File Attributes and Permissions

Objectives

1. ls - l : Listing fileAttributes
2. The –d option : Listing DirectoryAttributes
3. Fileownership
4. Filepermissions
5. chmod : Changing FilePermissions
6. Directory Permissions
7. Changing File Ownership

- You created files and directories , navigated the file system , and copied moved and removed files without anyproblem.
- You may have problems when handling a file or directory. Your file may be modified or even deleted byothers.
- A File also has a number of attributes(properties) that are stored in the inode.
- We will use ls –l command with additional options to display these attributes.
- We will mainly consider the two basic attributes permissions and and ownership.
- The UNIX file system allows the user to access other files not belonging to them and without infringing on**security**.

<u>ls - l : LISTING FILE ATTRIBUTES</u>

- ls –l to list seven attributes of all files in the current directory.
- It's the –l(long) option that reveals most. This option displays most attributes of a file-like its permissions ,size ,and ownershipdetails.

<u>ls lists seven attributes of all files in the current directory and they are:</u>

- File type andPermissions
- Links
- Ownership
- Group ownership
- Filesize
- Last Modification date andtime
- Filename

```
$ ls –l
total 72

1stcolumn          2nd      3rd          4th          5th        6th       7th     8th        9th

- rw - r- - r- -     1     kumar      metal     19514     may    1013:45       chap01
- rw - r- - r- -     1     kumar      metal     4174       may    1015:01       chap02
- rw - rw - rw -     1     kumar      metal     84           feb     12   12:30    dept.lst
- rw - r- - r- -     1     kumar      metal     9156       mar    121999       genie.sh
d rwx r - x r - x    2     kumar      metal     512         may    9   10:31     helpdir
d rwx r - x r - x    2     kumar      metal     512         may    9   09:57      progs
```

- Thelistisprecededbythewordstotal72,whichindicatesthatatotalof72 blocks are occupied by these files on disk.each block consisting of 512 bytes.

1) File type andpermissions:

- The first column shows the type and permission associated with eachfile.
- The first character in this column is mostly a **– (Ordinary file), d (Directory file), a,b or c(Devicefile).**
- In the UNIX system , a file can have three types of permissions **r (read), w(write),x(execute).**

2) Links:

- ThesecondcolumnindicatestheNumberofLinksassociatedwiththefile.
- The number of file names maintained by thesystem.
- A Link count greater than one indicates that the file has more than one name. This does not mean that there are two copies of thefile.

3) Ownership:

- When you create a file, you Automatically become its owner. The Third column shows kumar as the owner of thesefiles.
- The owner has the full Authority to tamper with a file’s contents and permissions.
- The owner can create, modify or remove files in adirectory.

4) GroupOwnership:

  - Whenopeningauseraccount,systemadministratoralsoassignstheuser to somegroup.
  - Fourth column represents the group owner of the file.Every user is attached to a groupowner.

5) Filesize:

  - The fifth column represents the sizes of the file in bytes .i.e amount of data it contains.

- Size is only the character count of the file, not a measure of disk space that itoccupies.
- The dept.lst contains 84 bytes , it would occupy 1024 bytes on diskon system that use a block size of 1024bytes.

6) **Last Modificationtime:**
   - The 6th,7th,and 8thcolumn indicates last modification time of thefile.A file said to be modified only if contents havechanged.
   - If you changed permissions,ownership then modification time remains unchanged.
   - If the file is old more than one year , its last modification date,year won't be displayed.
     Ex: In the file genie.sh has been modified more than a year ago.
7) **Filename:**
   - The last column displays the filename arranged in ASCII Collating sequence.

THE –d OPTION : LISTING DIRECTORY ATTRIBUTES

- ls to list the attributes of a directory, rather than its contents, you need to use the **–d(directory)**option.

Example:

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

- Directoriesareeasilyidentifiedinthelistingbythe1stcharacterofthe 1stcolumn , here shows asd.

Note:ls -d will not list all subdirectories in the current directory

FILE OWNERSHIP

- Whenyoucreateafile,youbecomeits**owner**,showsinthe3rdcolumn:
  Group owner of the file (fourth column).
- Several users may belong to a single group, people working on a project aregenerallyareassignedacommongroup,andallfilescreatedbygroup members (have separate user-id) will have same groupowner.
- But the **privileges** of the group are set by the owner of the file and not by the groupmembers.

- **Whenthesystemadministratorcreatesauseraccount,hehastoassign**

these parameters to the user:

The **user-id (UID)** – both its name and numeric representation

The **group-id (GID)** – both its name and numeric representation

- The file **/etc/passwd** maintains the UID (both the number and name) and GID(but only thenumber).
- **/etc/group** contains the **GID** (both number andname).

FILE PERMISSIONS

- UNIX follows a **three-tiered file protection** system that determines a file's accessrights.

Example: - rwx r-xr-- 1 kumar metal 20500 may10 19:21 chap02
(initial – represents an ordinaryfile)

- Each group represents a category and contains 3 slots , representing read ,write and execute permissions of thefile

r w x	r - x	r - -
owner/user	group owner	others

- Here r indicates read permission which means cat can display a file, w representswritepermissionyoucaneditafileusingeditor,xindicates executepermissionfilecanbeexecutedasaprogram.The–showsthat absence of corresponding permission.

Three types of categories:

1) **Owner(user):(rwx)**
   - The1st group(rwx)hasall3permissions.Thefileisreadable,writable, and executable by the owner of thefile.
   - 3rdcolumn shows kumar as owner and the first permissions group applies tokumar.
2) **group:(r -x)**
- The 2ndgroup (r – x) has a **hyphen** in the middle slot, which indicates the absence of write permission by the group owner of thefile.
- This group owner is metal and all users belonging to the metal grouphave read and execute permissionsonly.
3) **Others:(r --)**
- The 3rdgroup(r - -) has the write and execute bits absent. This set of permissions is applicable toothers.
- Those who are neither the owner kumar nor belonging to the metalgroup.

chmod : CHANGING FILE PERMISSIONS

- Afileoradirectoryiscreatedwithadefaultsetofpermissions,whichcan be determined by**umask**.

- To know the systems default permission create afile.
  cat > cse [ctrld]

```
$ ls-l    cse
-    r w - r - - r - -(default permission for createdfile)
```

- The chmod (change mode) command is used to set the permissions of one or more files for all three categories(user,group,other).

The command can be used in two ways:

- In a **relative** manner by specifying the changes to the currentpermissions
- In an **absolute** manner by specifying the finalpermissions

Relative Permissions

- **Whenchangingapermissioninarelativemanner,chmod**onlychanges the permissions specified in the command line and leaves the other permissions unchanged.
- In this mode it uses the followingsyntax:

```
chmod    category    operation    permission    filename(s)
          (u,g,o)      (+,-,=)       (r,w,x)
```

- Chmod takes as its argument an expression comprising some letters and symbols that describe category and type of permission being assigned or removed. The expression contains threecomponents:
- **user category** (user, group,others)
- **operation** to be performed (assign or remove apermission)
- Type of **permission** (read, write,execute)

Abbreviations used by chmod

Category	**operation**	**permission**
u->user	+ assign	**r** –read
g->group	-remove	**w** –write
o->others	=absolute	**x** – execute
a - >all (ugo)		

Examples:

```
cat >cbit
$ ls -l cbit
- r wx r - - r--    1 kumar metal1906 sep     23:38 cbit (defaultpermission)
```

// here cbit is an name of the file created.

```
$ chmod u+x cbit
 ls –l
```

```
-r w x r - - r--      1      kumar        metal1906    sep     23:38   svce
```

- The command **assigns** (+) **execute** (x) permission to the **user** (u), other permissions remain **unchanged**. Now can you execute the file if you are owner of the file but other categories(group,others) stillnot.

- To enable all of them to execute this file, you have to use multiple characters to represents the usercategory(ugo).
- The string ugo combines all three categories(user,group ,others).UNIX also offers a shorthand symbol a(all) to act as a synonym for thestring.

<u>Change permission to all group</u>

```
 $  chmodugo+xcbit        or
 $chmoda+x       cbit    or
 $chmod+x       cbit ; ls -lcbit
// You can use any of the above will give same output

$  chmod   ugo+x    cbit
 ls     -l    cbit
-rwx r – x r–x       1     kumar     metal     1906     sep     23:38     cbit
```

- **chmod accepts multiple file names in commandline**

When you need to assign the same set of permissions to a group of files this is what you should do,

$ chmod u+x notenote1note2   (give execute permission to user in all 3 files)

//here note,note1,note2 are the three different files

- **Permissions are removed with the –operator**

To remove the read permission from both group and others , use the expression go-r.

Let initially,

```
ls   -l    xstart
-r w x r - x r-x      1   kumar   metal   1906   sep23:38     xstart
```

Ex:$chmod   go-r xstart   (removereadpermissioninbothgroupandothers)

```
  ls   -l    xstart
Then, itbecomes
-r w x - - x -- x      1   kumar   metal   1906   sep23:38    xstart
   (here xstart is a filename)
```

- **chmod accepts multiple expressionsdelimitedby   a commas in commandline:**

To restore the original permissions to the file xstart ,remove the execute permission from all(a-x) and assign read permissions to group and others(go+r).

```
$ chmod a-x , go+r xstart
 ls –l xstart
-r w – r - - r--       1    kumar     metal     1906     sep     23:38     xstart
```

chmod accepts more than one permission in a commandline:

u+rwx is a valid chmod expression .so setting write and execute permissions for others is no problem.

```
$ chmod o+wx xstart
ls -l xstart
-r w – r - - rwx     1    kumar    metal     1906     sep   23:38     xstart
```

Absolute Permissions

- Here, we need not to know the current filepermissions.
- Wecansetallninepermissionsexplicitly.Astringof**threeoctaldigits**is used as anexpression.
- Thepermissioncanberepresentedbyoneoctaldigitforeachcategory.For each category, we add octaldigits.
- If we represent the permissions of each category by one octal digit, this is how the permission can be**represented:**
  - Read permission – **4** (octal100)
  - Write permission – **2** (octal010)
  - Execute permission – **1** (octal001)

Permissions	**Significance**
- --	nopermissions
- -x	execute only
- w -	writeonly
- wx	write andexecute
r--	readonly
r-x	read andexecute
r w -	read andwrite
r w x	read, write andexecute

- Wehavethreecategoriesandthreepermissionsforeachcategory,sothree octal digits can describe a file's permissionscompletely.
- The most significant digit represents user and the least one represents others. chmod can use this three-digit string as theexpression.

Example:

Using relative permission, we have,

```
$ chmod a+rw xstart
ls –l xstart
```

Binary	**Octal**
000	0
001	1
010	2
011	3
100	4
101	5
110	6

```
-r w - r w - rw-      1     kumar      metal    1906     may     1020:30     xstart
```

Using absolute permission, we have,

```
$chmod666      xstart
 ls –lxstart
-r w - r w - rw-      1     kumar      metal     1906     may     1020:30     xstart
```

Note: The 6 indicates read and write permissin(4+2).To restore the original permissions to the file, you need to remove the write permission (2) from group and others.

To restore the original permission to the file

```
$ chmod    644     xstart
 ls -l xstart
-r w -  r - - r--        1     kumar      metal     1906     may   10 20:30    xstart
```

- willassignallpermissionstotheowner,readandwritepermissionsforthe group and only execute permission to theothers.
  Ex:$ **chmod   761   xstart**
- The expression 777 signifies all permission for all categories , while 000 indicates absence of all permissions for allcategories.

<u>The Security Implications</u>

- Let the default permission for the file xstart is

```
  -r w -  r - - r--       1    kumar      metal     1906     may   1020:30    xstart
```

- These permissions are fairly safe ;only the user can edit the file.What are the implications if we remove all permissions in either of theseways.

```
 $ chmod  u-rw,go-r      xstart
    or

$  chmod   000    xstart
```

The listing in either case will look like this

```
    - - - - - - - -- -     1    kumar      metal     1906     may   1020:30     xstart
```

- This setting renders the file virtually useless, you can’t do anything with it ;But still user can delete thisfile.

- On the other hand you must not careless ,enable all permissions for all categories using neither of thesecommand.

```
$ chmod    a+rwx     xstart
        or
$ chmod     777    xstart
```

The resulting permissions setting is simply dangerous:

```
-r w x r w x rwx      1    kumar       metal     1906      may   1020:30     xstart
```

- The UNIX system by default, never allows this situation as you cannever have a secure system. Hence, directory permissions also play a very vital rolehere.

Using chmod Recursively (-R)

- It's possible to make chmod descend a directory hierarchy and applythe expression to every file and subdirectory it finds. This is done with-R

```
chmod   -R    a+x    shell_scripts
```

This makes all the files and subdirectories found in the shell_scripts directory, executable by all users.

DIRECTORY PERMISSIONS

- It is possible that a file cannot be accessed even though it has read permission, and can be removed even when it is writeprotected.
- The default permissions of a directoryare,

r w x r - x r-x   (755)

A directory must never be writable by group and others

Example:

```
$ mkdirc_progs
$   ls –ldc_progs
d r w x r - x r - x   2  kumarmetal512       May      9     09:57      c_progs
```

- If a directory has write permission for group and others also, be assured that every user can remove every file in thedirectory.
- As a rule, you must not make directories universally writable unless you have definite reasons to doso.

CHANGING FILE OWNERSHIP

- Usually, on BSD (Berkeley's Software Distribution) and AT&T systems, there are two commands meant to change the ownership of a file or directory.
- If kumar creates a file he become the owner and metal be the group owner. Only kumar can change the file major attributes like (permissions and groupownership).

- If sharma copies a file of kumar, then sharma will become its owner and he can manipulate theattributes.
- **chown** changing file owner and **chgrp** changing groupowner
    - On BSD, only **system administrator** can usechown
    - On other systems, only the **owner** can changeboth

chown: changing File Owner

- This command can be used to change the ownership of afile
- Syntax:
  chown USERNAME FILENAME
- Changing ownership requires **superuser** permission, so use **su**command

```
$ su
Password:*******                    //This is rootpassword
# -                                 // This is anothershell
```

- After the password successfully entered, su returns a # prompt ,same prompt used by root su lets acquire

Ex: $ ls -lnote

```
-r w x r - - --x     1      kumar     metal     347     may 10 20:30note

    $ chown   sharmanote                    //note is name of thefile
      $ ls -l note
   -r w x r - - -- x    1         sharma metal 347 may 10 20:30note
```

- Once ownership of the file has been given away to sharma, the user file permissions that previously applied to Kumar now apply tosharma.
- Thus, Kumar can no longer edit *note* since there is no write privilege for groupandothers.Hecannotgetbacktheownershipeither.Buthecancopy the file to his own directory, in which case he becomes the owner of the copy.

chgrp:changing group owner

- Thiscommandchangesthefile'sgroupowner.Nosuperuserpermissionis required.
- chgrp shares the similar syntax with chown,in the following example kumar changes the group of dept.lst to dba.
  Ex:$ ls -l dept.lst

```
 -r w- r - - r--    1    kumar    metal   139    jun816:43       dept.lst

    $ chgrp dba dept.lst
    $ ls –l dept.lst
    -r w - r - - r --     1    kumar    dba    139    jun    8    16:43    dept.lst
```

- This command will work on a BSD-based system if kumar is also a member of the group.if he is not,then only the superser can make the command work.
- Kumarcanreversethisactionandrestorethepreviousgroupownership(to metal) because he is still owner of the fileand consequently retains all related toit.

Using chown to Do Bth

- UNIX allows the administrator to use only chown to change both owner and group.
- The syntax requires the two arguments to be separated by a:

Example:

chown sharma :dba deptlist // ownership to sharma,group to dba

Note:

Like chmod,both chown and chgrp use the **–R** option to perform their operations in a **recursive** manner.

The Shells Interpretive Cycle

Introduction:

- WhenyoulogontotheUNIXmachine,youfirstseeaprompt.Thisprompt remails there until you key insomething.($)
- This command is a special its with you all the time and never terminates. Unless you log out this command is theshell.
- The shell first scans the command line formetacharacter.
- The metacharacters like >,|,*.It performs all the actions represented bythe symbol before the command can beexcutes.
- Ex: cat > foo
  rm–r*

The following activities are performed by shell in its interpretive cycle

- The shell issues the prompt($) and waits for you to enter a command.(ex: like ls chap*).
- After a command is entered , the shell scans the command line for metacharacters (like ‘ls chap*’) and expand abbreviations to recreate a simplified command line (‘ls chap1chap2’).
- It then passes on the command line to the kernel forexecution.
- The shell waits for the command to complete and normally can’t do any work while the command isrunning.
- Afterthecommandexecutioniscompletetheshellissuesprompt($)again and wait for the user to enter nextcommand.

SHELL OFFERINGS:

Categories of shell:

- The Bourne family comprising the Bourne shell (/bin/sh) and its derivatives the korn shell (/bin/ksh) andBash(/bin/bash)
- The C Shell (/bin/csh) and its derivatives ,Tcsh(/bin/tcsh).
- Whenyourunecho$SHELLtheoutputdisplaystheabsolutepathnameof the shell’s commandfile.

- Bash is near POSIX-compliant and is probably the best shell to use,Korn should be next.

PATTERN MATCHING-WILD CARDS

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

The shell’s 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

1) Metacharacter The * and?

- The metacharacter * is one of the characters of the shell’s wild cardset.
- It matches any number of characters(Including None).
- Ex: lschap*

chap chap01chap02 chap03 chap04 chap13 chapxy chapabchaprt

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

ls chap chap01 chap02 chap03 chap04 chap13 chapxy chapab chaprt

- The shell now hands over this command to the kernel which uses its process creation facilities to run thecommand.

The next wild card is ?,Which matches a single character.When used with the samestringchap(chap?),theshellmatchesallfivecharacterfilenamesbeginning with chap.

- Appending another ? creates the pattern chap??,which matches six-characterfilenmaes.
- Ex: $ lschap?
  chapx chapy chapz //five characterfilenames
- $ lschap??
  Chap01 chap02chap15chapxy //six-characterfilenames

2) **Matching the Dot**

- The behaviour of the * and ? In relation to the dot is not straightforward.
- The * doesn't match all files beginning with a . (dot) or / of apathname.
- If you want to list all the hidden files in your directory having at ssssleast three characters after the dot , the must be matchedexplicitly.
- Ex: **$ls.??***

bash_profile .exrc .netscape .profile

- If filename contains dot at anywhere but at the beginning, it need not be matched explicitly.
- Ex: **$ lsemp*lst**
  emp.lst emp1.lst emp22lst emp2.lstempn.lst

3) **The characterclass**

- The character class comprises a set of characters enclosed by the rectangularbrackets,[and],butitmatchesasinglecharacterintheclass.
- The pattern [abcd] is a character class and it matches a singlecharacter -an a,b,c, or d.
- Ex:1) **$ ls chap0[124]**
  chap01 chap02chap04
  2) $ ls chap[x-z]
  chapx chapy chapz
- The expression [a-zA-Z]* matches all filename beginning with an alphabet ,irrespective ofclass.
- **Negating the character class(!):** Not operator (!) can can be used to negate theclass.
- You can use the ! as the first character in the class to negate theclass.
- The two examples below should make this pointclear:
  Ex:*.[!co] // Matches all filenames with a single character extensions but not the .c or .ofiles
  [!a-Za-Z]* // Matches all filenames that don't begin with an alphabeticcharacter.
  $ ls chap[!1-3] //lists the filenames begin from chap but not chap1 chap2 chap3

4) **Matching Totally Dissimilarpatterns**

- This feature, not available in the Bourne shell enables us to match totally dissimilarpattern.
- Ex: **cp $HOME/prog_sourcse/*.{c,java}** //To copy all the c and java source programs from another directory to the currentdirectory.
- **cp/home/kumar/{project,html,scripts}/*** //To copy all files from 3 directories {project,html and scripts}to the currentdirectory**.**

5) **Roundingup**

- The *and? Lose their meaning when used inside the class, and matched literaly.
- - and ! Lose their significance when placed outside theclass.
- To summarize , simple set of command lines is presentedbelow:

ls*.c //Lists all files with extension .c
cpfoo foo* //copies foo to foo*(* loses meaning here).
rm*.[!][!0][!g] //Removes all files with three-character extensions except the once with the dot . logextension
lpnote[0-1][0-9] // prints files note00,note01,note03….throughnote19

cp??????Progs //copiestoprogsdirectoryallfileswithsixcharacter names.

Removing the special meaning of wild cards (ESCAPING AND QUOTING):

- The output below shows a file namedchap*
- $ lschap*

chap chap* chap01 chap02 chapx chapv chapt

- Tryingrmchap* would be dangerous; it removes the other filenames beginning with chapalso.
- We must able to protect all special character including (wildcards).
- The shell provides two solutions to prevent its own interference:
  Escaping: providing a \ (backslash) before the wild card to remove (escape) the specialmeaning.
  Quoting: Enclosing the wild card or even the entire pattern within quotes (like ‘chap*’).

ESCAPING (\) :

- Placinga \ immediately before a metacharacter turns off its special meaning.
- we can remove the file chap* without affecting the other filenames begin with chap.
- **$ rmchap\*** // Doesn’t remove chap1,chap2
- The\supressesthewild-cardnatureofthe*,thuspreventingtheshellfrom performing filename expansion on it. this feature is known asescaping.
- Toconsideranotherexample,ifyouhavefileschap01,chap02andchap03 in your current directory and still dare to create a file chap0[1-3] using echo
  >chap0[1-3] //creates a filechap0[1-3]
  then you should escape the two rectangular brackets when accessing the file:
  ```
  $ ls chap0\[1-3\]                    Must escape the[and]
     chap[1-3]
  $ rm chap0\[1-3\]
  $ ls chap\[1-3\]
  chap0[1-3]not found                  File removed
  ```
- **Escaping the Space:** Apart from metacharactes, there are other character like spacecharacter.
  Ex: rmMy\Document.doc without the \ rm would see twofiles
- **Escapingthe\itself:**Sometimesneedtointerpretethe\itselfliterally.you need another \ beforeit.
  Ex: $ echo \\\
  \

$ echo The newline character is \\nThe newline character is \n

- **Escaping the Newline Character:** The new line character is also special it makes the end of the command line. Some command lines that use several arguments can be long enough to overflow to the nextline.
- Toensurebetterreadability,youneedtosplitalineintotwolinesandinput a \ before you press [Enter].$

```
$ find /usr/local/bin /usr/bin-name “”.p1”-mtime +7 -size +1024 \[Enter]

>-size -2048 -atime +25 -print
```

- The \ here escape the meaning of the newline character generated by [Enter].

QUOTING: Another way to turn off the meaning of a metacharacter ,when a command is enclosed with quotes the meaning of all enclosed special characters are turnedoff.

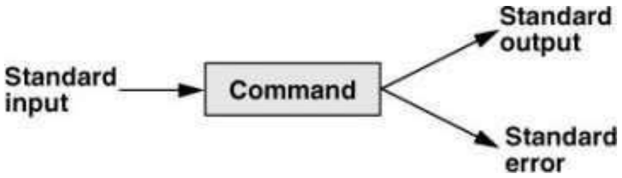
Ex:echo ‘\’	Displays a\
rm ‘chap*’	Removes filechap*
rm“MyDocument.doc”	Removes file MyDocument.doc

- Quotingisoftenabettersolution.Thefollowingexampleshowsprotection of four character using singlequotes.
  $ echo ‘The characters | ,< , > and $ are also special’
  The characters |, <, > and $ are alsospecial.
- Single quotes protect all special characters.Double quotes are more permissive ; they don’t protect $ and‘(backquote).
  Ex:**echo‘$HOME’** //displays$HOME
  $HOME
  echo “$HOME” //displays contents of environment variable $HOME
  /home/svce
- **Escaping in echo:**These escape sequences are always used within quotes to keep the shell out .But what isecho?
  Ex: $type echo
  echo is a shell builtin

REDIRECTION: THE THREE STANDARD FILES:

- The shell associates three standard files with theterminal:
  → two for display and
  → one for the keyboard.
  - When a user logs in, the shell makes available three standardfiles.
- Each standard file is associated with a defaultdevice:
  1) **Standard input:** The file representing input which is connected tothe keyboard.
  2) **Standard output**: The file representing output which is connected to thedisplay.

3) **Standard error:** The file representing error messages that come from the command or shell. This file is also connected to thedisplay.



1) **StandardInput**:

- The standard input can represent three inputsources:
  1) The keyboard, the defaultsource.
  2) A file using redirection with the <symbol.
  3) Another program using apipeline.
- By default, the shell directs standard input from thekeyboard.
  - ➢ Use wc without an argument and no symbols like <, | in command linewc obtain input from default source.provide input from the keyboard and mark end of input with[ctrl-d].
  - ➢ Ex: $wc

```
Hello
World
[ctrld]                    //end ofinput
2    2   10                //output
```

  - ➢ The shell's manipulative nature finds place here .It can reassign the standard input file to a disk file.It can redirect the standard input to originate from a file on disk.This redirection requires the <symbol
  - ➢ Ex: $catsample.txt

```
Hello
 World
 $ wc < sample.txt
    2   2   10    //outputs
```

2) Standardoutput:

- The standard output can represent three possible destinations:
  1] The terminal, the defaultdestination.
  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 thescreen.
  - ➢ The shell can effect redirection of this stream when it sees the > or >> symbols in the command line.You can replace the default destination with any file using > (right chevron) operator, followed by the filename.
  - ➢ Ex:$wc sample.txt >temp.txt

```
 $cat remp.txt
   2   2   10     //output of c stored in temp.txt
```

  - ➢ The 1st command sends the word count of sample.txt to temp.txt;The shell also provide the >> symbol (used twice) to append a file.

- $wc sample.txt >> temp.txt

```
  2 210
  2 210
```

3) Standard Errors: Each of the three standard files is represented by a number called **filedescriptor.**

- Afileisopenedbyreferringtoitspathname,butsubsequentreadandwrite operations identify the file by this filedescriptor.
- The kernel maintains a table of file descriptors for every process running in the system.The 1st three slots are generally allocated to three standard streams in thismanner.

  0 Standardinput
  1 Standardoutput
  2 Standard error
- Trying to "cat " a non-existent file produces the errorstream.
- Ex: $ catfoo

  cat: cannot open foo
- Cat files to open the file and writes to standard error.if you are not using the c shell you can redirect this stream to afile.
- The redirect output symbol (>) instructs the shell to redirect the error messages of acommandto the specified file instead of to thescreen.
- Ex: $cat foo>errorfile

  cat:cannotopenfoo //error stream can't be captured with>
- Redirecting the standard error requires use of 2>symbols.
- Ex: cat foo2>errorfile

  $cat errorfile

  cat:cannot open foo

Filters: Using Both Standard input and output:

- **UNIX commands can be grouped into fourcategories:**

  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) Commandslikels,pwd,whoetcdon'treadstandardinputbuttheywrite to standardoutput.
  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.
- Commandslikecat,wc,cmpetc.thatusebothstandardinputandstandard output are called filters.
- Let'susebccommandasafilterthistime,considerthefilecontainingsome

arithmetic expressions;

- ➢ Ex: $cat calc.txt

```
2^32
25*50
30*25   +   15^2
```

- ➢ Youcanredirectbc’sstandardinputtocomefromthisfileandsaveoutput in yetanother,
- ➢ $ bc <calc.txt>result.txt

```
$ cat result.txt
4294967296          //this is2^32
1250                //this is25^50
975                 //this is 30*25+15^2
```

Regular Expression

We often need to search a file for a pattern, either to see the lines containing (or not containing) it or to have it replaced with something else. This chapter discusses two important filters that are specially suited for these tasks – grep and sed. grep takes care of all search requirements we may have. sed goes further and can even manipulate the individual characters in a line. In fact sed can de several things, some of then quite well.

grep – searching for a pattern

It scans the file / input for a pattern and displays lines containing the pattern, the line numbers or filenames where the pattern occurs. It’s a command from a special family in UNIX for handling search requirements.

$grep *options pattern filename(s)*

$grep “sales” emp.lst

will display lines containing sales from the file emp.lst. Patterns with and without quotes is possible. It’s generally safe to quote the pattern. Quote is mandatory when pattern involves more than one word. It returns the prompt in case the pattern can’t be located.

$grep president emp.lst

When grep is used with multiple filenames, it displays the filenames along with the output.

```
$grep “director” emp1.lst emp2.lst
```

Where it shows filename followed by the contents

grep options

grep is one of the most important UNIX commands, and we must know the options that POSIX requires grep to support. Linux supports all of these options.

-i	ignores case formatching
-v	**doesn’t display lines matchingexpression**
-n	**displays line numbers along withlines**
-c	**displays count of number ofoccurrences**
-l	**displays list of filenames only**
-e exp	**specifies expression with thisoption**
-x	**matches pattern with entireline**
-f file	**takes pattrens from file, one perline**
-E	**treats pattren as an extendedRE**
-F	**matches multiple fixedstrings**

Examples:**$grep -i ‘agarwal’ emp.lst**

```
$grep -v ‘director’ emp.lst > otherlist
```

wc -l otherlist will display 11 otherlist

$grep –n ‘marketing’ mp.lst $grep

–c ‘director’ emp.lst $grep –c

‘director’ emp*.lst

will print filenames prefixed to the line count

```
$grep –l ‘manager’ *.lst
```

will display filenames *only*

```
$grep –e ‘Agarwal’ –e ‘aggarwal’ –e ‘agrawal’ emp.lst
```

will print matching multiple patterns

```
$grep –f pattern.lst emp.lst
```

all the above three patterns are stored in a separate file *pattern.lst*

Basic Regular Expressions (BRE) – An Introduction

It is tedious to specify each pattern separately with the -e option. grep uses an expression of a different type to match a group of similar patterns. If an expression uses meta characters, it is termed a regular expression. Some of the characters used by regular expression are also meaningful to theshell.

BRE charactersubset

The basic regular expression character subset uses an elaborate meta character set, overshadowing the shell’s wild-cards, and can perform amazing matches.

The character class

grep supports basic regular expressions (BRE) by default and extended regular expressions (ERE) with the –E option. A regular expression allows a group of characters enclosed within a pair of [ ], in which the match is performed for a single character in the group.

```
$grep “[aA]g[ar][ar]wal” emp.lst
```

A single pattern has matched two similar strings. The pattern [a-zA-Z0-9] matches a single alphanumeric character. When we use range, make sure that the character on the left of the hyphen has a lower ASCII value than the one on the right. Negating a class (^) (caret) can be used to negate the character class. When the character class begins with this character, all characters other than the ones grouped in the class are matched.

The *

The asterisk refers to the immediately preceding character. * indicates zero or more occurrences of the previous character.

g* nothing or g, gg, ggg, etc.

```
grep “[aA]gg*[ar][ar]wal” emp.lst
```

Notice that we don’t require to use –e option three times to get the same output!!!!!

The dot

A dot matches a single character. The shell uses ? Character to indicate that.

.* signifies any number of characters ornone

```
grep “j.*saxena” emp.lst
```

Specifying Pattern Locations (^ and $)

Most of the regular expression characters are used for matching patterns, but there are two that can match a pattern at the beginning or end of a 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 occurance only at a particular location.

^	for matching at the beginning of a line
$	for matching at the end of aline

```
$grep “^2” emp.lst
```

Selects lines where emp_id starting with 2

```
$grep “7…$” emp.lst
```

Selects lines where emp_salary ranges between 7000 to 7999

```
$grep “^[^2]” emp.lst
```

Selects lines where emp_id doesn’t start with 2

When meta characters lose their meaning

It is possible that some of these special characters actually exist as part of the text. Sometimes, we need to escape these characters. For example, when looking for a pattern g*, we have to use \

To look for [, we use \[To look for
.*, we use \.*

Extended Regular Expression (ERE) and grep

If current version of grep doesn’t support ERE, then use egrep but without the –E option. -E option treats pattern as an ERE.

+	matches one or more occurrences of the previouscharacter
?	Matches zero or one occurrence of the previous character b+

matches b, bb, bbb,etc.

b? matches either a single instance of b or nothing

These characters restrict the scope of match as compared to the *

```
$grep –E “[aA]gg?arwal” emp.lst
```

# ?include +<stdio.h>

The ERE set

ch+	matches one or more occurrences of characterch
ch?	Matches zero or one occurrence of characterch
exp1|exp2	matches exp1 orexp2
(x1|x2)x3	matches x1x3 orx2x3

Matching multiple patterns (|, (and))

```
$grep –E ‘sengupta|dasgupta’ emp.lst
```

We can locate both without using –e option twice, or

```
$grep –E ‘(sen|das)gupta’ emp.lst
```

SHELL PROGRAMMING

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
```

Ordinary and Environment Variables

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

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
```

8. 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.

9. 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.

10. LOGNAME

- This variable shows the username.

11. MAIL

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

12. MAILCHECK

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

13. 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
```

14. PS1 and PS2

- The shell has 2 prompts:
  - 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
```

  - The secondary prompt > is stored in PS2.

15. SHELL

- This variable specifies the current shell being used by the users.
- Different types of shells are:
  - 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.

16. 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.

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

6) 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.

7) 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.

8) 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
```

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:
  - ➢ A specific file for each individual user with responsibility for the user environment.
  - ➢ 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
```

read and readonly Commands

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:
```

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
```

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
```

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

  8) zero (0) upon successful execution and

  9) 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
```

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
```

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:

  10) Compare two numbers.

  11) Compares two strings or a single one for a null value.

  12) Checks files attributes.

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
```

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

                                                                    else
echo "Both strings are equal" echo "Strings
are unequal"
```

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

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
        else fi
echo “File exists \n”

echo “File does not exist \n”
        if [ -r "$file" ]
        then
        else fi
```

```
 echo "File is readable \n " echo "File is not
readable \n "
         if [ -w "$file" ]
         then
         else     echo "File is writable \n "
         fi       echo "File is not writable \n "
         if [ -x "$file" ]
         then
                          echo "File is executable \n " echo "File is not executable \n "
         else
         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
```

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
  else      execute statements
  fi        execute statements
```

- Syntax 3:

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

```
        else
    fi
```

- Here is how it works:
  - If the command succeeds, the statements within then-block are executed.
  - 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
```

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:

4) Firstly the expression is matched with pattern1.

5) If the match succeeds, then statement1 will be executed.

6) 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
```

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
```

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
```

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:
  - Performs arithmetic operations on integers and
  - 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
  ```

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
```

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
```

Possible Sources of List

- Possible sources of list are
  - List from variables
  - List from command substitution
  - List from wildcards and
  - 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
```

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:
```

set and shift Commands and Handling Positional Parameters

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
```

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
```

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
```

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
```

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:
    - Clean up temporary files and
    - Ignore signals

➢ **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.

➢ **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.

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-
```

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:
```

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
        ▪ Quit \n
        Enter your option: \c" read
choice
```

```
case “$choice” in
                •  ls –l;;
                •  ps –f ;;
                •  date ;;
                •  who ;;
                •  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
```

- 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:
```

- 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
        •   echo "The string has less than 10 characters" "
        *) echo "The string has $length characters" ;;
esac
```

```
Output:
Enter a string:
```

- 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 `
        done    reverse=`expr $reverse \* 10 + $a`
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
```

- 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
```

- 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:
```

- 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
```

- 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
fi      echo "ACCESS DENIED!"
```

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

- 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
```

- 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
fi      echo “You can proceed!”
```

```
Output:
Enter your name: Rama
```