
:ab s scanf                     // 's' is an abbreviation for scanf
```

• Here, when the "p" followed by the enter key is used, the text "printf" appears at the place of "p".
• Frequently used words can be abbreviated.
• The abbreviations used during a session are valid only for that session.
• If the user wants the abbreviations to be applicable permanently then corresponding abbreviations entries must be made in .exrc file.

The diagram below illustrates methods of moving between each mode, and some of the keys available in each mode.



Summary of some basic commands for vi:

Adding new text (from command-mode into insert mode)

i insert new text in front of cursor	o open a new line after (below) cursor
a insert new text after cursor	O open a new line before (above) cursor
A insert new text at the end of the line	

Moving around without the arrow keys (within command-mode)

k move backwards (up) one line	
h move backwards one character	l (lower case L) move forward one char
b move backwards one word	j move down (forward) one line
^ move to the start of a line	$ move to the end of a line

Editing text (from command-mode)

x delete one character (under cursor)	dd cut (delete) current line
r replace character (under cursor)	yy copy (yank) current line
u undo last editing command	p paste most recently cut/copied text

Search and Replace (from Ex-Mode)

/text search forward for next "text"	?text search backwards for previous "text"
:s/old/new/ replace old with new once	
:s/old/new/g replace all occurrences of old with new on current line only	:1,$s/old/new/g replace all occurrences of old with new on all lines from 1 to $ (end)

Save and Exit (from Ex-Mode;

:w filename write as filename	:q quit, exit, leave vi
:w write using existing filename	:q! quit, discarding changes

MODULE 3 (CONT.): THE SHELL

3.7 Shells Interpretive Cycle

• The shell performs following activities in its interpretive-cycle:

1) The shell
   → issues the prompt ($) and
   → waits for user to enter a command (like ls chap*).
2) After a command is entered, the shell
   → scans the command-line for metacharacters (like 'ls chap*') and
   → expands the abbreviations to recreate a simplified command-line ('ls chap1 chap2').
3) Then, the shell
   → passes the command-line to the kernel for execution and
   → waits for the command to complete its task.
4) After the command is executed, the shell
   → issues the prompt ($) again and
   → waits for the user to enter a next command.

3.8 Wild Cards and Filename Generation

• The metacharacters that are used to construct the generalized pattern for matching filenames belong to a category called wild-cards.

Wild Card/ Character class	**Match**
*	Any number of characters including none
?	A single character
[ijk]	A single character either an i, j or k
[x-z]	A single character that is within the ASCII range of the characters x and z
[!ijk]	A single character that is not an i, j, or k
[!x – z]	A single character that is not within the ASCII range of the characters x and z

3.8.1 Metacharacters * and ?

• The metacharacter "*" matches any number of characters including none.
• Examples:

```
$ ls chap*      // To list all files that begin with "chap"
chap chap01 chap02 chap03 chap04 chapx chapy chapz
```

• When the shell encounters this command line, it identifies the * immediately as a wild-card. It then looks in the current directory and recreates the command line as below

```
$ ls chap chap01 chap02 chap03 chap04 chapx chapy chapz
```

• The shell now hands over this command to the kernel which uses its process creation facilities to run the command.
• The metacharacter ? matches a single character.
• Example:

```
$ ls chap?              // To list all five-characters filenames beginning with "chap"
chapx chapy chapz
$ ls chap??             // To list six-characters filenames beginning with "chap"
chap01 chap02 chap03
```

• Both * and ? operate with some restrictions.
• The * doesn't match all filenames beginning with a dot (.) or forward slash(/).
• Example:

```
$ ls .C*                // to list all C extension filenames
$ cd /usr?local         // this doesn't match /usr/local
```

3.8.2 Character Class

- The character class comprises a set of characters enclosed by the rectangular brackets, [and],.
- The character class matches a single character in the class.
- For example:

Character Class	Match
[ijk]	A single character either an i, j or k
[x-z]	A single character that is within the ASCII range of the characters x and z.
chap0[124]	chap01, chap02, chap04
[!ijk]	A single character that is not an i, j, or k
[!x – z]	A single character that is not within the ASCII range of the characters x and z

- This can be combined with any string or another wild-card expression.
- Example:

```
$ ls chap0[124]      //Matches chap01, chap02, chap04 and lists if found.
$ ls chap[x-z]       //Matches chapx, chapy, chapz and lists if found.
```

Negating the Character Class (!)

- Not operator (!) can be used to negate the character class.
- For example,

```
$ls  [!a-zA-Z]* //To match all filenames that don't begin with an alphabetic character.
```

Matching Totally Dissimilar Patterns

```
$ cp $HOME/prog_sources/*.{c,java} .  // To copy all the C and Java source programs from
                                      //another directory to the current directory
$ cp /home/srm/{project,html,scripts/* .    // To  copy all files from 3 directories (project,
                                      // html and scripts) to the current directory.
```

3.9 Removing the Special Meanings of Wild Cards (Escaping and Quoting)

- Escaping is providing a \ (backslash) before the wildcard to remove its special meaning.
- Example:

```
$ rm chap\*      // to remove file named "chap*" "\" is used to suppress special meaning of *
$ cat chap0\[1-3\]                  // to list the contents of the file named "chap0[1-3]"
$ rm My\ Document.doc              //  to remove file named "My Documend.doc"
$ echo \\                      // outputs \
```

- Quoting is enclosing the wild-card within quotes to remove its special meaning.
- When a command argument is enclosed in quotes, the meanings of all enclosed special characters are turned off.
- Example:

```
$ rm 'chap*'                  // to remove  file named "chap*"
$ rm "My Document.doc"        //  to remove  file named "My Documend.doc"
$ echo "\"                    // outputs "\"
```

3.10 Redirection : Three Standard Files

- The shell associates three standard files with the terminal:
  - → two for display and
  - → one for the keyboard.
- When a user logs in, the shell makes available three standard files.
- Each standard file is associated with a default device:
  1) Standard input: The file representing input which is connected to the keyboard.
  2) Standard output: The file representing output which is connected to the display.
  3) Standard error: The file representing error messages that come from the command or shell. This file is also connected to the display.

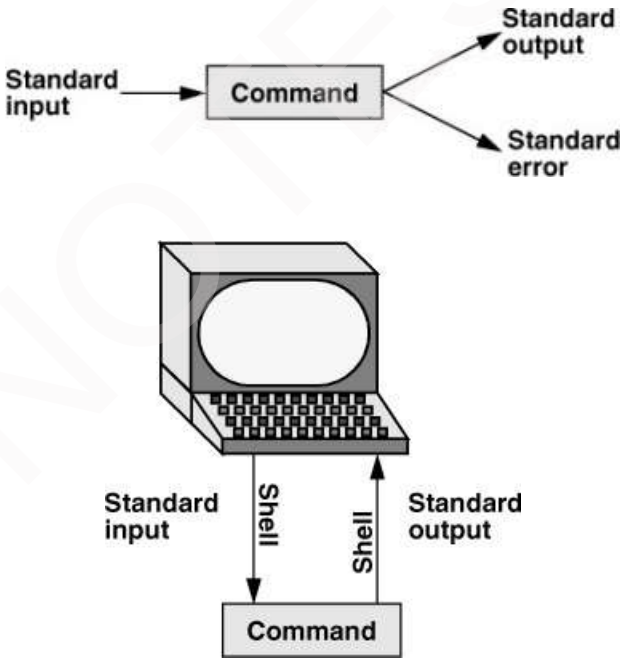


Figure 3.10: Default three Standard Files

3.10.1 Standard Input

- The standard input can represent three input sources:
  1) The keyboard, the default source.
  2) A file using redirection with the < symbol.
  3) Another program using a pipeline.
- By default, the shell directs standard input from the keyboard.
- Example:

```
$wc
hello
world
<ctrl+d>                    // end of input
2 2 10                      // output
```

- The redirect input symbol (<) instructs the shell to redirect a command's input to come from the specified file instead of from the keyboard.
- Example:

```
$ cat sample.txt
hello
world
$ wc < sample.txt
2 2 10                      // output
```

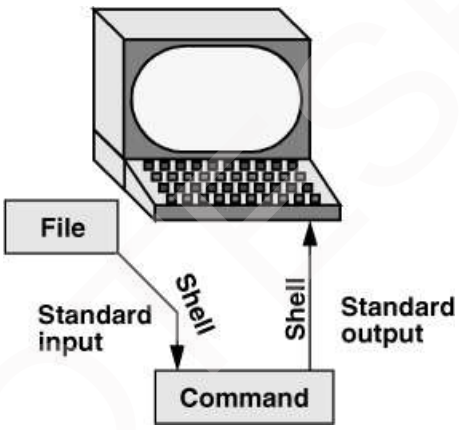


Figure 3.10.1 : Redirecting input

3.10.2 Standard Output

- The standard output can represent three possible destinations:
  1) The terminal, the default destination.
  2) A file using the redirection symbols > and >>.
  3) As input to another program using a pipeline.
- By default, the shell directs standard output from a command to the screen.
- Example:

```
$ cat sample.txt
hello
world
$ wc  sample.txt
2 2 10                      // output
```

- The redirect output symbol (>) instructs the shell to redirect the output of a command to the specified file instead of to the screen.
- Example:

```
$ wc  sample.txt > temp.txt
$ cat temp.txt
2 2 10                      // output of wc stored in temp.txt
```

- >> can be used to append to a existing file.

3.10.3 Standard Error

• By default, the shell directs standard error from a command to the screen.
• Example:

```
$ cat empty.txt
cat: cannot open empty.txt                    // error because empty.txt is non existent file
```

• The redirect output symbol (>) instructs the shell to redirect the error messages of a command to the specified file instead of to the screen.
• Example:

```
$ cat empty.txt >> errorfile.txt
$ cat errorfile.txt
cat: cannot open empty.txt                    // error message of cat stored in errorfile.txt
```

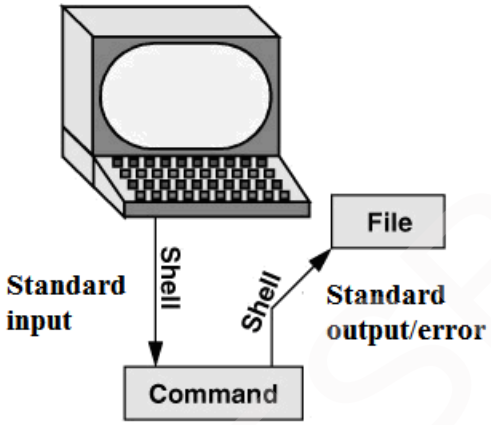


Figure 3.10.2 : Redirecting Output & Error

3.10.4 Filters: Using Both Standard Input and Standard Output

• UNIX commands can be grouped into four categories:
1) Directory-oriented commands like mkdir, rmdir and cd, and basic file handling commands like cp, mv and rm use neither standard input nor standard output.
2) Commands like ls, pwd, who etc. don't read standard input but they write to standard output.
3) Commands like lp that read standard input but don't write to standard output.
4) Commands like cat, wc, cmp etc. that use both standard input and standard output.
• Commands like cat, wc, cmp etc. that use both standard input and standard output are called filters.
• Example:

```
$ cat calc.txt
(4*2)+2
$ bc < calc.txt > result.txt
$ cat result.txt
10
```

• Here, this command performs arithmetic calculations that are specified as expressions in input file "calc.txt" and redirect the output to a file "result.txt".
• Running 2 or more commands have following disadvantages:
1) The process is slow. The second command cant act unless the first has completed its job.
2) You require an intermediate file that has to be removed after the first command has completed its run.
3) When handling large files, temporary files can build up early and eat up disk space in no time

3.11 Pipe : Connecting Commands

- Pipe is another form of output redirection.
- With piping, the output of a command can be used as input (piped) to a subsequent command.
- Syntax:

  $ command1 | command2
- Here, Output from command1 is piped into input for command2.
- The symbol '|' denotes a pipe.
- Example:

  $ ls -l | lp
- Here pipe is used to send the output of ls to the lp to print a hard copy of the listing of the current directory.
- Pipeline can be used with 2 or more commands. But the user should know the behavioral properties of each commands to place them there.
- Example:

  $ls -l | wc -l | lp // to print a count of files in current directory

When a Command Needs to be Ignorant of its Source

- If we wish to find total size of all C programs contained in the working directory, we can use the command:

  $ wc –c *.c
- However, it also shows the usage for each file(size of each file). We are not interested in individual statistics.
- Solution: We must make wc ignorant of its input source. So, feed the concatenated output stream of all the .c files to wc –c as its input:

  $ cat *.c | wc –c

3.12 tee : Splitting the Output

- This is an external command that handles a character stream by duplicating its input.
- This command
  - → saves one copy in a file and
  - → writes the other copy to standard output.
- This command is also a filter and hence can be placed anywhere in a pipeline.
- Example:

  $ who | tee users.lst //to display the output of who and save this output in a file users.lst

3.13 Command Substitution

- Command substitution allows to substitute the output of one command into a given command line.
- Syntax:

  `command`
- Here, the command must be enclosed between backquotes.
- Example:

```
$ echo Current date and time is `date`
Current date and time is Sat Nov 24 10:12 IST 2017
$ echo "There are `ls | wc –l` files in the current directory"
There are 3 files in the current directory
```

MODULE 3 (CONT.): FILTERS USING REGULAR EXPRESSION

3.14 grep

- g/re/p means "globally search for a regular expression and print all lines containing it".
- This command can be used to search a file(s) for lines that have a certain pattern.
- This command
  - → scans the file for a pattern and
  - → displays
    - i) lines containing the pattern or
    - ii) line numbers
- Syntax:
  - grep pattern file(s)
- Example:
  - grep "MH" student.lst // display lines containing "MH" from the file student.lst
- Patterns with and without quotes is possible.
- Quote is mandatory when pattern involves more than one word.
- Example:
  - grep "My Document" student.lst // display lines containing "My Document" from student.lst
- This command can be used with multiple filenames.
- Example:
  - grep "MH" student.lst vtu.lst rank.lst

3.14.1 grep Options

- Linux supports below listed options:

-i	ignores case for matching
-v	doesn't display lines matching expression
-n	displays line numbers along with lines
-c	displays count of number of occurrences
-l	displays list of filenames only
-e	Exp specifies expression with this option
-x	matches pattern with entire line
-f	file takes patterns from file, one per line
-E	treats pattern as an extended RE
-F	matches multiple fixed strings

- To understand the working of different options, let us consider we have following information in file student.lst.

```
$ cat student.lst
4       | MH   | 10    | IS    | 111
4       | MH   | 11    | CS    | 401
4       | GW   | 11    | CS    | 402
4       | VV   | 11    | CS    | 403
```

Ignores Case

- -i (ignore) option can be used to search all lines containing a pattern regardless of uppercase and lowercase distinction
- Example:

```
$ grep -i "MH" demo_file          // matches all the words such as MH mH Mh mh
4       | MH   | 10    | IS    | 111
4       | MH   | 11    | CS    | 401
```

Deleting Lines

- -v option can be used to print all lines that do not contain the specified pattern in a file.
- Example:

```
grep -v 'MH' student.lst
4       | GW   | 11    | CS    | 402
4       | VV   | 11    | CS    | 403
```

Displaying Line Number

- -n (line number) option can be used to display line numbers containing the pattern.
- Example:

```
grep -n 'MH' student.lst
1
2
```

Counting Lines Containing the Pattern

- -c (line count) option can be used to count number of lines containing the pattern.
- Example:

```
grep -c 'MH' student.lst
2
```

Displaying Filenames

- -l (list filename) option can be used to list out the files containing the pattern.
- Example:

```
grep -l 'MH' *.lst
```

Matching Multiple Pattern

- -e option can be used to match multiple pattern in a file.
- Example:

```
grep -e 'MH' -e 'VV' student.lst
```

Taking Pattern From File

- -f option can be used to place all matched pattern in a separate file, one pattern per line.

3.15 Basic Regular Expression (BRE)

- grep uses an expression of a different type to match a group of similar patterns.
- This command
  - → uses an elaborate metacharacter set and
  - → can perform amazing matches.
- If an expression uses metacharacters, it is termed a regular expression.
- Regular expression can be classified as
  1) BRE (Basic Regular Expression) and
  2) ERE (Extended Regular Expression)
- grep supports
  - → BRE by default and
  - → ERE with the –E option.
- Character subset of BRE is listed below:

Metacharcter/ Character Class	**Match**
*	Zero or more occurrences
g*	nothing or g, gg, ggg, etc
.	A single character
.*	nothing or any number of characters
[pqr]	a single character p, q or r
[c1-c2]	a single character within the ASCII range represented by c1 and c2
^[pqr]	a single character which is not p, q or r
^Pattern	Pattern at beginning of line
Pattern$	Pattern at end of line
^$	Line containing nothing

3.15.1 Character Class

- The character class comprises a set of characters enclosed by the rectangular brackets, [and].
- The character class matches a single character in the class.
- When you use range, make sure that the character on the left of the hyphen has a lower ASCII value than the one on the right.
- caret (^) can be used to negate the character class.
- For example:

Character Class	**Match**
[pqr]	a single character p, q or r
[x-z]	A single character between characters x to z.
^[pqr]	a single character which is not p, q or r

- Example:

```
$ grep -i "[Mm][Hh]" demo_file          // matches all the words such as MH mH Mh mh
```

3.15.2 Asterisk (*)

- The asterisk (*****) refers to the immediately preceding character.
- It indicates zero or more occurrences of the previous character.
  - g* → nothing or g, gg, ggg, etc.
  - lg* → l or lg, lgg, lggg, etc.
- Example:

```
$ grep  "isaa*c" demo_file          // matches isac isaac or isaaac
```

3.15.3 Dot (.)

- A dot (.) matches a single character.
- Example:

```
$ grep  "2..." demo_file            // matches all 4 character words beginning with 2
```

- Regular expression "**.***" → signifies any number of characters or none
- Example:

```
$ grep  prog.c.* demo_file          // matches all c and cpp extension filenames
```

3.15.4 Specifying Pattern Locations (^ and $)

• Following two metacharcters can match a pattern at the beginning or end of a line.

Metacharcter	Match
^Pattern	Pattern at beginning of line
Pattern$	Pattern at end of line

• Anchoring a pattern is often necessary when it can occur in more than one place in a line, and we are interested in its occurrence only at a particular location.

• Example:

```
grep "^2" emp.lst          //Selects lines starting with 2
grep "7…$" emp.lst         //Selects lines where salary between number b/w 7000 to 7999
```

3.16 Extended Regular Expression (ERE) and egrep

• ERE can be used to match dissimilar patterns with a single expression.

• This uses some additional characters as listed below:

Metacharcter/ Character class	Match
ch+	one or more occurrences of character ch
ch?	zero or one occurrence of character ch
exp1|exp2	exp1 or exp2
(x1|x2)x3	x1x3 or x2x3

• If current version of grep doesn't support ERE, then use egrep but without the –E option.

• -E option treats pattern as an ERE.

• Example:

```
$ grep  -E  "isaa*c" demo_file                    // matches isac isaac or  issaac
$ grep –E  'vijaykumar | jayakumar' demo_file     // matches multiple patterns
$ grep –E  '(vijay | jaya) kumar' demo_file       // matches multiple patterns
```

MODULE 4: SHELL PROGRAMMING MORE FILE ATTRIBUTES SIMPLE FILTERS

MODULE 4: SHELL PROGRAMMING

4.1 Shell Programming

- A shell script contains a list of commands which have to be executed regularly.
- Shell script is also known as shell program.
- The user can execute the shell script itself to execute commands in it.
- A shell script runs in interpretive mode. i.e. the entire script is compiled internally in memory and then executed.
- Hence, shell scripts run slower than the high-level language programs.
- ".sh" is used as an extension for shell scripts.
- Example: A shell script (program1.sh) to execute few commands.

```
#! /bin/sh
echo "Welcome to Shell Programming"        # print message
echo "Today's date : `date`"               # print date
echo "My Shell :$SHELL"                    # print shell name
```

- The hash symbol # indicates the comments in the script.
- The shell ignores all the characters that follow the # symbol. However, this does not apply to the first line.
- The first line
  "#! /bin/sh" indicates the path where the shell script is available.
- There are 2 ways to execute a shell script:

1) Execute Shell Script Using File Name

- By default, script is not executable. So, the chmod command can be used to make the script executable.
- The scripts are executed in a separate child shell process.
- The child shell reads and executes each statement in interpretive mode.

```
Run:
$ chmod +x program1.sh                  // add executable permission
$ program1.sh                           // execute the script program1.sh
Output:
Welcome to Shell Programming
Today's date: Mon Nov 4 11:02:45 IST 2017
My Shell: /bin/sh
```

2) Execute Shell Script by Specifying the Interpreter

- The user can also execute a shell script by specifying the interpreter in the command line.
- Here, the script neither requires a executable permission nor an interpreter line.

```
Run:
$ sh program1.sh                        //Execute using sh interpreter
$ bash program1.sh                      //Execute using bash interpreter
Output:
Welcome to Shell Programming
Today's date: Mon Nov 4 11:02:45 IST 2017
My Shell: /bin/sh
```

4.2 Ordinary and Environment Variables

• Shell variables are of 2 types: 1) Environment and 2) Ordinary

4.2.1 Environment Variable

• Environmental variables are used to provide information to the programs you use.
• These variables control the behavior of the system.
• They determine the environment in which the user works.
• If environment variables are not set properly, the users may not be able to use some commands.
• Environment variables are so called because they are available in the user's total environment
i.e. the sub-shells that run shell scripts and mail commands and editors.
• Some variables are set by the system, others by the users, others by the shell programs.
• env command can be used to display environment variables.
• For example:

```
$ env
HOME=home/kumar
IFS='  '
LOGNAME=kumar
MAIL= /var/mail/kumar
MAILCHECK=60
PATH=/bin:/usr/bin
PS1='$'
PS2='>'
SHELL=/usr/bin/bash
TERM= tty1
```

1) HOME

• This variable indicates the home directory of the current user.
• This variable is set for a user by the system admin in /etc/passwd.

2) IFS

• This variable contains a string of characters that are used as word separator in the command line.
• The string normally consists of the space, tab and newline characters.

3) LOGNAME

• This variable shows the username.

4) MAIL

• This variable specifies the path to user's mailbox.

5) MAILCHECK

• This variable determines how often the shell checks the file for the arrival of new mail.

6) PATH

• This variable specifies the locations in which the shell should look for commands.
• Usually, the PATH variable can be set as follows:
$PATH=/bin:/usr/bin

7) PS1 and PS2

• The shell has 2 prompts:
1) The primary prompt $ is the one the user normally sees on the monitor. $ is stored in PS1.
➢ The user can change the primary prompt as follows:
$ PS1="C>"
C> //similar to windows
2) The secondary prompt > is stored in PS2.

8) SHELL

• This variable specifies the current shell being used by the users.
• Different types of shells are:
1) Bourne shell /bin/sh 2) C-shell /bin/csh 3) Korn shell /bin/ksh
• This variable is set for a user by the system admin in /etc/passwd.

9) TERM

• This variable indicates the terminal type that is used.
• Every terminal has certain characteristics that are defined in a separate control file in the terminfo directory.
• If TERM is not set correctly, vi will not work and the display will be faulty.

4.2.2 Ordinary(or Local) Variable

- A variable is a character string to which the user assigns a value.
- The value assigned can be a number, text, filename, device, or any other type of data.
- Syntax:

  variable = value // variable definition
- The value of variables are stored in the ASCII format.
- For example:

  $ x=50

  $ echo $x //displays 50
- In command line, all words that are preceded by a $ are identified and evaluated as variables.
- A variable can be removed with unset and protected from reassignment by readonly. Both are shell internal commands.

  $ set count=5

  $ readonly size = 10
- The variables exist only for a short time during the execution of a shell script.
- The variables are local to the user's shell environment.
- The variables are not available for the other scripts or processes.
- As the variables are defined and used by specific users, they are also called user-defined variables.

Uses of Local variables

1) Setting pathnames: If a pathname is used several times in a script, we can assign it to a variable and use it as an argument to any command.

2) Using command substitution: We can assign the result of execution of a command to a variable. The command to be executed must be enclosed in backquotes.

3) Concatenating variables and strings: Two variables can be concatenated to form a new variable.

Example:
```
$ base=foo ; ext=.c
$ file=$base$ext
$ echo $file                    // prints foo.c
```

4.3 File .profile

- A profile file is a start-up file of an UNIX user.
- This file gets executed as soon as the user logs in.
- This file is a shell script that will be present in the home directory of each user.
- The system admin provides each user with a profile with a minimum working environment.
- However, the user can customize the profile as per their requirement.

  i.e. The user can

  → assign suitable values to the environment variables.

  → add and modify statements in the profile file.
- This file can be any one of the two:

  1) A specific file for each individual user with responsibility for the user environment.

  2) A universal file for all users with responsibility for the general environment.
- The user can view his ".profile" as follows:

```
$ cat .profile
MAIL= /var/mail/kumar
PATH=/bin:/usr/bin
PS1='$'
PS2='>'
SHELL=/usr/bin/bash
TERM= tty1
```

4.4 read and readonly Commands

4.4.1 read Command

- read command can be used for taking input from the keyboard.
- It is shell's internal tool for making scripts interactive.
- Syntax:

  read var_name
- It is used with one or more variables.
- The variables are used to hold inputs given with the standard input.
- Example: A shell script (program4.sh) to read a search string and filename from the terminal.

```
#!/bin/bash
echo "What is your name?"
read PERSON
echo "Hello, $PERSON"
```

```
Run:
$ program4.sh
Output:
What is your name? RAMA
Hello, RAMA
```

4.4.2 readonly Command

- readonly command can be used to make variables readonly i.e. the user cannot change the value of variables.
- During shell scripting, we may need a few variables, which cannot be modified.
- This may be needed for security reasons.
- Syntax:

  variable=value
- For example:

```
$ readonly PI=3.14
$ echo $PI                    //displays 3.14
$ PI=6.12                     // this will result in error
```

4.5 Command Line Arguments

- Shell scripts can accept arguments from the command line.
- ,'. Shell scripts can be run non-interactively and be used with redirection and pipelines.
- The arguments are assigned to special shell variables called shell parameters.
- The shell parameters are reserved for specific functions.
- Different shell parameters:

1) $#: Stores the number of command-line arguments.
2) $0, $1, $2, $3: These are called positional parameters which represent command line arguments.
   - $0: Stores the filename of the current script.
   - $1: Stores the first argument.
   - $2: Stores the second argument
   - $3: Stores the third argument
3) $*: Stores all the arguments entered on the command line ($1 $2 ...).
4) "$@": Stores all arguments entered on the command line, individually quoted ("$1" "$2")
5) $?: Stores the exit status of the last command that was executed.
6) $$: Stores Pid of the current shell.
7) $!: Stores PID of the last background job.

- Example: A shell script (program2.sh) to read and display various shell parameters from the command line.

```
#!/bin/sh
echo "Total Number of Parameters : $#"
echo "File Name: $0"
echo "First Parameter : $1"
echo "Second Parameter : $2"
echo "Quoted Values: $*"
echo "Quoted Values: $@"
$echo "Exit value: $?"
echo "PID of current shell: $$"
```

```
Run:
$ program2.sh "RAJA RAM" "MOHAN ROY"
Output:
Total Number of Parameters : 2
File Name : program2.sh
First Parameter : RAJA RAM
Second Parameter : MOHAN ROY
Quoted Values: RAJA RAM MOHAN ROY     // stored as "RAJA RAM MOHAN ROY"
Quoted Values: RAJA RAM MOHAN ROY     // stored as array= {"RAJA RAM" , "MOHAN ROY"}
Exit value: 0                         // zero implies success
PID of current shell: 12345
```

4.6 exit and Exit Status of a Command

- exit command can be used to terminate a program(or script).
- This command returns value which will be available to the script's parent process.
- The $? variable contains exit status of the last command executed.
- Exit status is a numerical value returned by every command upon its completion.
- A command returns an exit status of
  1) zero (0) upon successful execution and
  2) non-zero upon unsuccessful execution i.e. an error condition.
- Exit status can be used to devise program-logic that branches into different paths depending on success or failure of a command.
- Example: A shell script to find relationship between 2 numbers.

```
#! /bin/usr
x=5; y=7
test $x –eq  $y; echo "5=7: $? \n"
test $x –ne  $y; echo "5!=7: $? \n"
test $x –gt  $y; echo "5>7: $? \n "
test $x –ge  $y; echo "5>=7: $? \n "
test $x –lt   $y; echo "5<7: $? \n "
test $x –le  $y; echo "5<=7: $? \n"
```

```
Output:
5=7: 1                  // Returns nonzero exit status i.e. failure → False
5!=7: 0                 // Returns zero exit status i.e.  success → True
5>7: 1                  // False
5>=7: 1                 // False
5<7: 0                  // True
5<=7: 0                 // True
```

4.7 Logical Operators for Conditional Execution

- Two logical operators can be used for conditional execution: 1) && and 2) ||

1) && Operator

- Syntax:

  cmd1 && cmd2

- Here, cmd2 gets executed only when cmd1 succeeds.

2) || Operator

- Syntax:

  cmd1 || cmd2

- Here, cmd2 gets executed only when cmd1 fails.
- Example: A script to illustrate the usage of && and ||.

```
$ cat student.lst
4       | MH   | 10    | IS    | 111
4       | MH   | 11    | CS    | 401
4       | GW   | 11    | CS    | 402
4       | VV   | 11    | CS    | 403
$ grep 'VV' student.lst && echo "Pattern found"
4       | VV   | 11    | CS    | 403
Pattern found

$ grep 'ZZ' student.lst || echo "Pattern not found"
Pattern not found
```

4.8 test Command and its Shortcut

• Usually, if-construct cannot directly handle the true or false value returned by evaluation of an expression.
• So, test command can be used to handle the true or false value returned by evaluation of an expression.
• Test command
→ uses certain operators to evaluate the condition on its right and
→ returns either a true or false exit status.
• Then, if-construct uses the exit status for making decisions.
• Test command
→ does not display any output
→ sets the parameter $? (exit status).
• Test command works in 3 ways:
1) Compare two numbers.
2) Compares two strings or a single one for a null value.
3) Checks files attributes.

4.8.1 Numeric Comparison

Operator	Meaning
-eq	Equal to
-ne	Not equal to
-gt	Greater than
-ge	Greater than or equal to
-lt	Less than
-le	Less than or equal

• Syntax:
test $op1 -operator $op2
• Operators always begin with a – (Hyphen) followed by a two character word .
• Numeric comparison can be done on integer values only. (The decimal values are truncated).

Shorthand for test

• [and] can be used instead of test.
Test $x –eq $y is equivalent to [$x –eq $y]
• Example: A shell script to find relationship between 2 numbers.

```
#! /bin/usr
x=5; y=7
test $x –eq  $y; echo "5=7: $? \n"
test $x –ne  $y; echo "5!=7: $? \n"
test $x –gt  $y; echo "5>7: $? \n "
test $x –ge  $y; echo "5>=7: $? \n "
test $x –lt  $y; echo "5<7: $? \n"
test $x –le  $y; echo "5<=7: $? "
```

```
Output:
5=7: 1                  // False
5!=7: 0                 // True
5>7: 1                  // False
5>=7: 1                 // False
5<7: 0                  // True
5<=7: 0                 // True
```

4.8.2 String Comparison

• Test command is also used for testing strings.

Operator	True if
s1=s2	String s1=s2
s1!=s2	String s1 is not equal to s2
-n stg	String stg is not a null string
-z stg	String stg is a null string
stg	String stg is assigned and not null

• Example: A shell script to check if 2 strings are equal or not.

```
#!/bin/sh
echo "Enter the first string: \c"
read str1
if [ -z "$str1" ] ; then
        echo "You have not entered the string"; exit 1
echo "Enter the second string: \c"
read str2
if [ -z "$str2" ] ; then
        echo "You have not entered the string"; exit 1
if[ $str1= $str2]
then
        echo "Both strings are equal"
else
        echo "Strings are unequal"
```

```
Output:
Enter the first string: MAM
Enter the second string: MAM
Both strings are equal
```

4.8.3 File Tests

• Test command can be used to check various file attributes such as file type (-, d or l) & file permission (r, w, x).

Test	True if
-e file	File exists
-f file	File exists and is a regular file
-d file	File exists and is a directory
-L file	File exists and is a symbolic link
-r file	File exists and readable
-w file	File exists and is writable
-x file	File exists and is executable
-s file	File exists and has a size greater than zero
f1 –nt f2	File f1 is newer than f2
f1 –ot f2	File f1 is older than f2
f1 –ef f2	File f1 is linked to f2

• Example: A shell script (program8.sh) to check whether a file has permission for read, write and execute.

```
#! /bin/usr
echo -n "Enter file name:"
read file
if [–e $file] ;
then
        echo “File exists \n”
else
        echo “File does not exist \n”
fi
if [ -r "$file" ]
then
         echo "File is readable \n "
else
        echo "File is not readable \n "
fi
if [ -w "$file" ]
then
         echo "File is writable \n "
else
        echo "File is not writable \n "
fi
if [ -x "$file" ]
then
         echo "File is executable \n "
else
        echo "File is not executable \n "
fi
```

```
Run:
$ ls –l student.lst
-rw-rw-rw-   1   kumar   group   870   jun   8   15:52   student.lst
$ program8.sh
Output:
Enter file name: student.lst
File exists
File is readable
File is writable
File is not executable
```

4.9 if Statement

- if statement is basically a "two-way" decision statement.
- This is used when we must choose between two alternatives.
- Three forms of if…else statement:
  1) if...fi statement
  2) if...else...fi statement
  3) if...elif...else...fi statement
- Syntax 1:

```
if command is successful
then
        execute statements
fi
```

- Syntax 2:

```
if command is successful
then
        execute statements
else
        execute statements
fi
```

- Syntax 3:

```
if command is successful
then
        execute statements
elif command is successful
  then
        execute statements
  else
        execute statements
fi
```

- Here is how it works:
  1) If the command succeeds, the statements within then-block are executed.
  2) If the command fails, the statements within else-block are executed.
- Example: A script to check whether an integer is positive or negative.

```
#! /bin/sh
echo "Enter any non zero integer: \n"
read num
if [$num -gt 0]; then
        echo "Number is positive number"
else
        echo "Number is negative number"
```

```
Output:
Enter any non zero integer:
5
Number is positive number
```

4.10 case Statement

- case statement is basically a "multi-way" decision statement.
- This is used when we must choose among many alternatives.
- This also handles string tests, but in a more efficient manner than if statement.
- Syntax:

```
case expression in
        pattern1) statement1 ;;
        pattern2) statement2 ;;

        pattern3) statement3 ;;
        …
esac
```

- Here is how it works:
  1) Firstly the expression is matched with pattern1.
  2) If the match succeeds, then statement1 will be executed.
  3) If the match fails, then the expression is matched with pattern2 and this process continues.
- Each statement is terminated with a pair of semicolon (;;).
- This can match only strings but cannot handle numeric and file tests.
  However, this can also handle numbers but treating them as strings.
- This is very effective when the string is fetched by command substitution.
- Example: A script to display appropriate message based on grades (A to D).

```
#! /bin/sh
echo "enter grade A to D \n"
read grade
case "$grade" in
                A) echo "Excellent!" ;;
                B) echo "Well done" ;;
                C) echo "You passed" ;;
                D) echo "Better try again" ;;
                *) echo "Invalid grade" ;;
esac
echo "Your grade is $grade"
```

```
Output:
enter grade A to D
B
Well done
Your grade is B
```

4.10.1 Matching Multiple Patterns

- case statement can also specify the same action for more than one pattern.
- Example: A script to test a user response for both y and Y (or n and N).

```
#! /bin/sh
echo "Do you wish to continue? [y/n]:"
read ans
case "$ans" in
        Y | y ) ;;
        N | n ) exit ;;
esac
```

4.10.2 Wild-Cards

- case statement has a superb string matching feature that uses wild-cards.
- case statement uses
  - → filename matching meta-characters * and ?
  - → string matching character.
- Example: A script to test a user response for YES, yes, Yes, yEs (or no, NO, No, nO).

```
#! /bin/sh
echo "Do you wish to continue? [y/n]:"
read ans
case "$ans" in
        [Yy] [eE]* ) ;;                    # Matches YES, yes, Yes, yEs, etc
        [Nn] [oO]  ) exit ;;               # Matches no, NO, No, nO
                * ) echo "Invalid Response"
esac
```

4.11 expr: Evaluate an Expression

- expr command can be used to
  → evaluate an expression and
  → output the corresponding value.
- This command combines the following two functions:
  1) Performs arithmetic operations on integers and
  2) Manipulates strings.

1) Numeric Computation

- Five operators used on integers: +, -, *, / and %.
- Syntax:
  expr $op1 operator $op2
- Example:

```
$ x=5 y=3
$ expr  $x + $y              // outputs 8
$ expr  $x - $y              // outputs 2
$ expr $x \* $y          // * must be escaped to prevent shell from interpreting * as wildcard
                             //outputs 15
$ expr  $x / $y              //outputs 1
$ expr  $x % $y              // outputs 2
$ z =`expr $x + $y`          // command substitution to assign a variable
$ echo $z                    // outputs 8
```

2) String Handling

- Three functions used on strings:
  i) Finding length of string
  ii) Extracting substring
  iii) Locating position of a character in a string
- Syntax:
  expr "exp1" : "exp2"
- On the left of the colon (:), the string to be worked upon is placed.
  On the right of the colon(:), a regular expression is placed.

i) Length of the String

- The regular expression ".*" is used to print the number of characters matching the pattern.
- Syntax:
  expr "string" : ".*"
- Example:
  $ expr "vtunotesbysri" : '.*' // outputs 13

ii) Extracting a Substring

- expr command can be used to extract a string enclosed by the escape characters "\(" and "\)".
- Syntax:
  expr "string" : "\(substring \)"
- Example:
  $ expr "vtunotesbysri" : " \(sri \)" // outputs 'sri'

iii) Locating Position of a Character

- expr command can be used to find the location of the first occurrence of a character inside a string.
- Syntax:
  expr "string" : "[^ch]*ch" //ch → character
- Example:
  $ expr "vtunotesbysri" : "[^u]*u" // outputs 3

4.12 while Statement

• while loop can be used to execute a set of statements repeatedly as long as a given condition is true.
• Syntax:

```
while condition is true
do
        execute statements
done
```

• The statements enclosed between do and done are executed repeatedly as long as condition is true.
• Example: A script to display a message 3 times using while loop.

```
#! /bin/sh
num=1
while [$num -le 3]
        do
                echo " Welcome to Shell Programming "
                expr $num = $num +1;
        done
```

```
Output:
Welcome to Shell Programming
Welcome to Shell Programming
Welcome to Shell Programming
```

4.13 for Statement

- for loop can be used to iterate over all items(or strings) within a list.
- Syntax:

```
for variable in list
        do
                statements
        done
```

- Here, list consists of a set of items(or strings).
- Each item of the list is picked up and assigned to the "variable".
- The iteration continues until all items are picked from the array.
- Example: A script to display elements of an array.

```
#! /bin/sh
print("Here are the numbers in the list: \n");
for var in 10 20 30 40 50 60;
        do
                echo "$var  \t"
        done
```

```
Output:
Here are the numbers in the list
10 20 30 40 50 60
```

4.13.1 Possible Sources of List

- Possible sources of list are
  1) List from variables
  2) List from command substitution
  3) List from wildcards and
  4) List from positional parameters

1) List from Variables

- Example: A script to evaluate & display a set of variables using for-loop.

```
#! /bin/sh
x="Dream"
y="Believe "
z="Achieve"
for var in $x $y $z;
        do
                echo "$var  \t"
        done
```

```
Output:
Dream        Believe        Achieve
```

2) List from Command Substitution

- Command substitution can be used for creating a list.
- Useful: when list is large.
- Example: A script to display current date using for-loop.

```
#! /bin/sh
for var in `date`
        do
                echo "$var  \t"
        done
```

```
Output:
Mon    Nov    4       08:02:45      IST     2017
```

3) List from Wildcards

- The shell can use wildcards for matching filenames.
- Example: A script to print all files with pdf extension.

```
#! /bin/sh
for file in *.pdf
do
        echo "Printing $file \n"
        lp $file
done
```

```
Output:
Printing chap1.pdf
Printing chap2.pdf
Printing chap3.pdf
```

4) List from Positional Parameters

- Example: A script (program4.sh) to read & display a positional parameters using for-loop.

```
#! /bin/sh
for var in "$*"                                  # even "$@" can be used
        do
                echo "$var \t"
        done
```

```
Run:
$ program4.sh A B C
Output:
A       B       C
```

4.14 set and shift Commands and Handling Positional Parameters

4.14.1 set

- set command can be used to assign positional parameters ($1, $2 and $3) to command line arguments.
- This command can be used for picking up individual fields from the output of a program.
- Example:

```
$ set 98 23 62
```

- Here, above line assigns
  - → 98 to $1
  - → 23 to $2
  - → 62 to $3.
- This command can also be used to assign the other parameters $# and $*.
- Example:

```
$ set `date`
$ echo $*
Mon Nov 4 08:02:45 IST 2017
```

- Example:

```
$ echo "The date today is $2 $3, $6"
The date today is Nov 4, 2017
```

4.14.2 shift

- shift command is a shell built-in that operates on the positional parameters.
- Each time shift command is called, it shifts/transfers all the positional parameters down by one.
- For example: $2 becomes $1
  $3 becomes $2
  $4 becomes $3, and so on.
- Example:

```
$ echo "$@"                          # $@ and $* are interchangeable
Mon Nov 4 08:02:45 IST 2017
$ echo $1 $2 $3
Mon Nov 4

$ shift                              # shifts 1 place
$ echo $1 $2 $3
Nov 4 08:02:45

$shift 2                             # shift 2 places
$echo $1 $2 $3
08:02:45 IST 2017
```

4.14.3 Set -- : Helps Command Substitution

- Problem with set command:
  When set command is used with command substitution, the output of the command may begin with a -(hypen). In this case, set command interprets -(hypen) as an option and does not work correctly.
- For example:

```
$set 'ls -l student,lst'
-rwxr-xr--: bad option
```

- Solution: Use --(double hypen) immediately after set command.

```
$set --  'ls -l student.lst'
 -rwxr-xr-- 2  kumar  group  163   Jul     13      21:36  student.lst
```

4.15 here (<<) document

• The << symbol can be used to read data from the same file containing the script. This file is called as a here document.
• The term 'here' signifies that the data is here rather than in the file.
• Any command using standard input can also take input from a here document.
• Syntax:

```
command << delimiter
document
delimiter
```

• For example:

```
$ mailx kumar << MARK
        Explore
        Dream
        Discover
MARK
```

• The string (MARK) is delimiter.
• The shell treats every line delimited by MARK as input to the command mailx.
• kumar at the other end will see 3 lines of message text with the date inserted by command.
• The word MARK itself doesn't show up.

Using Here Document with Interactive Programs:

• A shell script can be made to work non-interactively by supplying inputs through here document.
• For example:

```
$ wc -l << END
        Decide
        Commit
        Succeed
END
3                       //outputs number of lines = 3
```

4.16 trap

• trap is a signal handler.
• Whenever the interrupt key (Ctrl+C) is pressed, a signal SIGINT is sent to terminate the shell script.
• However, it is not a good practice. For instance, the user may end up leaving a lot of temporary files on the disk.
• trap command can be used to perform clean up operation when a script receives a terminate signal.
• This command is normally placed at the beginning of the shell script.
• Syntax:

```
trap command_list signal_list
```

• The signal_list contains the signal names (SIGINT, SIGTERM, SIGQUIT).
• The command_list contains the commands to be executed when the signals are received by the script.
• Two common uses of trap:
    1) Clean up temporary files and
    2) Ignore signals

1) Cleaning up Temporary Files

• The user can remove some files and then exit if someone tries to abort the script from the terminal.
• Example:

```
$ trap 'rm temp.txt ; exit'  SIGINT
```

• Here, a file temp.txt will be automatically removed if a signal SIGINT is received by the script.

2) Ignoring Signals

• A script can be made to ignore a specific signal by using a null command list.
• Example:

```
trap ' ' SIGINT
```

• Here, the script ignores a signal SIGINT when it is received.

4.17 Simple Shell Program Examples

1) A shell script to accept a filename as argument and displays the last modification time if the file exists and a suitable message if it does not.

```
#!/bin/bash
echo "Enter name of the file: \c"
read filename
if [ -e $filename ]
then
        echo 'Last modification time is: \c'
        echo `ls -l $filename | cut -d " " -f 6,7,8`
else
        echo "file does not exist"
fi
```

```
Output:
Enter name of the file: student.lst
Last modification time is: Nov 04 12:04:11
```

2) A shell script to accept 2 file names & check if the permission for these files are identical and if they are not identical, display each filename followed by permission.

```
#!/bin/bash
echo "Enter 2 filenames: \c"
read f1 f2
file1 =`ls -l $f1 | cut -c 2-10`
file2 =`ls -l $f2 | cut -c 2-10`
if [ $file1 == $file2 ] then
        echo "Common file permission: $file1"
else
        echo "Different file permissions "
        echo " permission of $f1: $file1"
        echo " permission of $f2: $file2"
fi
```

```
Output:
Enter 2 filenames:  p1.c p2.c
Different file permissions
file permission for p1.c is rw-r--r--
file permission for p2.c is rwxr-xr-x
```

3) A shell script to print first 10 numbers (1 to 10)

```
#!/bin/sh
x=0
while [$x –le 10];
        do
                echo "$x  \t"
                x=`expr $x+1`
        done
```

```
Output:
1 2 3 4 5 6 7 8 9 10
```

4) A shell script (program4.sh) to accept any number of arguments and print them in a reverse order. For example if A B C are entered then output is C B A.

```
#!/bin/bash
n=$#
if [ $n -lt 2 ];
then
        echo "please enter 2 or more arguments" exit
else
        echo "The command line arguments in reverse order:"
        while [ $n -ne 0 ]
                do
                        eval echo "\$$n"      #display values in positional parameters $3 $2 $1
                        n = `expr $n - 1`
                done
fi
```

```
Run:
$ program4.sh A B C
Output:
C B A
```

5) A shell script to create a menu, which displays the list of files, process status, current date and current users of the system.

```
#! /bin/sh
echo " MENU  \n
        1. List of files          2. Processes of user  \n
        3. Today's Date           4. Users of system    \n
        5. Quit  \n
        Enter your option:  \c"
read choice
case "$choice" in
                1) ls –l;;
                2) ps –f ;;
                3) date ;;
                4) who ;;
                5) exit ;;
                *) echo "Invalid option"
esac
```

```
Output:
MENU
1. List of files          2. Processes of user
3. Today's Date           4. Users of system
5. Quit
Enter your option: 3
Mon Oct 8 08:02:45 IST 2007                    // date command executed
```

6) A shell script to read a string from terminal and display suitable message if it doesn't have at least 10 characters using expr.

```
#!/bin/sh
echo "Enter a string: \c"
read str
length = `expr "$str" : ".*" `
if [ $length -lt 10 ]
then
        echo "The string has less than 10 characters"
else
        echo "The string has $length characters"
fi
```

```
Output:
Enter a string: vtunotesbysri
The string has 13 characters
```

7) A shell script to read a string from terminal and display suitable message if it doesn't have at least 10 characters using case.

```
#!/bin/sh
echo "Enter a string: \c"
read str
length = (${#str}<10)
case $length in
        1) echo "The string has less than 10 characters" ''
        *) echo "The string has $length characters" ;;
esac
```

```
Output:
Enter a string: vtunotesbysri
The string has 13 characters
```

8) A shell script to check whether a given number is palindrome or not

```
#!/bin/sh
echo "Enter the number: \c"
read n
number=$n
reverse=0
while [ $n -gt 0 ]
        do
                a=`expr $n % 10 `
                n=`expr $n / 10 `
                reverse=`expr $reverse \* 10 + $a`
        done
echo $reverse
if [ $number -eq $reverse ]
then
        echo "Number is palindrome"
else
        echo "Number is not palindrome"
fi
```

```
Output:
Enter the number: 1221
Number is palindrome
```

9) A shell script to read a pattern and filename from the terminal. And search for the pattern in the file.

```
#! /bin/sh
echo "Enter the pattern to be searched: \c"
read pname
echo "Enter the file to be used: \c"
read fname
echo "Searching for pattern $pname from the file $fname"
grep $pname $fname
echo "Selected records shown above"
```

```
Output:
Enter the pattern to be searched : MH
Enter the file to be used: student.lst
Searching for pattern MH from the file student.lst
4       | MH   | 10    | IS    | 111
4       | MH   | 11    | CS    | 401
Selected records shown above
```

10) A shell script (program10.sh) to compute sum of numbers passed in command line

```
#!/bin/sh
sum=0
for I in "$@"
        do
                sum =`expr $sum + $I`
        done
echo "sum is $sum"
```

```
Run:
$ program10.sh 2 4 6
Output:
sum is 12
```

11) A shell script (program11.sh) to compute length of strings in the file (student.lst)

```
#!/bin/sh
sum=0
for I in `cat student.lst `;
        do
                echo "string is $I \n"
                x= `expr "$I":'.*'`
                echo "length is $x \n"
        done
```

```
Run:
$ cat student.lst
RAMA KRISHNA
$ program11.sh
Output:
string is RAMA
length is 4
string is KRISHNA
length is 7
```

12) A shell script to validate the password. Let VALID_PASSWORD="secret"

```
#!/bin/sh
echo "Please enter the password:"
read PASSWORD
if [ "$PASSWORD" == "$VALID_PASSWORD" ]; then
        echo " Login successful"
else
        echo "ACCESS DENIED!"
fi
```

```
Output:
Please enter the password: secret
Login successful
```

13) A shell shell script to append doc extension to all filenames.

```
#!/bin/sh
for file in ch1 ch2 ch3;
        do
                cp $file ${file}.doc
                echo $file copied to $file.doc
        done
```

```
Output:
ch1 copied to ch1.doc
ch2 copied to ch2.doc
ch3 copied to ch3.doc
```

14) A shell script to check if the length of the name is greater than 20 characters.

```
#!/bin/sh
echo "Enter your name: \c"
read name
if [`expr "$name" : ".*" `-gt  20] ; then
        echo "Name is very long"
else
        echo "You can proceed!"
fi
```

```
Output:
Enter your name: Rama Krishna
You can proceed!
```

MODULE 4(CONT.): MORE FILE ATTRIBUTES

4.18 File inodes and the inode Structure

- Every file is associated with a table called the inode table (index node).
- The inode table contains all attributes of a file except filename and file content.
- This table can be accessed by the inode number.
- The inode table contains the following attributes of a file.
    1) File type (regular, directory or symbolic)
    2) File permission (rwx)
    3) Number of links (pointers)
    4) The UID of the owner
    5) The GID of the group owner
    6) File size in bytes
    7) Date and time of last modification of the file
    8) Date and time of last change of the inode
    9) An array of pointers that keep track of all disk blocks used by the file.
- When a new files is created, following 2 information are entered in the directory file:
    1) Filename and
    2) inode number.
- How a command gets executed (say ls -l emp.sh) ?
    1) Firstly, the kernel locates the inode number of the file from the directory.
    2) Then, the kernel reads the inode table to fetch data relevant to the file.
- The file system has a separate memory for storing inode tables in a contiguous manner.
- This memory is accessible only to the kernel.
- The inode number is actually the position of the inode table in this memory.

Displaying inode number of a file (-i)

- –i (inode) option can be used to know the inode number of a file.
- Example:

```
$ ls –il emp.sh
9059 -rw-r--r-- 1 kumar metal 51813 Jan 31 11:15 emp.sh
```

- In the first column, 9059 is the inode number of file named emp.sh.

4.19 File Links – Hard and Soft links

• Consider the following Example.

• Example:

```
$ ls -l backup.sh restore.sh
-rwxr-xr-- 2 kumar group 163 Jul 13 21:36 backup.sh
-rwxr-xr-- 2 kumar group 163 Jul 13 21:36 restore.sh
```

• In the second column, 2 is the link count of both files (backup.sh and restore.sh)

• Since all attributes are identical, the 2 files may be duplicate copies of each other.

Comparing inode numbers of 2 files (-i) :

• –i (inode) option can also be used to check if 2 files are linked to each other.

• Example:

```
$ ls -li backup.sh restore.sh
274 -rwxr-xr-- 2 kumar group 163 jul 13 21:36 backup.sh
274 -rwxr-xr-- 2 kumar group 163 jul 13 21:36 restore.sh
```

• In the first column, 274 is the inode number of both files.

• Since both files have same inode number(274), there's actually one file with a single copy on disk.

• Same inode number indicates that both filenames are linked to the same physical file directly.

4.19.1 ln: Creating Hard Links

• This command can be used to create a hard link between 2 or more files.

• Syntax:

```
ln filename1 filename2
```

• Example:

```
ln emp.lst employee              // This command links emp.lst with employee
ls -li emp.lst employee          // To check if 2 files have the same inode number
518 -rwxr-xr-x 2 kumar group 915 may 4 09:58 emp.lst
518 -rwxr-xr-x 2 kumar group 915 may 4 09:58 employee
```

• In the third column, 2 is the link count of 2 files (emp.lst and employee).

• Example:

```
ln employee emp.dat              // This command links employee with emp.dat
ls -li emp*                      // To check if 3 files have the same inode number
518 -rwxr-xr-x 3 kumar group 915 may 4 09:58 emp.dat
518 -rwxr-xr-x 3 kumar group 915 may 4 09:58 emp.lst
518 -rwxr-xr-x 3 kumar group 915 may 4 09:58 employee
```

• In the third column, 3 is the link count of 3 files (emp.lst employee and emp.dat).

• Multiple files can be linked together. Here, the destination filename must be a directory.

• Example:

```
ln chap1 chap2 chap3 content     // "content" is a directory
```

• A file is considered to be completely removed from the file system when its link count drops to 0.

Where to use Hard Links? (Applications of Hard Link)

1) ln data/ foo.txt input_files

• The above command creates link in directory input_files.

• With this link available, your existing programs will continue to find foo.txt in the input_files directory.

• It is more convenient to do this that modifies all programs to point to the new path.

2) Links provide some protection against accidental deletion, especially when they exist in different directories.

3) Because of links, the user need not maintain 2 programs as 2 separate disk files if there is very little difference between them.

• A file's name is available to a C program and to a shell script. (emp.c emp.sh).

• A single file with 2 links can behave in 2 different ways based on the name by which it is called.

Disadvantage of Hard Links

1) Directories cannot be linked.

2) Files across 2 different file system cannot be linked.

For example: Cannot link a file in the /usr file system to another file in the /home file system.

4.19.2 Soft Links

- Soft links are also known as symbolic links.
- Symbolic links are files that hold the pathname of the original file.
- These links doesn't hold the file's content. Rather, they hold the pathname of the file that actually has the content.

ln -s: Creating Soft Links

- This command can be used to create a soft link between 2 or more files.
- Syntax:

  ln -s filename1 filename2
- Example:

```
ln -s note note.sym                    // This command links note with note.sym
ls -li note note.sym
9948 -rw-r--r--      1 kumar group 80 feb 16 14:52 note
9952 lrwxrwxrwx      1 kumar group 04 feb 16 15:07 note.sym -> note
```

- In the second row,

  "l" indicate symbolic link

  -> indicates note.sym contains the pathname for the note

  4 indicates size of symbolic link
- Since 2 files have the different inode numbers, they are not identical.
- If note is deleted, its data will be lost. In this case, note.sym will point to a nonexistent file and becomes a dangling symbolic link (dangling pointer).
- Like other files, a symbolic link has a separate directory entry with its own inode number.

Advantage of Soft Links

1) Directories can be linked.

   For example: To link all filenames in a directory to another directory, simply link the directories.

2) Files across 2 different file system can be linked.

Soft link	Hard link
ln -s	ln
Can create soft link for both files and directories.	Files only.
Symbolic link points the link to the file or directory name.	Hard link is the reference or pointer to the exact file.
The link will not be accessible if the original file or folder is deleted/removed.	Can access.
Different inode value.	Same inode number.
Soft link is possible in different partition.	Not possible.

MODULE 4(CONT.): SIMPLE FILTERS

4.20 Filters

• You can connect two commands together so that the output from one program becomes the input of the next program.
• Two or more commands connected in this way form a pipe.
• To make a pipe, put a vertical bar (|) on the command line between two commands.
• When a program takes its input from another program, it performs some operation on that input, and writes the result to the standard output. It is referred to as a filter.
• For example:

```
$ ls -l | grep "Aug"
-rw-rw-rw-   1 john  doc     11008 Aug  6 14:10 ch02
-rw-rw-rw-   1 john  doc      8515  Aug  6 15:30 ch07
```

• Here, grep is used in a pipe so that only those lines of the input files containing a given string are sent to the standard output.
• To understand the working of different commands (like head, tail, cut, paste & sort), let us consider we have following information in file student.lst.

```
$ cat student.lst
4       | MH   | 10    | IS    | 111
4       | MH   | 11    | CS    | 401
4       | GW   | 11    | CS    | 402
4       | VV   | 11    | CS    | 403
3       | NA   | 10    | IS    | 058
3       | BR   | 11    | CS    | 404
1       | EW   | 11    | CS    | 405
1       | AM   | 10    | IS    | 021
1       | RN   | 10    | IS    | 054
1       | DS   | 11    | CS    | 406
```

4.21 head

• This command is used to display the top of the file.
• By default, it displays the first 10 lines of the file.
• It is mainly useful to verify the contents of a file.
• Syntax:

```
head [count option] filename
```

• Example:

```
head emp.lst  # displays first 10 lines of emp.lst
```

Displaying first n lines of a file (-n)

• -n option is used to specify a line count and to display the first n lines of the file.
• Example:

```
$ head -n 3 student.lst                    // displays first 3 lines of student.lst
4       | MH   | 10    | IS    | 111
4       | MH   | 11    | CS    | 401
4       | GW   | 11    | CS    | 402
```

Invoke vi editor with the Last Modified File (-t)

• -t option is used to display filenames in order of their modification time.
• Example:

```
vi `ls –t | head –n 1`          # Opens last modified file for editing
```

• Here, the first filename is received from the list and the filename is used as an argument to vi.

4.22 tail

- This command is used to display the end of the file.
- By default, it displays the last 10 lines of the file.
- Syntax:

  tail [+- count] filename[s]
- Here, count can be one of the two:

  1) -count option specifies a line count i.e. number of lines to be selected.

  2) +count option specifies starting line number i.e. The line number from where the selection should begin.
- Example:

```
$ tail student.lst              // Displays last 10 lines of student.lst

$ tail -n 3 student.lst         // Displays the last 3 lines of student.lst
1       | AM   | 10   | IS   | 021
1       | RN   | 10   | IS   | 054
1       | DS   | 11   | CS   | 406

$ tail +9 student.lst           // Displays all lines, beginning from line# 9 to the end of the file.
1       | RN   | 10   | IS   | 054
1       | DS   | 11   | CS   | 406
```

tail options

Monitoring the file growth (-f):

- -f option can be used to measure the length of a file when running program continuously writes to the file.
- Example:

  tail –f /oracle/install.log

Extracting bytes rather than lines (-c)

- -c option is used to display(or extract) bytes rather than lines.
- Extraction is performed relative to the beginning or end of file.
- Example:

```
tail -c -8 work.c          // copies last 8 bytes from work.c
tail -c +8 work.c          // copies all bytes after skipping 8 bytes
```

4.23 cut

- This command can be used to extract the file contents vertically.
- This command can be used to extract required fields or columns from a file.

4.23.1 cut Options

Cutting columns (-c)

- -c option can be used to extract specific column by specifying the column numbers.
- Range of column numbers can be specified using the hyphen(-).
- Comma (,) can be used as the column delimiter.
- Example:

```
$ cat student.lst
4       | MH   | 10    | IS    | 111
4       | MH   | 11    | CS    | 401
4       | GW   | 11    | CS    | 402
4       | VV   | 11    | CS    | 403

$ cut -c 5-6, 10-11 student.lst
MH      10
MH      11
GW      11
VV      11
```

- Example:

```
cut -c -3, 5-6,  15- student.lst        // 15- means column 15 to end of line, -3 is same as 1-3.
```

Cutting Fields (-f)

- -f option can be used to extract specific field from a file.
- The tab is used as the default field delimiter.
- However, different delimiter can be specified with following 2 options:
  1) -d for the field delimiter &
  2) -f for the field list.
- Example 3:

```
$ cut -d \| -f 2,5 student.lst          OR      cut –d "|" -f 2,5 student.lst
MH      111
MH      401
GW      402
VV      403
```

- The above command extracts the second and fifth fields of student.lst.
- Delimiter "|" has to be escaped (by \ or " ") to prevent the shell from interpreting as the pipeline character.

Extracting User List from who output:

- cut can be used to extract the first word of a line by specifying the space as the delimiter.
- Example 4:

```
$ who | cut -d " " -f 1                 // extracts list of users only
```

4.24 paste

- This command can be used to merge 2 files line by line.
- Syntax:

  paste cutlist1 cutlist2
- The tab is used as the default delimiter.
- However, different delimiter can be specified with –d option.
- Example:

```
$ cat file1
KA
PB
GJ

$ cat file2
KARNATAKA
PUNJAB
GUJARAT

$ paste file1 file2
KA KARNATAKA
PB PUNJAB
GJ GUJARAT

$ paste -d "|" file1 file2                    // delimiter "|" placed
KA      |       KARNATAKA
PB      |       PUNJAB
GJ      |       GUJARAT
```

Joining Lines (-s)

- -s option can be used to merge the files in sequentially.
- It reads all the lines from a single file and merges all these lines into a single line.
- Example:

```
$ paste -s file1 file2
KA PB GJ
KARNATAKA PUNJAB GUJARAT

$ paste -s -d "," file1 file2        //a delimiter "," can be used for sequential merging of files
KA, PB, GJ
KARNATAKA, PUNJAB, GUJARAT
```

4.25 sort

- Sorting is the ordering of data in ascending or descending sequence.
- This command orders a file.
- By default, it reorders lines in ASCII collating sequence i.e. whitespace first, then numbers, then uppercase letters and finally lowercase letters.
- It can identify fields and it can sort on specified fields.
- Syntax:

  sort filename
- Example:

  $ sort emp.lst
- This default sorting sequence can be altered by using certain options.

4.25.1 sort Options

Option	Description
-tchar	uses delimiter char to identify fields
-k n	sorts on nth field
-k m,n	starts sort on mth field and ends sort on nth field
-k m.n	starts sort on nth column of mth field
-u	removes repeated lines
-n	sorts numerically
-r	reverses sort order
-f	folds lowercase to equivalent uppercase
-m list	merges sorted files in list
-c	checks if file is sorted
-o flname	places output in file flname

Sorting Primary Key (-k)

- –k option can be used to sort on a specific field (primary key).
- Example:

  sort –t "|" –k 2 student.lst //primary key is second field which will be sorted

The Sort order in Reverse (-r)

- –r option can be used to sort in reverse order .
- Example:

  sort -t "|" -r -k 2 student.lst

Sorting on Secondary Key

- –k option can also be used to sort on more than one field. (primary key & secondary key).
- Example:

  sort –t "|" –k 5,5 –k 2,2 student.lst //primary key is third field, secondary key is second field
- Here, –k 2,2 indicates sorting starts on the second field and ends on the same field.

Sorting on Columns

- –k option can also be used to sort specific column position within a field.
- Example:

  sort –t "|" –k 5.1,5.2 student.lst //sort on & 2nd column positions within 5th field.
- General format: -k m.n where n is the character position in the mth field.

Numeric Sort (-n)

- –n option can be used to sort on numeric values rather than in ASCII collating sequence.
- Example:

  sort –n student.lst

Removing Repeated Lines (-u)

- –u option can be used to remove repeated (duplicate) lines from a file.
- Only one of the duplicate entries is retained in the output.
- Example:

  cut -d "|" -f 3 student.lst | sort -u | tee desig.txt

Specify Output Filename (-o)

• –o (output) option is used to specify the output filename.
• The input and output filenames can also be the same.
• Example:

```
sort –o student.lst  student.out          // output filename is student.out
sort –o student.lst  -k 1 student.out     // sort on first field &  output to file student.out
```

Check for Default Order(-c)

• -c (check) option is used to check whether the file has actually been sorted in the default order.
• Example:

```
sort –c student.lst
sort –t "|" –c –k 1 student.lst           // check if first field is already sorted
```

Merge Sorted Lists(–m)

• –m (merge) option can be used to merge 2 or more files that are sorted individually.
• Example:

```
sort –m file1 file2 file3
```

4.26 umask and Default File Permissions

• When user creates a file/directory, the permissions assigned to the file/directory depends on
→ system's default setting and
→ value of the variable umask.
• umask command can be used to set & display the user mask of file/directory.
• The value of user mask variable informs the system which permissions are to be denied, rather than granted.
• umask command can be used to change the permissions to required values during file/directory creation itself.
• The default permissions:
1) rw-rw-rw- (octal 666) for regular file
2) rwxrwxrwx (octal 777) for directory
• The default is transformed by subtracting the user mask from it to remove one or more permissions.
• Syntax:

```
umask [octal_value]
```

• Example:

```
$ umask              // displays the current user mask setting
022
```

• Example:

```
$ umask 042          // sets new umask value
```

• The default is transformed by subtracting the user mask from it to remove one or more permissions.
• Example:

For regular file:

```
 666 Default system value
-022 umask value
 644 Default user value    → 110 100 100  →  rw-r—r--
```

• When user mask is set, it is temporary only for that session. When logout the settings, revert back to the default.

4.27 Two Special Files: /dev/null and /dev/tty

/dev/null

• Sometimes, the user wants to execute a command, but does not want the output to be displayed on the screen.
• In such cases, the user can discard the output by redirecting it to the file /dev/null.
• Syntax:

```
command > /dev/null
```

• Here command is the name of the command you want to execute.
• The file /dev/null is a special file that automatically discards all its input.
• Size of this file is always zero.
• The content of this file is always emptied immediately after receiving data.
• Example:

```
$ cmp foo1 foo2 >/dev/null
$ cat /dev/null
$                                   // Size is always zero i.e. empty file
```

• Applications :
  1) Checking whether a shell program runs successfully without seeing its output on the screen.
  2) Redirecting unwanted outputs into this file.
  3) Redirecting error messages away from the terminal so that they don't appear on the screen.
• It is a pseudo-device; it's not associated with any physical devices.
• Whether the user redirects or appends output to this file, its size always zero.

/dev/tty

• This file indicates the user's terminal.
• It is a file under the device directory that belongs to the root.
• It represents the terminal assigned to each user.
• This file can be used to display the output on the monitor by copying/redirecting the output on to /dev/tty.
• Applications :
  1) Useful when a program needs to interact with a user even though its standard input/output are connected to files rather than the terminal.
  2) The user may monitor the output of the first command piped to the next command.
  ➢ Example:

```
ls -l | tee /dev/tty | wc -l
Total 3
dwrxr-xr-x   2    root   root   9582   jan 04      2009   file1
-wrxr-xr-x   1    root   root   9582   dec 25      2014   file3
-wrxr-xr-x   2    root   root   2864   feb 14      2005   file5
```

  ➢ Here, the use of tee and sending a copy of its input onto /dev/tty displays the output on the terminal apart from being forwarded to the second command.
  ➢ Example:

```
cut -c 1,5 shortlist | tee /dev/tty | sort
2233 |
9876 |
2365 |
```

MODULE 5: THE PROCESS THE PERL

MODULE 5: THE PROCESS

5.1 Meaning of a Process

- A process is a program in execution.
- The process is said to be born when the program starts execution.
- The process remains alive as long as the program is active.
- The process is said to be dead when the program completes execution.
- Usually, the name of the process is same as the name of the program being executed.
  For ex: A process named grep is created when grep command is being executed.
- The kernel is responsible for management of process.
- Kernel determines the priorities for processes so that multiple processes can share CPU resources.
- The processes have attributes. (just like files).
- The kernel uses process table to store the attributes of processes. (just like inode table).
- Two important attributes of a process:

1) The Process-Id (PID)

- PID is a unique integer used to identify each process uniquely.
- PID is allocated by the kernel when the process is born.
- PID can be used to control a process.

2) The Parent PID (PPID)

- The PID of the parent is available as a process attribute**.**

The Shell Process

- When a user logs into the system, the kernel creates a shell process.
- Any command entered by the user is actually the standard input to the shell process.
- When the user logs out of the system, the kernel kills the shell process.

5.2 Parent and Child Process

- A process that creates another process is called the parent process.
- A newly created process is called the child process.
- When user enters a command at the prompt($), the shell becomes the parent process, which in turn creates a child process.
- For example:

1) When the command cat emp.lst is run from the keyboard, the shell process creates a child process for running the cat program.

- The shell process remains active as long as the command is active.
- The shell is said to be the parent of cat, while cat is said to be the child of the shell.
- The ancestry of every process is ultimately traced to the first process (PID 0) that is set up when the system is booted.

2) When commands like "cat sample.lst | grep MH student.lst" is run from the keyboard, the shell process creates two child processes simultaneously:

1) one for running the cat program and
2) another for running the grep program.

- The shell is said to be the parent of cat & grep.

Wait or not Wait?

1) A parent may wait for the child to die so that the parent can create the next process.
- The death of the child is intimated to the parent by the kernel.
  Example: Shell process

2) A parent may not wait for the child to die and may continue to create other processes.
  Example: init process

5.3 Mechanism of Process Creation

1) Fork

- The parent process uses fork() to create a child process.
- After fork() executes, the parent & child have different PIDs and PPIDs.
- When fork is invoked, the kernel replicates address space of the parent process to the child process.
- When a process is forked, the child inherits following attributes of the parent.
    1) User name of the real and effective user (RUID and EUID).
    2) Real and effective group owner (RGID and EGID).
    3) The current directory from where the process was run.
    4) The file descriptors of all files opened by the parent process.
    5) Environment variables like HOME, PATH.
- '.' child runs in its own address space, changes made to these parameters don't affect the parent.
- Forking process is responsible for the multiplication of processes in the system.
- At this point, both parent & child continue execution at the statement following fork.

2) Exec

- Forking only creates a process but it doesn't execute a new program.
- The child process uses exec() to execute its own new program.
- This operation replaces the entire address space with that of the new program.
- The PID and PPID of the child remains unchanged.
- At the end of the execution, exit() is called to terminate the child.

3) Wait

- When the child process starts its execution, the parent process can do 2 things:
    1) Wait to gather the child's exit status.
    2) Continue execution without waiting for the child.
- In first case, the parent uses wait() to wait for the child to complete its task.
- Finally, parent picks up the exit status of the child and continues with other tasks.

5.3.1 Example

- When command ls is entered from the keyboard, shell performs following tasks:
    1) The shell calls fork(). This creates a new process to run the ls program.
    2) The ls process calls exec(). This replaces the shell program with the ls program and then starts executing the ls program.
    3) The ls program performs its task. In the meanwhile, the shell executes wait().
- The entire mechanism of process creation is pictorially shown below:

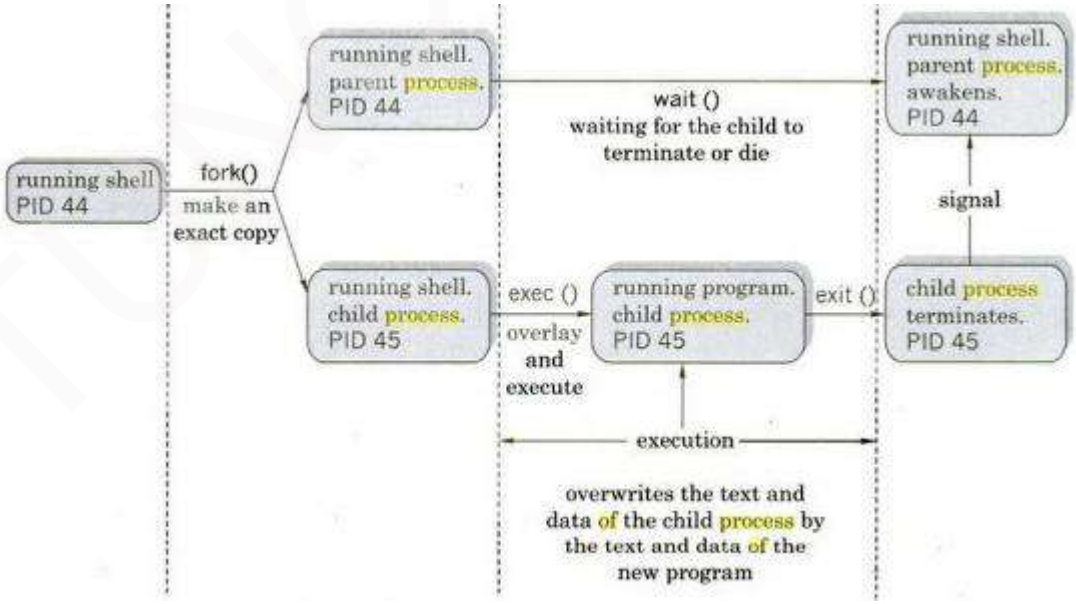
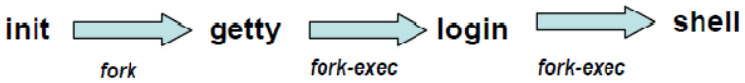


Figure 5.3.1: Mechanism of Process Creation

5.3.2 How the Shell is created?



• When the system goes to multiuser mode, init process calls fork to create a getty process for each active communication port (say pts/01 pts/02).
• Each getty process prints the login prompt on their terminal and then goes off to sleep.
• When a user tries to log in, getty process wakes up.
• getty process calls fork-exec which creates & executes login program to verify login name and password.
• On successful verification, login process calls fork-exec which creates & executes shell program.
• init process goes to sleep mode until termination of child processes.
• When the user logs out, it is intimated to init, which then wakes up.

5.4 ps

- This command is used to display the attributes of processes that are running currently.
- This command reads through the kernel's process tables to fetch the attributes of processes.
- By default, this command displays the process owned by the user running the command.
- If user execute the ps command immediately after logging in, will display the following details.
- Example:

```
$ ps
PID    TTY     TIME        CMD
4245   pts/7   00:00:00    bash
5314   pts/7   00:00:00    ps
```

- The header includes the following information:
  PID – Process Identification number TTY – terminal with which the process is associated
  Time - CPU time taken by the process CMD – Process name.

5.4.1 ps Options

Full Listing (-f)

- –f (full) option can be used to get a detailed list of attributes of all processes including parent processes.
- Example:

```
$ ps -f
UID    PID    PPID   C    STIME      TTY    TIME      COMMAND
root   14931  136    0    08:37:48   ttys   0 0:00    rlogind
sps    14932  14931  0    08:37:50   ttys   0 0:00    -sh
```

- The header includes the following information:
  UID – Login name of the user PID – Process ID PPID – Parent process ID
  C – An index of recent processor utilization, used by kernel for scheduling
  STIME – Starting time of the process in hours, minutes and seconds
  TTY – Terminal ID number
  TIME – Cumulative CPU time consumed by the process
  CMD – The name of the command being executed

Displaying Processes of a User (-u)

- –u (user) option can be used to know the activities of any user.
- Example:

```
$ ps -u kumar
```

Displaying All User Processes (-a)

- –a (all) option can be used to display all user processes. However, it doesn't display the system-processes.
- Example:

```
$ ps -a
PID    TTY          TIME         CMD
705    pts/05       00:00:00     sh
785    pts/09       00:00:03     vi
```

System Processes (-e)

- In addition to the user processes, a no. of system-processes keep running all the time in the system.
- Most system-processes are not associated with any controlling terminal (Ex: pts/01).
- System-processes are created
  → during system startup or
  → when the system goes into multiuser mode.
- System-processes are known as daemons because they are called w/o a specific request from a user.
- –e (every) option can be used to display all processes including both user & system processes.
- Example:

```
$ ps –e
PID    TTY          TIME         CMD
1      ?            41:55        init
234    pts/08       0:03         sh
```

- In the TTY column, ? indicates system processes which don't have a controlling terminal.

5.5 Background Processes

• When you start a process (run a command), there are two ways you can run it:

1) Foreground Processes

• By default, every process that you start runs in the foreground. It gets its input from the keyboard and sends its output to the screen.

2) Background Processes

• A background process runs without being connected to your keyboard.
• If the background process requires any keyboard input, it waits.
• The advantage of running a process in the background is that you can run other commands; you do not have to wait until it completes to start another!
• There can be only one job in the foreground, the remaining jobs have to run in the background.
• There are two ways of starting a job in the background:
    i) with the shell's & operator
    ii) nohup command.

5.5.1 & : No Logging out

• The & is the shell's operator used to run a process in the background.
• The user can direct the shell to execute the command in the background.
• In this case, the parent doesn't wait for the child's termination.
• Example:

```
$ sort –o emp.dat emp.dat &
[1]1413 The job's PID
```

• Here, the shell immediately returns a PID(1413) of the invoked command.
• The prompt is returned($).
• The shell is ready to accept the next command even though the previous command has not been terminated yet.
• The shell remains the parent of the background process.

5.5.2 nohup

• nohup statement can be prefixed to a command to permit execution of the process even after the user has logged out.
• Here also, the shell operator & must be used.
• Synyax:

```
nohup command-string [input-file] output-file &
```

• Example:

```
$ nohup sort emp.lst &
1252    Sending output to nohup.out
```

• Here, the shell immediately returns a PID(1252) of the invoked command.
• nohup sends the standard output(or error) of the command to the file nohup.out.
• When the user logs out, the child becomes an orphan and the init process becomes parent of orphan.
• In this case, the kernel reassigns the PPID of the orphan to the system's init process (PID 1).
• In this way, the user can terminate a parent (the shell) without terminating its child.

5.6 nice

• Processes are sequentially assigned resources for execution.
• The kernel assigns the CPU to a process for a duration of time (called time slice).
• When the time elapses, the process is placed in a queue.
• The execution of processes is scheduled depending on the priority assigned to the process.
• The nice command is used to control background process dispatch priority.
• The nice command is used with & operator to reduce the priority of processes.
• The main idea of nice: background processes must demand less attention from the system than interactive processes.
• nice values are system dependent and typically range from 1 to 19.
• A high nice value implies a lower priority.
• A program with a high nice number is friendly to other programs, other users and the system; it is not an important job.
• The lower the nice number, the more important a job is and the more resources it will take without sharing them.
• Example:

```
$ nice wc –l hugefile.txt      OR      $ nice wc –l hugefile.txt &
```

• The default nice value is set to 10.

Changing priority of Process (-n)

• The user can specify the nice value explicitly with –n (number) option
where number is an offset to the default.
• The priority is increased by that number up to a limit of 20.
• Example:

```
$ nice –n 5 wc –l hugefile.txt &      ;nice value increased by 5 unit
```

5.7 Signals

- Signals are software interrupts sent to a program to indicate that an important event has occurred.
- The events can vary from user requests to illegal memory access errors.

Signal Name	Signal Number	Description
SIGHUP	1	Hang up detected on controlling terminal or death of controlling process
SIGINT	2	Issued if the user sends an interrupt signal (Ctrl + C)
SIGQUIT	3	Issued if the user sends a quit signal (Ctrl + D)
SIGKILL	9	If a process gets this signal it must quit immediately
SIGTERM	15	Software termination signal (sent by kill by default)

Default Actions

- Every signal has a default action associated with it.
- The default action for a signal is the action that a script performs when it receives a signal.
- Some of the possible default actions are:
    1) Terminate the process
    2) Continue a stopped process
    3) Ignore the signal

Sending Signals

- There are several methods of delivering signals to a program or script.
- For example: When you press the Ctrl+C key, a SIGINT is sent to the script & default-action is script terminates.
- Following are some reasons for terminating a process:
    1) It's using too much CPU time.
    2) It's running too long without producing the expected output.
    3) It's producing too much output to the screen or to a disk file.
    4) It appears to have locked a terminal or some other session.
    5) It's using the wrong files for input/output because of an error.
    6) It's no longer useful.
- When a process terminates normally, the program returns its exit status to the parent.
- The exit status is a no. returned by the program providing the results of the program's execution.

5.8 kill Command

- Terminating a process prematurely is called killing.
- The kill command can be used to terminate a background process.
- <ctrl-c> can be used to terminate a foreground process.
- Syntax:

    kill PIDs OR kill –s NUMBER

- Example:

```
$ kill 123          // To kill a process whose PID is 123
$ kill 123 342 73   // To kill several processes whose PIDs are 123, 342, and 73
```

- Kill command uses SIGTERM (15) as default signal for terminating a process.
- The programs can send/receive more than 20 signals, each of which is represented by a number.

Killing the last background Job

- $! : stores the PID of the last background job.
- Example:

```
$ Sort -o emp.lst emp8.lst &
$ kill $!
```

Killing the login shell

- $$ --- Stores PID of the current shell.
- Example:

    $ kill -9 $$ OR kill -s KILL 0 (Kills all process including login shell)

Using kill with other Signals

- If the process ignores the signal SIGTERM, the user can terminate with SIGKILL signal (9).
- Example:

    $ kill -9 123 OR $ kill –s KILL 123

5.9 bg and fg Commands

- A job refers to a group of processes that are created by combining a set of commands(Eg: ls | wc).
- Examples of Job control:
  1) Move a job to the background (bg)
  2) Bring the job back to the foreground (fg)
  3) List the active jobs (jobs)
  4) Suspend a foreground job ([Ctrl-z])
  5) Kill a job (kill)

5.9.1 bg

- This command can be used to move currently running foreground process to run in background.
- Syntax:

  bg [PID...] OR bg %jobname
- Example:

```
$ bg          // moves current job to background
[2] + sh1.sh &
        + symbol indicates ----- > Current job
        - symbol indicates ------> Previous job
$ bg %0       // moves job with id 0 to background
$ bg %sort    // moves sort process to background
```

5.9.2 fg

- This command can be used to move currently running background process to run in foreground.

  fg %jobno OR fg %jobname
- Example:

```
$ fg          // brings most recent background process
$ fg %2       // brings 2nd background process to foreground
$ fg %sort    // brings sort process to foreground
```

5.9.3 jobs

- This command can be used to list out all the current jobs running in the system.
- Example:

```
$ sort employee.dat > sortedlist.dat &
[2] 530
$ grep 'director' emp.dat &
[3] 540
$ jobs
[3] + Running grep 'director' emp.dat &
[2] – Running sort employee.dat > sortedlist.dat &
[1] Suspended wc –l hugefile.txt
```

5.10 Executing a Command at a Specified Point of Time: at Command

- UNIX provides facilities to schedule a job to run at a specified time of day.
- The user can schedule least important jobs at a time when the system load is low.
- Job scheduling can be done using 2 commands: 1) at and 2) batch.

5.10.1 at: One-Time Execution

- This command
  - → takes the time the job is to be executed as its argument and
  - → displays the at> prompt.
- Input has to be given from the standard input (keyboard).
- Syntax:
  at [hh:mm] [am/pm] [now/noon/midnight/today/tomorrow]
- Example:

```
$ at 12:00
at > emp.sh
[ctrl-d]
Commands will be executed using /usr/bin/sh
Job 1 at Sat Nov 5 12:00:00: 2016
```

- Here, the job goes to the queue known as "at queue".
  At 12:00 a.m, the script file emp.sh will be executed.
  The output shows job number, the date and time of scheduled execution.
- We can also use the –f option to take the input from the file.
- Example:

```
$ at -f scriptfile 7 am Monday
$ at 10:23 Friday
at > lp /usr/sales/reports/*
at > echo "Files printed,!" | mail -s "Job done" boss
[Ctrl-d]
```

- The above job
  - → prints all files in the directory /usr/sales/reports and
  - → notifies the user named boss that the job is done.
- Example:

```
$ at 1 pm today
at > echo "Lunch with Director at 1 PM" >/dev/term/43
```

- The above job displays a message on user screen (/dev/term/43) at 1:00 pm.
- Example:

```
at midnight
at -l                   //Lists scheduled jobs
at –r                   //Remove the scheduled job
```

5.10.2 batch: Execute in Batch Queue

- This command allows the operating system to decide an appropriate time to run a process.
- This command can be used to schedule jobs for later execution.
- The jobs are executed as soon as the system load permits.
- This command uses an internal algorithm to determine the execution time.
- Example:

```
$ batch < emp.sh
Commands will be executed using /usr/bin/bash
Job 34 at sun dec 8 12:34:12 2008
```

- Example:

```
$ batch
sort /usr/sales/reports/* | lp
echo "Files printed, Boss!" | mailx -s "Job done" boss
```

- The above job
  - → sorts a collection of files
  - → prints the results and notifies the user named boss that the job is done.

5.11 cron : Executing a Command Periodically

- cron program is a daemon which is responsible for running scheduled tasks periodically.
- It can be useful for system admin to schedule administration tasks such as
  1) Backup and
  2) System logfile maintenance.
- It can also be useful for ordinary users to schedule regular tasks such as
  1) Calendar reminders and
  2) Report generation.
- Difference between at/batch and cron (or chronograph):
  1) at/batch schedule commands on a one-time basis.
  2) cron schedules commands on a regular basis.
- crontab file is used to specify the date and time of scheduled execution (command).
- Times can be specified in terms of minutes, hours, days of the month, months of the year, or days of the week.
- cron is listed in a shell script as one of the commands to run during a system boot-up sequence.
- Individual users don't have permission to run cron directly.
- cron "wakes up" every minute to see if there are commands to run.
- cron "goes to sleep" if there are no commands to run.
- cron looks for commands to run in a control file in /var/spool/cron/crontabs
- A typical entry in the crontab file will have the following format.

	minute	hour	day-of-month	month-of-year	day-of-week	command
Range	0-59	0-23	1-31	1-12	0-6	
Example	31	17	2	2	0	/backup

5.11.1 crontab: Creating a crontab File

- This command is used to place the control file in the directory containing crontab files.

```
$ crontab cron.txt          ;cron.txt contains cron commands
```

- crontab places the control file in /var/spool/cron/crontabs directory.
- –l option is used to see the contents of the control file.
- –r option is used to remove the control file.

5.12 find

- This command
  → recursively examines a directory tree to look for files matching some criteria and
  → then takes some action on the selected files.
- Syntax:
  find path_list selecton_criteria action
- Here,
  path_list consists of one or more directories to be examined
  selection-criteria contains conditions used for matching a file
  action specifies some action to be taken on the selected files
- Example:
  $find / -name a.out -print // locates all files named a.out
  /home/kumar/a.out
  /home/user/a.out
- Here,
  1) path_list **/** indicates that the search should start from root directory.
  2) Then, each file in the list is matched against the selection criteria.
  The selection criteria always consists of an expression in the form –operator argument.
  If the expression matches the file, the file is selected.
  3) Finally, a display selected files on the terminal.
- The group of filenames with a wild-card pattern can be matched.
  Here, the pattern should be quoted to prevent the shell from looking at it:
  find . -name "*.c" –print // All files with extension .c
  find . -name '[A-Z]*' –print // All files begin with an upper case letter

Selection Criteria	**Selects File**
-inum n	Having inode number n
-type x	If of type x, f ordinary file, d directory , l symbolic link
-type f	If an ordinary file
-perm nnn	If octal permissions match nnn completely
-links n	If having n links
-user usname	If owned by usname
-group gname	If owned by group gname
-size +x	If size is greater than x blocks
-mtime -x	If modified in less than x days
-newer flname	If modified after flname
-mmin -x	If modified in less than x minutes
-atime +x	If accesses in more than x days
-name flname	flname
-iname flname	As above, but match is case insensitive
-prune	Don't descend directory
-mount	But don't look in other file system

Action	**Significance**
-print	Prints selected file on standard output
-ls	Executes ls –lids command on selected files
-exec cmd	Executes UNIX command cmd followed by {} \;
-ok	Same as –exec except it asks for yes or no confirmation of the user

5.12.1 Selection Criteria

Locating a File by inode Number (-inum)

• -inum option can be used to search the files based on inode number.
• Example:

```
find / -inum 13975 -print 2> /dev/null
/usr/bin/gzip
/usr/bin/gunzip
```

File Type and Permission (-type and -perm)

• –type option can be used to search the files based its type.
f indicates ordinary file
d indicates directory file
l indicates symbolic file
• Example:

```
find /TC/BIN -type d -print 2>/dev/null      //shows current and hidden directories
./c_programs
./cpp_programs
```

• –perm option can be used to indicate the permission of file to be matched.
• Example:
find /TC/BIN -perm 666 -type f -print 2> /dev/null
• Here, –perm 666 selects files having read and write permissions for all categories of users.

Finding Unused File (-mtime and -atime)

• The -mtime and -atime option can be used to search the files based on the modified and accessed time respectively.
• Example:

```
find . -mtime -2 -print                  //list all that have been modified since less last 2 days
find /home -atime +365 -print        // files that are not been accesses for more than a year
```

5.12.2 find Operators (!, -o , -a)

• There are 3 operators can be used with the find.
1) ! operator is used before an option to negate its meaning.
• Example:
find . ! -name "*.c" –print // Finds all files excluding .c file.
2) -o operator represents an OR condition.
• Example:
find /home \(-name "*.sh " -o -name "*.pl" \) -print //searches files with .sh or .pl extn
3) –a operator represents the AND condition.
• Example:
find $HOME –perm 777 –a –type –d -print //all directories providing all access rights to all

5.12.3 Options Available in the Action Component

Display (-print)

• -print option can be used to display pathnames of matching files on the standard output.

Displaying the listing (-ls)

• -ls option can be used to view the listing for the selected files.
• Example:

```
find . -type f -mtime +2 -mtime -5 -ls
4521 1 –rw-r—r--  11user user       234   Mar 23 11:12 ./c_pgms/prime.c
```

• Here, above command display a special listing of those regular files that are modified in more than two days and less than five days.

Taking Action on Selected Files (-exec and -ok)

• The –exec option can be used to a specific command.
• This takes the command to execute as its argument, followed by {} and finally /;
• Example:

```
find $HOME -type f -atime +365 -exec rm {} \; //Uses rm command to remove all
                                              //ordinary files unaccessed for more than a year.
```

MODULE 5(CONT.): PERL

5.13 Structure of a perl Script

• Consider a perl script to accept username and display greeting message.

```
#!/usr/bin/perl
print("Enter your name: \c");                    # display message
$name=<STDIN>;                                   # read from keyboard
Print("Hi $name, Welcome to Perl Programming \n");
```

```
Output:
Enter your name:  rama
Hi rama, Welcome to Perl Programming
```

• The # symbol indicates the comments.
• The shell ignores all the characters that follow the #. However, this does not apply to the first line.
• The first line
#!/usr/bin/perl indicates the path where the perl script is available.
• The variables are prefixed with $ symbols & they are also called scalars.
• $ symbol is prefixed for both 1) variable definition ($x=12) 2) variable evaluation ($z=$x+1).
• <STDIN> is a file handle representing standard input. (e.g. keyboard).
• print() can be used for displaying message as well as computed result.
• Like C, all statements end with a semicolon (;) except the comment lines.
• Perl permits the use of escape sequence like "\n". Moves the cursor to the beginning of the next line.
• .pl is used as an extension for perl scripts.
• Perl permits the use of all logical operators, relational operators, the conditional operators & others.
• The variables are neither declared nor initialized.

5.14 Running a perl Script

• There are two ways of running a perl script:

1) At the Command Line

• The statements on command line with the –e option can be executed.
• This method is useful for running short scripts.
• Syntax:
$perl -e <perl code>
• Here, -e option is used to run perl script sent in as program.
• Example: A perl statement to display a message.

```
$ perl -e "print Welcome to Perl Programming "
Welcome to Perl Programming
```

2) From a Text File

• The statements can be placed in .pl file and then executed.
• This method is useful for running long scripts.
• Example: A perl script (program14.pl) to display a message.

```
#!/usr/bin/perl
print ' Welcome to Perl Programming ';                    # display message
```

• The # symbol indicates the comments in the script.
• The shell ignores all the characters that follow the #. However, this does not apply to the first line.
• The first line
"#!/usr/bin/perl" indicates the path where the perl script is available.
• By default, script is not executable. So, the chmod command can be used to make the script executable.

```
Run:
$ chmod +x program14.pl                  // add executable permission
$ program14.pl                           // execute the script program14.pl
Welcome to Perl Programming
```

5.15 chop() and chomp() Functions

5.15.1 chop()

- chop() function can be used to remove or eliminate the last character (new line character).
- chop() returns the character removed or discarded.
- Example: A perl script to demonstrate the use of chop function.

```
#!/usr/bin/perl
print("enter your name:");
$name = <STDIN>;
chop($name);                                    # removes newline character
if($name ne " ")
{
        print ("$name, have a nice day\n");
}
else
{
        print ("you have not entered your name\n");
}
```

```
Output:
Enter your name: rama
rama, have a nice day
```

5.15.2 chomp()

- chop() function removes only a new line character, \n, if that appears as the last character.
  1) If more than one new line character exists at the end, then only one new line character is removed.
  2) If no new line character is present, then nothing is removed.
- chomp() function returns the number of characters removed.
  1) If return value is 0, then no new line character is removed.
  2) If return value is 1, then the new line characters are removed.
- Example: A perl script to demonstrate the use of chomp function

```
#!/usr/bin/perl
$string1 = "rama";
$retval = chomp( $string1 );
print " Choped String is : $string1 \n";
print " Number of characters removed : $retval \n";
$string1 = " rama \n\n";
$retval = chomp( $string1 );
print " Choped String is : $string1 \n";
print " Number of characters removed : $retval \n";
```

```
Output:
Choped String is : rama
Number of characters removed: 0
Choped String is : rama
Number of characters removed: 2
```

5.16 Variables and Operators

• Perl has three basic data types: 1) scalars 2) arrays of scalars, and 3) hashes of scalars, also known as associative arrays.

1) Scalar

• Scalars are simple variables.
• They are preceded by a dollar sign ($). Ex: $var
• A scalar is either a number, a string, or a reference.
• A reference is actually an address of a variable.

2) Arrays

• Arrays are ordered lists of scalars that you access with a numeric index, which starts with 0.
• They are preceded by an "at" sign (@). Ex: $array

3) Hashes

• Hashes are unordered sets of key/value pairs that you access using the keys as subscripts.
• They are preceded by a percent sign (%). Ex: %array

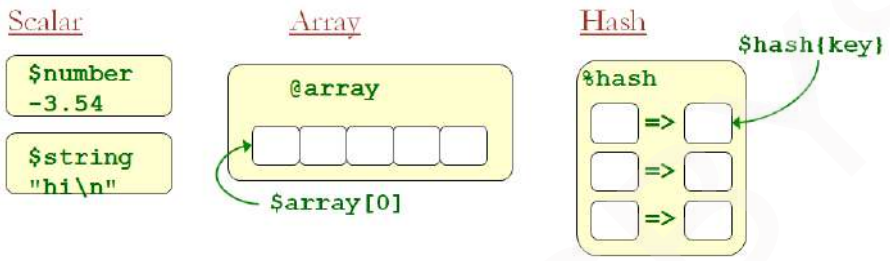


Figure 5.16: Basic data types

5.16.1 Variables

• The variables have no data type.
• The variables need not be initialization.
• The variables have to be prefixed with $ symbols (& they are also called scalars).
• $ symbol is used for both
  1) variable definition ex: $x=12
  2) variable evaluation ex: $z=$x+1
• Example:

```
$var=10;
print $var;                    //outputs 10
```

• Features of variables:

1) If a string is used for numeric computation or comparison, the string will be converted to a number.
2) An undefined variable is assumed to be a null string and a null string is numerically 0.

• Example:

```
perl –e '$x++; print ("$x\n");'        // returns 1 i.e. 0+1.
```

3) If the first character of a string is not number, the entire string becomes numerically equivalent to zero.
4) If a string is present in the middle of an expression, the string will be converted to an integer.

• Example:

"12k34" is converted to the integer 12, not 12k34.

5) To compare 2 strings, the ASCII value of each character starting from the left will be matched.

• Example:

x is greater than yy1234 because x is greater than y in the ASCII collating sequence.

6) For numeric comparison, operators like ==,!=,!=,>,<,>= and <= are used.
For string comparison, operators like eq, ne, gt, lt, ge and le are used.
7) A string can be unquoted or single/double quoted.
8) A ternary/conditional operator (?:) can be used.

• Example:

```
$max=($x>$y)? $x : $y;         // Sets max to x if x>y; otherwise sets max to y.
```

9) Every comparison/expression returns a value, and can set this value to a variable.

• Example:

```
$max=($x>$y)                   // Sets max to 1 if x>y; otherwise sets max to 0.
```

10) Some more examples:

```
$x =$y =$z =5;                          // Multiple assignment
$name =“rama \t\t krishna \n”;          // Two tabs and newline
$y =‘A’; $y++;                          // y becomes B
$z =“PO1”; $z++;                        // z becomes PO2
$todays_date = ‘date’                   // Uses Command substitution
$name = ‘rama”
$result =“\U$name\E”;                   // $result is RAMA
$result =“\u$name\E”                    // $result is Rama
```

• Example: A perl script to demonstrate the use of scalar variables.

```
#!/usr/bin/perl
$age = 25;              # An integer assignment
$name = "rama";         # A string
$salary = 25000.;       # A floating point
print "Age = $age \n";
print "Name = $name \n";
print "Salary = $salary \n";
```

```
Output:
Age = 25
Name = rama
Salary = 25000
```

5.16.2 Concatenation Operators . and Repetition Operator x

• Following 2 operators can be used to operate on strings:
  1) The . operator, which joins two strings together.
  2) The x operator, which repeats a string;.
• The . operator joins the second operand to the first operand.
• Example:

```
$a = “rama" . “krishna";                // $a is now ““ramakrishna"
$x =”perl”; $y=”.com”;  $z=$x . $y;     //  $z is now “perl.com”
```

• This join operation is also known as string concatenation.
• The x operator (the letter x) makes n copies of a string, where n is the value of the right operand:
• Example:

```
$a=“R"x3;                               //  $a is now RRR
$x=“@”x5;                               //  $x is now @@@@@
```

5.17 String Handling Functions

• Following are some string handling functions:

1) length(): determines the length of its argument.
2) index(s1, s2): determines the position of a string s2 within string s1.
3) substr(str,m,n): extracts a substring from a string str,
where m represents the starting point of extraction and
n indicates the number of characters to be extracted.
substr() can also extract characters from the right of the string, and insert or replace a string.
4) uc(str): converts all the letters of str into uppercase.
5) ucfirst(str): converts first letter of all leading words into uppercase.
6) reverse(str): reverses the characters contained in string str.

• A perl script to illustrate the use of string manipulation functions

```
#!/usr/bin/perl
$x =" abcdijklm";
print length($x);              # outputs 9
print index($x,j);             # outputs 5
substr($x,4,0) ="efgh";        # inserts "efgh" into the string $x from 4th position on the left
                               # 0 indicates non-replacement
print "$x";                    # outputs $x  as abcdefghijklm
$y = substr($x,-3,2);          # extracts 2 characters from the 3rd position on the right
print "$y"                     # outputs $y as kl
$name ="rama";
print uc($name);               #displays RAMA
print ucfirst($name);          # displays Rama
print reverse($name);          # displays amar
```

5.18 Default Variable: $_

• There are some variables which have a predefined and special meaning in Perl. These variables are referred to as special variables. (usual variable indicator are $, @, or %).
• A special variable $_ is called default variable.
• $_ contains the 1) default input 2) default argument and 3) pattern-searching string.

1) Default Input

• The line read from standard input will be assigned to default variable $_.
• Normally, $_ represents last line read.
• <>, chop and pattern matching operate on $_ by default.
• Example for <>:

```
while (<>)                              # reading line from a file $_= <STDIN>
{
        print;
}
```

• Here, <> is called NULL file handle which reads input from the keyboard (standard i/p).
• Example for chop():

chop(<STDIN>); will be equivalent to $var = <STDIN>; chop($var);

• Here, a line is read from standard input and assigned to default variable $_
• Then, the last character (in this case a \n) will be removed by the chop().

2) Default Argument

• Many functions use $_ as a default argument when no argument is mentioned explicitly.
• For ex: If no argument is specified for print function, it will print the value of default variable $_
• Example: A script to display the each element of list.

```
#!/usr/bin/perl
foreach $branch ('cs','ec','cv')
{
        print;                          # this is same as print $_
}
```

```
Output:
cs      ec      cv
```

5.19 $. : Representing the Current Line Number

• A special variable "$." is used to store the current line number.
• It is used to
→ represent a line address and
→ select lines from anywhere.
• Example:

```
perl –ne 'print if ($. < 4)' P1.dat                 // is similar to head –n 3 P1.dat
perl –ne 'print if ($. > 7 && $. < 11)' P1.dat      // display line 8 to 10
```

• Range operator(..) : The range operator is represented by two double dots ..
• Example:

```
perl –ne 'print if (1..3)' P1.dat                   // Prints lines 1 to 3 from P1.dat
perl –ne 'print if (8..10)' P1.dat                  // Prints lines 8 to 10 from P1.dat
```

• You can also use compound conditions for selecting multiple segments from a file.
• Example:

```
if ((1..2) || (13..15)) { print ;}                  // Prints lines 1 to 2 and 13 to 15
```

• Example: A script to print from 1 to 5

```
#!/usr/bin/perl
foreach $num in (1..5)
{
      print $num \t";                   # print from 1 to 5
}
```

```
Output:
1    2    3    4    5
```

5.20 Overview of Decision Making Structure – if else

• Three forms of if…else statement:

1) if statement 2) if...else... statement 3) if...elif...else... statement

5.20.1 if Statement

• This is basically a "one-way" decision statement.

• This is used when we have only one alternative.

• Syntax:

```
if(expression)
    {
        statement1;
    }
```

• Firstly, the expression is evaluated to true or false.

If the expression is evaluated to true, then statement1 is executed.

If the expression is evaluated to false, then statement1 is skipped.

5.20.2 if else Statement

• if construct is basically a "two-way" decision statement.

• This is used when we must choose between two alternatives.

• Syntax:

```
if(expression)
{
        statement1;
}
else
{
        statement2;
}
```

• Firstly, the expression is evaluated to true or false.

If the expression is evaluated to true, then statement1 is executed.

If the expression is evaluated to false, then statement2 is executed.

5.20.3 if elif else Statement

• Syntax :

```
if (expression)
{
                statement1;
}
elseif (expression)
{
        statement2;
}
else
{
        statement3;
}
```

• Example: A script to check whether an integer is positive or negative.

```
#! /bin/usr/perl
print ("Enter any non zero integer: \n");
$num = <STDIN>;
if ($num >= 0) then
        print ("Number is positive number");
else
        print ("Number is negative number");
```

```
Output:
Enter any non zero integer: 5
Number is positive number
```

5.21 Overview of loop control structures – while for foreach

5.21.1 while Loop

• A while loop construct can be used to execute a set of statements repeatedly as long as a given condition is true.

• Syntax:

```
while (expression)
{
        statement1
}
```

• Firstly, the expression is evaluated to true or false.

• If expression is evaluated to false, the control comes out of the loop w/o executing the body of loop.

• If the expression is evaluated to true, the body of the loop is executed.

• After executing the body of the loop, the expression is again evaluated to true or false. This cycle continues until expression becomes false.

• Example: A script to display a message 3 times using while loop

```
#! /bin/usr/perl
$i=1;
while(i<=3)
{
        print ("Welcome to Perl Programming \n");
        i=i+1;
}
```

```
Output:
Welcome to Perl Programming
Welcome to Perl Programming
Welcome to Perl Programming
```

5.21.2 for Loop

• A for loop construct can be used to execute s set of statements repeatedly as long as a given condition is true.

• Syntax:

```
for(expr1;expr2;expr3)
{
        statement1;
}
```

• Here, expr1 contains initialization statement expr2 contains limit test expression
expr3 contains updating expression

• Firstly, expr1 is evaluated. It is executed only once.

• Then, expr2 is evaluated to true or false.

• If expr2 is evaluated to false, the control comes out of the loop w/o executing the body of the loop.

• If expr2 is evaluated to true, the body of the loop (i.e. statement1) is executed.

• After executing the body of the loop, expr3 is evaluated.

• Then expr2 is again evaluated to true or false. This cycle continues until expression becomes false.

• A script to display a message 3 times using while loop

```
#! /bin/usr/perl
for($i =1; $i<=3; $i++)
{
        print ("Welcome to Perl Programming \n");
        i=i+1;
}
```

```
Output:
Welcome to Perl Programming
Welcome to Perl Programming
Welcome to Perl Programming
```

5.21.3 foreach Loop

- foreach construct can be used to iterate over all items within an array.
- Syntax:

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

- Here, list consists of a set of items.
- Each element of the array @arr is picked up and assigned to the variable $var.
- The iteration continues until all elements are picked from the array.
- Example: A perl script to display elements of an array .

```
#!/usr/bin/perl
@list = ("10", "20", "30", "40", "50", "60");
print("Here are the numbers in the list: \n");
foreach $temp (@list)
{
        print("$temp \t ");
}
```

```
Output:
Here are the numbers in the list:
10      20      30      40      50      60
```

5.22 Lists and Arrays

- A group of values is known as lists or arrays.
- The lists can be assigned to special variables known as array-variables.
- A list is a collection of scalar values enclosed in parentheses "(" and ")".
- Example:

  (1, 5.3, "hello", 2);
- Different ways of forming a list:

  1) Lists can contain scalar-variables.
  - Example:

    (17, $var, "a string")

  2) A list-element can also be an expression.
  - Example:

    (17, $var1 + $var2, 26 << 2)

  3) Scalar-variables can also be replaced in strings.
  - Example:

    (17, "the answer is $var1")

  4) A list can be created using the range operator:
  - Example:

    (1..10) same as (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

  5) The range operator can be used to define part of a list.
  - Example:

    (2, 5..7, 11) same as (2, 5, 6, 7 and 11)

5.22.1 Arrays

- An array is a variable that stores an ordered list of scalar values.
- Array variables are preceded by an "at" (@) sign.
- The arrays need not contain similar type of data.
- The arrays can dynamically grow or shrink at run time.
- Array-variables vs . Scalar-variables
  - @ is used to denote array-variables. $ is used to denote scalar-variables.
  - However, the same name can be used in an array-variable and in a scalar-variable.
  - Example:

    $var = 1;
    @var = (11, 27.1, "a string");
  - Here, $var and @var are two completely separate variables.

Array Creation

- Array variables are prefixed with the @ sign and are populated using either parentheses or the qw operator.
- Example:

  @array = (1, 2, 'Hello');
  @array = qw/This is an array/; same as @array = ("This", "is", "an", "array");

Accessing Array Elements

- To refer to a single element of an array, the dollar sign ($) is used.
- Syntax:

  $var[n]

  where n is the index value.
- The index begins with a 0 (zero).
- Example:

  print "$month[0] and $month[11]"; // prints jan and dec

Array Length

- Length of the array is determined by

  $length = @array;

Reading a File into an Array

```
@line = <>;                    // reads entire line from command line
print @line;                   // contains lines of the file
```

Last Index

- A special variable "$#" contains last index of an array.
- $# can be used to
  - → set the array to a specific size or
  - → delete all its element.
- Example:

```
$#month = 11;                    // array size is set to 12
$#month = -1;                    // no elements, elements are deleted
```

- Example: A perl script to illustrate array concept.

```
#!/usr/bin/perl
@days_betwn = ("wed", "thu");              # array with 2 elements
@days=(mon,tue, @days_betwn, fri);
@days[5,6]=qw/sat sun/;                    # qw function provides double quotes and comma
$length=@days;                             # length of the array is found
print ("The third day is $days[2] \n");
print("The days of week are @days \n");    # will print the contents of the array "days"
print("The number of elements in the array is $length \n");
print("The last subscript of the array IS $#days \n");
```

```
Output:
The third day is wed
The days of the week are mon, tue, wed, thu, fri, sat, sun
The number of elements in the array is 7
The last subscript of the array is 6
```

5.22.2 ARGV[]: Command Line Arguments

- The command line arguments are stored in the system array @ARGV[].
- The first argument is $ARGV[0].
  The second argument is $ARGV[1].
- Example: A perl script (program22.pl) to determine whether the given year is leap year or not, which takes year as the command line argument.

```
#!/usr/bin/perl
#leap_year.pl
if (@ARGV == 0)
{
        die("You have not entered the year \n");
}
$year = $ARGV[0] ;
$feb_days = ($year % 4 ==0) ? 29:28;
if ($feb_days eq "29")
        print("$year is certainly a leap year");
else
        print("$year is not a leap year");
```

```
Run 1:
$ program22.pl
Output 1:
You have not entered the year

Run 2:
$ program22.pl 2004
Output 2:
2004 is certainly a leap year
```

5.22.3 splice operator, push(), pop()

5.22.3.1 push() & pop()

- Array can be used to implement a stack or a queue.
- To use an array as stack, push() & pop() can be used.
  1) push() can be used for adding elements at the end of an array.
  2) pop() can be used for deleting elements at the end of an array.
- Example:

```
@list = (1,2,3,4,5);
push(@list,6);                  // now the list contents are 1 2 3 4 5 6
pop(@list);                     // now the list contents are 1 2 3 4 5
```

5.22.3.2 unshift() & shift()

- To use an array as queue, shift() & unshift() can be used.
  1) unshift() can be used for adding elements at the end of an array.
  2) shift() can be used for deleting elements at the beginning of an array.
- Example:

```
@list = (1,2,3,4,5);
unshift(@list,6);               // now the list contents are 1 2 3 4 5 6
shift(@list);                   // now the list contents are  2 3 4 5 6
```

- Example: A perl script to implement stack operations.

```
@array = (1,2,3,4,5);
print "Before pushing element @array \n";
push(@array, 6);
print "After pushing element @array \n";
pop(@array);
print "After popping element @array \n";
```

```
Output:
Before pushing element 1 2 3 4 5
After pushing element 1 2 3 4 5 6
After popping element 1 2 3 4 5
```

- Example: A perl script to implement queue operations.

```
@array = (1,2,3,4,5);
print "Before shifting element @array \n";
unshift (@array, 6);
print "After shifting element @array \n";
shift (@array);
print "After deleting element @array \n";
```

```
Output:
Before shifting element 1 2 3 4 5
After shifting element 1 2 3 4 5 6
After deleting element  2 3 4 5 6
```

5.22.3.3 splice() operator

- Splice operator can be used for adding or removing of elements at any locations of the array.
- Syntax:

```
splice(@array, $offset, [$length], [$list]);
        where @array : An array on which the splice works
              Offset : From where the insertion or removal begins
              $length : no. of elements to be removed. If it is 0, elements have to be added
              $list: Items to be i nserted are specified by $list
```

- Example:

```
@list = (1, 2, 3, 4, 5, 9);
splice(@list, 5, 0, 6..8);      // Adds from the 6th location,  list becomes 1 2 3 4 5 6 7 8 9
splice(@list, 0, 2);                    // Removes from beginning, list becomes 3 4 5 6 7 8 9
```

5.22.4 split()

- split function can be used to break up a line or expression into fields.
- After splitting, the fields can be assigned to
  → a set of variables ($var1, $var2, $var3) or
  → an array (@arr).
- Syntax:
  ($var1, $var2, $var3 …….) = split(/sep/ , str);
  @arr = split(/sep/ , str);
- Here, the above 2 lines split the string "str" on the pattern "sep"
  "sep" is a separator which can be a regular expression or a literal string
  "str" is a string on which split function works

Default Behaviour

1) When no string is specified explicitly, the split function works on the default variable $_
- Syntax:
  @arr = split(/sep/);

2) When no field separator is specified explicitly, white spaces are taken as the field separator by default.
- Syntax:
  @arr = split(str);

5.22.4.1 Splitting into Variables

- After splitting, the fields can be assigned to a set of variables ($var1, $var2, $var3).
- The field will be stored in a scalar-variable as a separate elements.
- Example: A perl script to split command line arguments.

```
#!/usr/bin/perl
print("enter 3 numbers: \n");
$numstring = <STDIN>;
if ($numstring eq " ");
        die("Nothing entered \n")
else
        ($f_num, $s_num, $t_num) = split( / /, $numstring);
print("the third, second and first numbers are \n");
print("$t_num, $s_num and $f_num ");
```

```
Output:
Enter 3 numbers: 100 200 300
The third, second and first numbers are 300, 200 and 100
```

5.22.4.2 Splitting into an Array

- After splitting, the fields can be assigned to an array (@arr).
- The field will be stored in a array-variable as a separate elements.
- Example: A perl script to split elements of an array.

```
#!/usr/bin/perl
$numstring = (100, 200, 300)
($var1, $var2, $var3) = split( /,/ ,  $numstring);
print("the third, second and first numbers are \n");
print("$t_num, $s_num and $f_num ");
```

```
Output:
The third, second and first numbers are 300, 200 and 100
```

5.22.5 join()

- join() function
  - → combines its arguments into a single string and
  - → uses the delimiter as the first argument.
- Syntax:

  join (sep, list);

  where "sep" is a separator which can be any string.
- Here, "sep" will be placed between the individual elements of the "list".

  "list" can be
  - → an array name
  - → a list of variables or
  - → string to be joined.
- Example:

```
$result = join(" ", "this", "is" , "an" , "example");      // $result = this is an example
$week = join( "|" , "mon", "tue", "wed");                  // $week = mon | tue | wed
```

- Example: A perl script to illustrate both split and join functions.

```
#!/usr/bin/perl
$x=join( ":" , 10,20,30,40);
print "$x \n";
@y=split(/:/ , $x);
print "@y \n";
$x=join("-", @y);
print "$x \n";
```

```
Output:
10:20:30:40
10 20 30 40
10-20-30-40
```

5.23 Associative Arrays – Keys and Value Functions

- Associative array is also called as hash.
- A hash is a set of key/value pairs. (key_n, value_n).
  1) First element is referred to as the key and
  2) Second element is referred to as the value of the key-element.

Initialization

- Syntax:
  %array = ("key1", value1, "key2", value2, "key3", value4);
  where %array is hash variable-name
  key1 key2 key3 are keys
  value1 value2 value3 are values
- Example:
  %month=("jan", 1, "mar", 3, "jun", 6); OR %month=("jan"=>1, "mar"=>3, "jun"=>6);
- When declaring the array, key & value is delimited by commas (or more friendly => notation).
- This uses string as index, so there is no need of ordering by index.
- If an array is assigned to a hash, then
  1) elements at even index becomes values and
  2) elements at odd index becomes the corresponding keys.

Accessing

- A key can be used to find the data element(or value) associated with it.
- Syntax:
  $array{"keyN"} retrieves valueN where $array is hash variable-name
- Example:
  $month{"mar"} will retrieve 3.

Adding

- New key-value pair can be added to an array by assigning a new value to a new key-element.
- Syntax:
  $array{"keyN"} = valueN
- Example:
  $month{"feb"} = 2;

Deleting

- A certain key-value pair can be deleted in an array by a key-element.
- Syntax:
  delete $array{"keyN"}.
- Example:
  deleting $month{"feb"} removes key-value pair (feb=>2).

5.23.1 Keys and Values Function

- keys() function can be used to return all the keys in the form of a list, which is stored in an array.
- Syntax:
  @list=keys(%array);
- values() function can be used to return all the values in the form of a list, which is stored in an array
- Syntax:
  @list=values(%array);
- Example: A perl script (program23.sh) to read the short names of week from command line and display their long names

```
#!/usr/bin/perl
%days=("mon", "monday", "tue", "tuesday", "wed", "wednesday");
foreach $name (@ARGV)
{       print ("The short name $name stands for $days{$name} \n");       }
$days{"thu"} = "thursday ";                    # adding new key value pair
print("\n");
print("-----------------------\n");
@short_list=keys(%days);
print("the short names are @short_list \n");
@long_list=values(%days);
print("the long names are @long_list \n");
```

Output:

```
$ program23.pl mon wed
The short name mon stands for monday
The short name wed stands for wednesday
-----------------------
the short names are mon tue wed thu
the long names are monday tuesday wednesday thursday
```

5.24 Regular Expressions – Simple and Multiple Search Patterns

- Regular expression is the pattern which contains metacharacters.
- Pattern matching is done by writing a Regular expression within a pair of forward slashes / and /.
- Syntax:

  /SEARCH_PATTERN/
- There are three regular expression operators
  1) Match Regular Expression - m//
  2) Substitute Regular Expression - s///
  3) Transliterate Regular Expression - tr///

5.24.1 Match Operator (m / /)

- The match operator can be used to check whether a variable contains a specified data (pattern) or not.
- Binding operator =~ can be used for pattern matching.
- Syntax:

  $variable =~ m/ SEARCH_PATTERN /;
- Here, SEARCH_PATTERN will be matched against the $variable.
- Two forward slashes(//) are used as delimiters of the SEARCH_PATTERN.
- Example:

  $var =~ m/MYSORE/; //find a match to MYSORE in $var
- By default, search is made on the current line or string.
- The default search behavior can be changed using following single character modifiers:

  g – Globally finds all matches.
  i – Makes the match case insensitive.
  o – Evaluates the expression only once.

5.24.2 Substitute Operator(s/ / /)

- The substitute operator can be used to

  → search for the pattern and
  → then replace/substitute the pattern with the replacement string.
- Syntax:

  s/ SEARCH_PATTERN / REPLACEMENT_PATTERN /;
- Here, the s prefix indicates that the pattern between the first / and the second / is to be replaced by the string between the second / and the third /.
- Example:

  $var =~ s/MYSORE/MYSURU/; // find MYSORE in $var and substitute with MYSURU

5.24.3 Translation Operator (tr/ / /)

- Translation operator can be used to translate characters from one form to another.
- Translation does not use regular expressions for its search on replacement values.
- Syntax:

  tr/ FIRST_PATTERN / SECOND_PATTERN /;
- Here, the translation replaces all occurrences of the characters in FIRST_PATTERN with the corresponding characters in SECOND_PATTERN.
- Example:

  $var =~ tr/A-Z/a-z/; // converts all appearance of uppercase to lowercase

5.24.4 Multiple Search Pattern (|)

- "|" operator can be used to search multiple patterns.
- Syntax:

  /SEARCH_PATTERN1 | SEARCH_PATTERN2 | SEARCH_PATTERN3 /;
- Example:

  /LCM | HCF/ //matches any string that contains subpatterns LCM or HCF
- Here, pipe character (|) behaves like a logical OR and hence helps selecting one of the many alternate patterns mentioned inside the search pattern.

5.24.5 More Complex Regular Expressions

Metacharacter	Meaning	Example
.(dot)	Matches exactly any one character in that position	/day./ matches with patterns like day1, days
*(asterisk)	Matches zero or any number of times the character in the preceding position	/ab*c/ matches like ac, abc, abbc...
+(plus)	Matches one or any number of times the character in the preceding position	/ab+c/ matches with abc, abbc .. [ac is not matchable]
? (Question mark)	Matches zero or one time the character in the preceding position	/ab?c/ matches with ac or abc
[abc] (Character class)	Matches any character in the set.	[abc]d matches with ad, bd or cd

Symbol	Significance
\w	Match word character [a-zA-Z0-9]
\d	Matches a digit [0-9]
\s	Matches white space character [\t\n]
\W	Doesn't match word character [^a-zA-Z0-9]
\D	Doesn't match matches a digit [^0-9]
\S	Doesn't match matches white space character [^\t\n]

5.25 File Handles and Handling File

5.25.1 File Handle

- File handle is a program variable that acts as a reference between
  1) perl program and
  2) Operating system's file structure.
- File handle names
  → do not begin with the special characters and digits
  → preferably uses uppercase letters.
- Streams are default file handles which are referred as STDIN, STDOUT and STDERR.
  1) STDIN connects to keyboard.
  2) STDOUT & STDERR connect to display screen.
- NULL file handle can be used to allow scripts to get input from
  → STDIN or
  → files listed on the command line.
- NULL file handle is written as <>, and is called diamond operator, angle operator or line-reading operator.
- Example:

```
perl -e 'while (<>)  { print ; }'              // input is taken from keyboard
perl -e 'while (<>)  { print ; }' test.txt'    // input is taken from file
```

5.25.2 File Handling Functions

open()

- open( ) function can be used to create a new file or to open an existing file.
- Syntax:
  open(FILEHANDLE, "access_mode filename");

access_mode	Meaning
<	Opens an existing text file for reading purpose
>	Opens a text file for writing, if it does not exist then a new file is created
>>	Opens a text file for writing in appending mode
+>	Opens a text file for reading purpose
+<	Opens a text file for writing purpose

- Example:

```
open(INFILE, "<file1.txt");        // Opens the file file1.txt in read only format.
open (OUTFILE, ">>file2.txt");     // Opens the file file2.txt in append mode.
```

close()

- close( ) function can be used to close an opened file.
- Syntax:
  close(FILEHANDLE);

die()

- die() function can be used to quit the script and display a message for the user to read.
- Syntax:
  die(LIST);
- The elements of LIST are printed to STDERR, and then the script will exit, setting the script's return value to $! (errno).

- Perl script to copy the first three lines of one file "file1.txt" into another file "file2.txt".

```
#!/usr/bin/perl
open(INFILE,"<file1.txt") || die("Cannot open file");      # Open the file1.txt in read mode
open(OUTFILE, ">file2.txt");                               # Open the file2.txt in write mode
while(<INFILE>)                                            #INFILE & OUTFILE are file handles
{
        print OUTFILE if(1..3);
}
close(INFILE);
close(OUTFILE);
```

5.26 Defining and Using Subroutines

- A subroutine is a block of code to perform a specific task.
- The use of subroutines results in a modular program.
- The advantages of modular approach:
  1) Code reuse
  2) Ease of debugging and
  3) Better readability.
- Syntax:

```
sub subroutine_name
{
        # Body of the subroutine
}
```

- A subroutine is called by the & symbol followed by the subroutine-name
- Inside the subroutine, the argument variable @_ is used to hold all the arguments.
- You can use @_ in your subroutines:
- Example: A script to read 2 values and display their sum using subroutine.

```
sub sum                                 # Function definition
{
        $x=$a
        $y=$b
        return $x+$y;
}

#!/usr/bin/perl                         # A main script to read 2 values and display their sum.
print "enter first number: \c"
$a=<STDIN>
print "second number: \c"
$b=<STDIN>
$result=&sum($a, $b);                   # Function call sum()
print "the sum is: $result"
```

```
Output:
enter first number: 5
enter second number: 2
the sum is: 7
```

5.27 Simple perl Program Examples

1) A script to convert a given temperature in Centigrade to Fahrenheit.

```
#!/usr/bin/perl
Print("Enter a temperature in Centigrade: ");
$centigrade=<STDIN>;
$fahr=$centigrade*9/5 + 32;
print "The temperature in Fahrenheit is $fahr \n";
```

```
Output:
Enter a temperature in Centigrade: 40.6
The temperature in Fahrenheit is 104.9
```

2) A perl script (program2.pl) to find the square root of command line arguments

```
#!/usr/bin/perl
foreach $num (@ARGV)
{
        print "The square root of $num is" sqrt($num);
}
```

```
Run:
$ program2.pl 25 36
Output:
The square root of 25 is 5
The square root of 36 is 6
```

3) A perl script (program3.pl) to determine whether the given year is leap year or not, which takes year as the command line argument.

```
#!/usr/bin/perl
#leap_year.pl
if (@ARGV == 0)
{
        die("you have not entered the year \n");
}
$year = $ARGV[0] ;
$lastdigit= substr($year, -2 , 2);
if ($lastdigit eq "00")
{
        $yesorno = ($year % 400 == 0 ? "certainly" : "not");
}
else
{
        $yesorno = ($year % 4 == 0 ? "certainly" : "not");
}
print("$year is"  $yesorno  "a leap year");
```

```
Run 1:
$ program3.pl
Output 1:
You have not entered the year

Run 2:
$ program3.pl 2004
Output 2:
2004 is certainly a leap year
```

4) A perl script(program4.pl) to convert a given decimal number to binary equivalent.

```
#!/usr/bin/perl
foreach $num (@ARGV)
{
        $temp = $num;
        until ($num == 0)
        {
                $bit = $num % 2;
                unshift(@bit_arr, $bit);
                $num = int($num/2);
        }
$binary_num = join(" ",@bit_arr);
print ("Binary form of $temp is $binary_num\n");
}
```

```
Run:
$ program4.pl 4 7 65
Output:
Binary form of 4 is 100
Binary form of 7 is 111
Binary form of 65 is 1000001
```

5) A perl script (program5.sh) to read the letters N E W S from command line and display their abbreviation.

```
#!/usr/bin/perl
%region=("N","North", "S","South", "E","East", "W","West");
foreach $letter (@ARGV)
{
        print("The letter $letter stands for $region{$letter}". "\n");
}
@key_list=keys(%region);
print("The subscripts are @key_list\n");
@value_list=values%region;
print("The values are @value_list\n");
```

```
Run:
$ program5.pl N E W S
Output:
The letter N stands for North
The letter E stands for East
The letter W stands for West
The letter S stands for South
The subscripts are N E W S
The values are North East West South
```

6) A perl script to display words of an array .

```
#!/usr/bin/perl
@list = ("This", "is", "a", "list", "of", "words");
print("Here are the words in the list: \n");
foreach $temp (@list)
{
        print("$temp ");
}
print("\n");
```

```
Output:
Here are the words in the list:
This is a list of words
```

7) A perl script using subroutine get_words to count the number of occurrences of the word "perl".

```
sub get_words    #A subroutine to read a line of input from a file and break it into words.
{
        print "enter a line \n"
        $inputline = <>;
        @words = split(/\s+/, $inputline);
}

#!/usr/bin/perl
$thecount = 0;
&get_words;                          #Call the subroutine
while ($words[0] ne " ")
{
        for ($index = 0; $words[$index] ne " "; $index+= 1)
        {
                $thecount+= 1 if $words[$index] eq "perl";
        }
        print "count of perl word : $thecount";
}
```

```
Output:
enter a line: welcome to perl programming
count of perl word : 1
```