Smitha. N.
Asst. Prof
CSE.

Quoting:

→ another way to turn off the meaning of a metacharacter by enclosing command argument in quotes -

→ $ echo '|'
|

$ rm 'chap*'
removes file chap*

$ rm "My document.doc"
removes file My document.doc.

→ Single quotes is flexible because they protect all special characters.

Eg: $ echo 'The characters |, <, > and $ are special'
The characters |, <, > and $ are special.

Three standard files and redirection

→ Terminal is a generic name that represents the screen, display & keyboard

→ We can see command output and error message on the terminal (display)

→ command input is provided through the terminal (keyboard)

→ commands read from & write with three std files that the shell makes available to every command.

→ These special files are streams of characters which many commands see as input and output.

→ When user logs-in, the shell makes available three files representing three streams, each streams associated with a default device (terminal).

Standard Input :-

→ wc, cat command if used without arguments, they read file representing the std i/p.

→ It represent three input sources :

* The keyboard, the default source.
* A file using redirection with < symbol (& metacharacter)
* Another program using a pipeline.

→ wc without argument, then wc obtains its input from default source.

```
$wc
hi how are you
[ctrl-d]
 1  4  14
```

→ $wc < sample.txt

```
 3  14  71
```

Here redirects the std o/p file to a disk file.

This means it can redirect the std i/p to originate from a file on disk.

Using < Symbol.

How it works?

① On seeing the <, the shell opens the disk file, sample.txt for reading.

② It unplugs the std input file from its default source & assigns it to sample.txt.

③ wc reads from std input which has earlier been reassigned by the shell to sample.txt.

Standard output :-

→ There are three possible destinations of this stream:

* The terminal, the default destination.
* A file using the destination symbols > &>>
* As input to another program using a pipeline.

→ We can replace the default destination (the terminal) with any file by using the > operator, followed by the filename.

```
$wc sample.txt > newfile
$cat newfile
3   14   71   sample.txt
```

* First command sends the word count of sample.txt to newfile, nothing appears on the terminal screen.
* If output doesn't exists, shell creates it, if file exists, the shell overwrites it.

* Use >> symbol to append to a file

$ wc sample.txt >> newfile.

* How it works?

→ wc sample.txt > newfile

→ on seeing the >, the shell opens the disk file, newfile, for writing

→ It unplugs the Standard output file from its default destination & assigns it to newfile.

→ wc (and not the shell) opens the file sample.txt for reading.

Standard Error :

→ Std files are represented by a number called a file descriptor.

→ 0 → Std i/p
1 → std o/p
2 → std error

→ 1> & > mean the same thing to the shell

→ $ cat foo
$ cat : cannot open foo
$ cat foo > errorfile
$ cat : cannot open foo

Error Stream can't be captured with >.

→ $cat foo 2> errorfile

$ cat errorfile

cat : cannot open foo .

Using file discriptor '2' can be used to stream error to file.

Filters : Using both standard I/P & Std o/P :-

Pipes :-

→ Std i/p & std o/p constitute two separate streams than * individually manipulated by the shell.

→ In UNIX, it is possible to take one command o/p as input to other command.

→ Eg:- $ who > user.txt

$ cat user.txt

root console Aug 1 07:51 (:0)

romeo pts/10 Aug 1 07:56 (pc123.hoarau.com)

$ wc -l < user.txt

5 counts the number of users.

disadv

* Process is slow, second command can't act unless the first has completed its job.
* intermediate file is created that has to be removed after job. It will eat up disk space

Solution :- pipes.

$ wc | wc -l

5

No intermediate files created.

When a command Needs to be Ignorant of its Source:-

* Sometimes a command being ignorant of its source & destination.

* Eg: ① wc -c *.c

```
2078    swap.c
231     array.c
:
94101   total.
```

② cat *.c | wc -c

```
94101
```

Basic & Extended Regular Expressions :-

→ We often search a file for a pattern, either to see the lines containing (@ not containing) it @ to have it replaced with something else.

→ We have two filters for this purpose

* grep and
* sed.

Grep :- Searching for a Pattern

→ Grep scans input for a pattern and displays lines containing the pattern, the line numbers @ filenames where the pattern occurs.

→ Syntax :-

```
grep options pattern filename(s).
```

* Grep searches for pattern in one @ more filename @ std i/p if no filename is specified.

Eg:- Display lines containing the string sales from the file emp.lst

```
$ grep "sales" emp.lst
2233 | a.K.Shukla | g.m | sales | 12|12|52|6000
2476| anil aggarwal | manger | sales | 01|5|59| 5000
```

→ Grep is also a filter, it can search its standard input for the pattern and also save the std output in a file.

```
$who | grep kumar > foo
```

→ Grep returns $ prompt, if pattern can't be located.

→ Grep is used with multiple filenames, it displays the filenames along with the output.

① Ex1-

```
$ grep "director" emp1.lst emp2.lst
emp1.lst: 6006 | chanchal | director | sales | 3/9/68 | 6700
emp2.lst: 2345 | barun | director | personnel | 1/5/47 | 7800
```

grep options :-

→ Ignoring case (-i) :-

*. If we are not sure of the case, use the -i (ignore) option. This option ignores case for pattern matching

*. Eg:- $grep -i 'agarwal' emp.lst
3564 | sudir Agarwal | executive | personnel | 06/7/47 | 7500

*. Deleting Lines (-v) :- grep can play an inverse role too. (-v) options selects all lines except those ~~option~~ containing the pattern.

Eg:- Create a file otherlist containing all but directors.

```
$ grep -v 'director' emp.lst > otherlist
$ wc -l otherlist
11 otherlist
```

*. Displaying Line Numbers (-n) :-

→ (-n) option displays the line numbers containing the pattern, along with the lines:

```
$grep -n 'marketing' emp.lst
3:5678 | sumit chakrobarty | d.g.m | marketing | 19/04/600
```

* Counting lines containing Pattern (-c) :-

→ -c option counts the number of lines containing the pattern

→ Eg1-

① $grep -c 'director' emplst

4.

② $grep -c 'director' emp*.lst

emp.lst : 4

emp1.lst : 2

emp2.lst : 2

empOld.lst : 4

Here, multiple files are checked and above output displayed.

Displaying Filenames (-l) :-

→ -l options displays only the names of files containing the pattern.

Eg1- $grep -l 'manger' *.lst

desig.lst

emp.lst

4. matching Multiple Patterns (-e)

→ With the -e option we can match the three agarwals by using grep like this

```
$ grep -e "Agarwal" -e "aggarwal" -e "agarwal" emplst
2476|anil aggarwal|manger|sales|05|01|59|5000
3564|sudhir Agarwal|Executive|personnel|07|06|47|7500
0110|v.k.agarwal|g.m|marketing|12|31|40|9000
```

5. Taking patterns from a file (-f) :-

→ we can place all three patterns in a separate file, one pattern per line.

→
```
$ grep -f pattern.lst emp.lst
```

Here all the patterns are kept in a separate file i.e pattern.lst

Basic Regular Expressions (BRE)

→ Typing each pattern separately with -e option is tedious task.

→ We can match similar filenames with a single expression to match a group of similar patterns.

→ If an expression uses any of the metacharacters then this expression is termed as Regular Expression.

→ Regular Expressions take care of some common query and substitution requirements.

→ POSIX identifies Regular Expressions as 2 categories
  → Basic &
  → Extended.

→ grep supports Basic Regular Expressions (BRE) by default & Extended Regular Expressions (ERE).

The Basic Regular Expression (BRE) Character Subset:-

Symbol @ Expression	Matches
① *	Zero @ more occurrence of the previous character
② g*	Nothing @ g, gg, ggg etc
③ .	A single character
④ .*	Nothing @ any no of character.
⑤ [pqr]	A single character p, q, @ r.
⑥ [c1 - c2]	A single character within the ASCII range represented by c1 and c2
[1 - 3]	A digit between 1 and 3
[^pqr]	A single character which is not a p, q @ r.
[^a-z A-Z]	A non-alphabetic character
^pat	Pattern pat @ beginning of l
pat$	pattern pat at end of lin
bash$	bash at end of line.
^bash$	bash a$ the only word in
A$	line containing nothing

① The character class :-

* A Regular Expression lets us to specify a group of characters enclosed within a pair of rectangular brackets, [ ], match is performed for a single character in the group.

* Eg:- agarwal, aggarwal, Agarwal

RE:- [aA]g[ar][ar]wal

[aA] → matches the letter a in both lowercase and uppercase.

[ar][ar] → matches any of 4 patterns.

aa ar ra rr.

$grep "[aA]g[ar][ar]wal" emp.lst

3564 | subir Agarwal | Executive | personnel | 07 | 06 | 47 | 7500

0110 | V.K. agrawal | g.m | marketing | 12 | 31 | 40 | 01000

* Negating a class (^) :- (caret)

RE uses ^ , Shell uses (!) (bang).

Eg:- [^P] → any character other than P.

⑨ The * :- (asterisk)

→ * refers to the immediately preceding character.

→ $g*

g gg ggg gggg .... + null string.

matches the single character g @ any no of gs.

@ if previous character may not occur at all.

it's also matches a null string.

→ if one pattern aggarwal contains an extra g.

while other patterns don't

∴ $grep "[a-z]gg*[ar][ar]wal" emp.lst.

→ anil aggarwal

→ Sudhir Agarwal

→ U. K. agrawal

---

③ The Dot :-

→ A . matches a single character. but shell uses

? character to indicate that.

→ Eg:- 2.... → four-character pattern expression

a 2

→

The Regular Expression .*

→ signifies any number of characters (or) none

→ Eg:- $ grep "j.* saxena" emp.lst

2345| j.b. saxena |g.m |marketing |03|p-|75|80

(4) Specifying Pattern locations (^ and $):

→ Match a pattern at the beginning (or) end of a line.

→ There are two characters that are used:

^ (caret) – for matching at the beginning of a line

$ → for matching at the end of a line.

→ Eg. Extract those lines where the emp-id begins with a 2.

if 2... is used, will help ∴ character followed by three characters can occur anywhere in the line.

→ Using grep, if we want to specify at the beginning of the line, ^ does it easily:

Eg:- $ grep "^2" emp.lst.

2233 | - - -
2365 | - - -
2476 | - - -

→ 1114, select those line wheere salary lies b/w 7000 and 7999, use $ at the end of pattern.

$ grep "7...$" emp.lst.

- - - | 7000
- - - | 7800
- - - | 7500.

→ Select only those lines wheere emp-ids don't be with a 2?

$ grep "^[^2]" emp.lst.

→ List only directory.

$ ls -l | grep "^d"

Extended Regular Expressions (ERE) and egrep :-

→ ERE makes it possible to match dissimilar patterns with a single expression

→ use $ grep -E option (or)
$ egrep

→ ERE set used by grep, egrep.

Expression	Significance
(1) ch+	Matches one (or) more occurrences of character ch.
(2) ch?	Matches zero (or) one occurrence of character ch.
(3) exp1 | exp2	matches exp1 (or) exp2
(4) (x1 | x2) x3	Matches x1x3 (or) x2x3
(5) (lock|ver)wood	Matches lockwood (or) verwood.

The + and ? :-

→ ERE set includes 2 special characters, + & ? often used in place of * to restrict the matching scope.

→ they signify the following

+ → Matches one (or) more occurrence of pre
? → " ——— zero (or) one ——— "

→ Both case emphasis is on the previous character.

Eg:- b+ matches b, bb, bbb etc.

but b* matches nothing as well.

b? matches single instance of b or nothing.

Both restrict scope of match as compared to the *.

→ Agarwal & aggarwal, g occurs only once or twice so gg? restricts the expansion to one or two gs only

Eg:- $grep -E "[aA]gg?arwal" emp.lst.

---|anil aggarwal |

---|Sudhir Agarwal |

→ multiword string like #include<stdio.h>, if we don't how many spaces separated by #include <stdio.h>.

#include <stdio.h> #include <stdio.h> #include

Exp :- #[ ]?include[ ]+<stdio.h>

② Matching Multiple Patterns

→ | , is the delimiter of multiple patterns

→ Eg:- $grep -E 'sengupta|dasgupta' emp.lst

---| barun sengupta |

---| s.n. dasgupta |

④ $grep -E '(sen|das)gupta' emp.lst

Same o/p.

Shell programming :-

Shell script :-

→ group of command to be executed regularly, should be stored in a file and the file is executed as a shell script @ shell program

→ .sh extension is not mandatory.

→ Eg:-

```
#!/bin/sh
#script.sh
echo "Today's date: `date`"
echo "My shell: $SHELL"
```

Here,

* #!/bin/sh is called interpreter line, which starts with #! followed by pathname of the shell to be used for running the script.

*. To execute the file script.sh, make it executable

```
$ chmod +x script.sh
$ script.sh
Today's date: Tue OCT 27 10:02:42 IS
My shell: /bin/sh
```

ordinary and Environment variables :-

* A variable assignment is of the form variable = value, but its evaluation requires the $ as prefix to the variable name.

Eg:- $count = 5

① $ echo $count

5.

② $total = $count

$echo $total

5.

* When shell reads the command line, it interprets any word preceded by a $ as a variable and replace the word by the value of the variable.

* Variables names begin with a letter but can contain numerals and _ as the other characters, names are case-sensitive.

* Shell variables by default is string type.

* All shell variables are initialized to null strings by default. Explicit assignment as possible.

i.e X = " " (or) x = ' ' (or) x =

* A variable can be removed with unset & protected from reassignment by readonly.

unset X — X is now undefined

readonly X — X can't be reassigned.

Environment variables:-

→ Dynamic named value that can affect the way running processes will behave on a computer.

→ Give information about the system behaviour.

→ UNIX environment is defined by environment variables.

→ Shell undergoes initialization when user login to system.

① /etc/profile :- maintained by System admin contains the Shell initialization inform

② .profile file controlled by User.

- the type of terminal you use.

→ list of dir in

→ List of variables affecting look & feel of terminal

→ List of some env variables are

HOME, PATH, LANG, TERM, RANDOM.

• Profile file :-

* . Start-up file of an UNIX user,
* . It is a shell script that is executed by the shell when user logs in to the system.
* . It contains commands that are meant to be executed only once in a session.
* . It contains some of variables that has to be assigned in this script.
* . profile lets us to customize our operating environment to suit our requirements.
* . Every time we make changes in this file, we should either log out & log in again (or) use a special command (called dot) to execute the file

. . profile

* . profile should be located in home directory and it is executed after /etc/profile

Shell Programming :-

→ The shell has a whole set of internal-commands that can be combined together as a language.

→ Shell programs runs in interpretive mode, not compiled.

→ Basically used for system administration.

Shell Scripts :-

→ group of commands that have to be executed regularly are stored in a file. This file is executed as a shell script @ shell program

→ .sh extension is not mandatory

→ vi editor could be used to create the shell script.

Eg:- script.sh.

```
#!/bin/sh                          → interpreter
# script.sh : Sample shell script  → comment
echo "Today's date: `date`"
echo "This month's calendar:"
cal `date "+%m 20%y"`
echo "My shell : $SHELL"
```

#!/bin/sh :- interpreter line starts with #! followed by the pathname of the shell to be used for running the script.

Command line Arguments :

→ Shell scripts also accept arguments from the command line.

→ When arguments are specified with a shell script, they are assigned to certain special "variables" called positional parameters.

→ The first argument is read by the shell into the parameter $1, the second argument into $2 and so on.

→ $* → It stores the complete set of positional parameters as a single string.

→ $# → It is set to the number of arguments specified. This lets you design scripts that check whether the right number of arguments have been entered.

→ $0 → Holds the command name itself. You can link a shell script to be invoked by more than one name. $0 behave differently depending on the name by which it is invoked.

```
#!/bin/sh
#emp2.sh : uses command line arguments
echo "program : $0          #$0 contains the program name.
The number of arguments specified is $#
The arguments are $*"
grep "$1" $2
echo "\n Job over"
```

```
$emp2.sh director emp.lst
program : emp2.sh
The number of arguments specified is 2
The arguments are director emp.lst
1006 | chanchal singvi | director | sales | 03/09/38 | 67
6521 | lalit chowdury | director | marketing | 26/09/45/82
Job over.
```

Here $0 is assigned to first argument of command itself.

$1 assigned to second word

$2 assigned to third word

Exit and EXIT Status of command.

→ Shell script & C program have lot in common, both use the same command/function to terminate a program.

ie exit in the shell &
exit() in C.

→ exit command usually run with a numeric argument:

exit 0 — Used when everything went fine
exit 1. — Used when something went wrong.

→ We don't need to place this command at the end of every shell script. Bec2 shell understands when script execution is complete.

→ Eg:- grep command if it fails to locate a pattern, then we say command has failed i.e exit function in the grep code has invoked with a non-zero argument (exit(1)). This value 1 is communicated to the calling program usually a shell.

Smitha. N
Asst. Prof
CSE, CMRIT

Evaluate a command's exit status :-

→ The shell offers a variable ($?) and a command (test) that evaluates a command's exit status.

→ The Parameter $? :- Stores the exit status of the last command.

* It has value 0 if the command succeeds
* non-zero value if command fails.
* This parameter is set by exit's argument.
* If no exit status is specified, then $? is set to zero (true).

Eg:- $ grep director emp.lst > /dev/null ; echo $?
0 success

$ grep manager emp.lst > /dev/null ; echo $?
1 failure - in finding patte

* The exit status is extremely important for programmers. They use it to devise program logic that branches into different paths depending on success or failure of a command.

Logical operators for conditional execution:-

→ The shell provides two operators that allow conditional execution.

ie the && and ||

Eg:- 1) cmd1 && cmd2

2) cmd1 || cmd2.

here 1) cmd2 is executed only when cmd1 is success.

Eg:- grep 'director' emp.lst && echo "pattern found in file"

o/p:-

1006 | chanchal singhvi | director | sales | 03/09/38 | 6700

6521 | lalit chowdury | director | marketing | 26/09/45 | 8200

pattern found in file.

→ The || operator plays an inverse role, second executed only when the first fails.

Eg:- grep 'manger' emp.lst || echo "Pattern not found"

pattern not found

here if pattern not found, we display pattern not found

Eg:- grep "$1" $2 || exit 2

echo "pattern found - Job over".

→ if pattern nt found, no point in continuing, so program is aborted.

→ else if pattern found, then this message is printed.

The if conditional :-

→ The if statement makes two-way decisions depending on the fulfillment of a certain condition.

→ Syntax:-

```
if command is succesful
then
    execute commands
else
    execute commands.
fi
```

```
if command is succesful.
then
    execute commands
fi.
```

```
if command is succesful
then
    execute commands
elif command is succesfull
  then . . .
. else . . .
```

Eg:-

```
#! /bin/sh
#emp3.sh : using if and else
if grep "^$1" /etc/passwd 2> /dev/null
then
    echo "pattern found - Job over"
else
    echo "pattern not found"
fi.
```

O/p: $ emp3.sh ftp

ftp:* 325:15:FTP User:/users1/home/ftp:/bin/false

pattern found - Job over

$ emp3.sh mail

pattern not found.

While : Looping :-

→ Using Loop we can perform execution of instruction repeatedly.

→ The shell features three types of loops:

* while
* until &
* for.

→ While statement repeatedly performs a set of instructions until the control command returns a true exit status.

Syntax:-

```
while condition is true
do
    commands
done
```

→

```
#!/bin/sh
answer=y
while [ "$answer" = "y" ]
do
    echo "Enter the code & description: \c" >/dev/tty
    read code description
    echo "$code | $description" >> newlist
    echo "Enter any more (Y/N)? \c" > /dev/tty
    read anymore
```

```
case $anymore in
    y* | Y* ) answer=y ;; #Also accepts yes, YES etc.
    n* | N* ) answer=n ;; # Also accepts no, NO etc.
    *) answer=y ;; #Any other reply means y
esac
done.
```

O/P

```
$ cmp5.sh.
Enter the code and description: 03 analgesics
Enter any more (y/n) ? y
Enter the code and description: 04 antibiotics.
Enter any more (y/n) ? [Enter]
Enter the code and description: 05 OTC drugs
Enter any more (y/n) ? n

$ cat newlist
03 | analgesics
04 | antibiotics
05 | OTC drugs
```

Using while to wait for a file :-

→ Application of while loop.

→ Program needs to read a file that is created by another program, but it has to wait until the file is created.

→

```
#!/bin/sh
while [ ! -r invoice.lst ]      #while invoice.lst can't
do                               read
        sleep 60                 #sleep for 60 second
done
alloc.pl                         #Execute this program a
                                 exiting loop.
```

filename : monitfile.sh.

→ Here monitfile.sh periodically monitors the for the existence of the file, and then exec the program once the file has been located sleep command makes the script pause for the duration (in seconds) as specified in its argumen

→ The loop executes repeatedly as long as the file invoice.lst can't be read (!-r mea not readable), if file becomes readable, th loop terminates & alloc.pl executes.

Setting up an Infinite loop :-

→ As System administrator, if we want to see the free space available on our disk every five minutes. We need an infinite loop, implemented by using true as a dummy control command with while.

```
while true ; do
do    df -t
      sleep 300
done &
```

df reports free disk space.

& after done runs loop in background.

Until :-

→ Operates with a reverse logic used in while.

→ loop body executed as long as the condition remains false.

setting up an Infinite loop :-

→ As System administrator, if we want to see the free space available on our disk every five minutes. We need an infinite loop, implemented by using true as a dummy control command with while.

```
while true ; do
do    df -t
      sleep 300
done &
```

#df reports free disk space.

#& after done runs loop in background

Until :-

→ Operates with a reverse logic used in while.

→ loop body executed as long as the condition remains false.

for : Looping with A List :-

→ for doesn't test a condition, instead uses a list instead.

Syntax:-

```
for variable in list
do
        Commands                    Loop body.
done
```

here,

* Each white-space separated word in list is assigned to variable in turn, and commands are executed until list is exhausted.

→ Eg:- $for file in chap20 chap21 chap22 chap23 ; do
> cp $file ${file}.bak
> echo $file copied to $file.bak
> done

chap20 copied to chap20.bak
chap21 copied to chap21.bak
chap22 copied to chap22.bak
chap23 copied to chap23.bak .

Possible sources of the List :-

→ List can consist of practically any of the expressions that the shell understands & processes

① List from Variables :-

→ We can use series of variables in the command line. The shell execute it before loop.

→ Eg :- for var in $PATH $HOME $MAIL

do

echo "$var"

done.

O/P :-

/bin : /usr/bin : /home/local/bin : /usr/bin : /X11 : : /oracle

/home/henry

/var/mail/henry.

② List from command Substitution :-

→ command substitution used to create the list.

→ This is most suitable if list is large

→ Clean arrangement ∴ we can change list without having to change the script.

Eg :- for file in 'cat dist'

Here for command line pick up its list from the file dist.

③ List from Wildcards :-

→ If list consists of wildcards, the shell interprets them as filenames.

Eg:-

```
for file in *.htm *.html ; do
    sed 's/strong/STRONG/g
        s/img src/IMG SRC/g' $file > $$
    mv $$ $file
    gzip $file
done.
```

Here,

→ Each html filename is assigned to the variable file in turn.

→ Sed performs some substitution on each file & writes the output to a temporary file.

→ This filename is numeric - expanded from the variable $$ (the PID of the current shell).

→ The temporary file is written back to the original file with mv.

→ And finally file is compressed with gzip.

(4). List from Positional Parameters :-

→ for is also used to process positional parameters that are assigned from command-line arguments.

Eg:-
```
#!/bin/sh
for pattern in "$@" ; do
    grep "$pattern" emp.lst || echo "Pattern $pattern not found"
done.
```

Here,

* Scans a file repeatedly for each argument.
* Shell parameter "$@" (not $*) to represent all the command line arguments.

O/P :-
```
$emp6.sh 2345 1265 "jai sharma" 4379
2345|j.b. saxena|g.m|marketing|12/03/45|8000
1265|s.n. dasgupta|manager|sales|12/01/63|5800
9876|jai sharma|director|production|12/03/50|7000
Pattern 4379 not found.
```

→ for patten in "$@"

for pattern "$@" is implied.

Here a blank list defaults to "$@" parameter.

i.e why we can use $* instead of $@.

basename : Changing Filename Extensions :-

→ basename is external command.

→ used in changing the extensions of a group of filenames.

→ It extracts the "base" filename from an absolute pathname

$ basename /home/henry/project3/dec2bin.pl
dec2bin.pl

→ When used with 2 arguments, it strips off the second argument from the second argument.

$ basename ux2nd.txt .txt
ux2nd.

#.txt stripped

→ Eg:- for file in *.txt ; do
leftname = `basename $file .txt`
mv $file ${leftname}.doc
done.

→ for pickup second.txt as the first file

→ leftname stores seconds. (with a dot)

→ mv simply adds a doc to the extracted string (seconds.)

The case conditional :-

→ case statement is conditional offered by the shell.

→ matches an expression for more than one alternative

→ multi-way braching.

→ Syntax :-

```
case expression in
    pattern1) commands1 ;;
    pattern2) commands2 ;;
    pattern3) commands3 ;;
    . . .
esac
```

Here, case matches expression with pattern1.
if match succeeds, then executes commands1.
if fails, then Pattern2 is matched & so forth.
Each command list is terminated with a pair of semicolons. Entire construct is closed with esac.

→ Eg1- Script accepts from 1 to 5, and perform some action

→ case handle numbers but only by treating them as string.

```
#! /bin/sh
echo "      Menu\n
1. List of file \n  2. Process of user \n 3. Today's Date
4. Users of System \n  5. Quit to Unix \n Enter your option:"
read choice
case "$choice" in
    1) ls -l ;;
    2) ps -f ;;
    3) date ;;
    4) who ;;
    5) exit ;;
    *) echo "Invalid option"
esac.
```

O/P

```
$ menu.sh
        Menu
1) List of file
2. Procen of user
3. Today's Date
4. Users of Systems
5. Quit to Unix
Enter your option: 3
Tue Jan 7 18:03:26 IST 2020.
```

Matching multiple patterns:-

→ case can also specify the same action for more than one pattern.

→ case use | to delimit multiple patterns.

Eg: echo "Do you wish to continue ? (Y/N):"
read answer
case "$answer" in
Y/y) ;;
n/N) exit ;;
esac

Build-cards: case uses them:

→ uses the filename matching metacharacters *, ?
& character class. but only to match strings
& not files in the current directory.

→ Eg: case "$answer" in
[yY] [eE] *) ;;
[nN] [oO]) exit ;;
*) echo "Invaild response"
esac

Read and Readonly:

Read :-

* Read statement is the shells internal tool for taking input from the user.
  ie making script interactive.

* Input supplied through the standard input is read into these variables.

  Eg:- read variable_name.

* Script pauses at this point to take input from the keyboard, entered content is stored in variable_name since it is a form of assignment, no $ used

  Eg:- #! /bin/sh
  echo "Enter the name"
  read fname.
  echo "Enter name is $fname".

* A single read statement can take one or more variable, let us to enter multiple arguments.

  Eg:- read fname lname.

Readonly :-

→ Shell provides a way to mark variables as read-only by using the readonly command.

→ after marking variable as read-only, we cannot change its value.

Syntax :-

```
readonly var_name
```

→ Trying to change the value @ unsetting variables that are marked read-only will result in an error.

→ Syntax to make a function readonly:

```
readonly -f fun_name
```

→ readonly -p

displays all read only variables.

→ readonly -f

displays all read only functions.

The test command and its shortcut :-

* if statements can't directly handle, test statements.
  ∴ test command is used.

* test command uses certain operators to evaluate the condition on its right and returns either a true (0) false exit status, which is then used by if for making decisions.

* test works in three ways:
  → Compares two numbers.
  → Compares two strings (or) a single one for a nu
  → Checks a file's attributes.

* Numeric comparison :-
  → The numerical comparison operators used by test ha a form different from what you would h seen anywhere.

  Eg 1- $x=5 ; y=7 ; z=7.2

  $ test $x -eq $y ; echo $?

  1

  Noteq

  $ test $x -lt $y ; echo $?

  0

  True

Numerical comparison operators used by test.

Operator	Meaning.
-eq	Equal to
-ne	Not Equal to
-gt	Greater than
-ge	Greater than @ equal to
-lt	less than
-le	less than @ equal to.

Ex1- #!/bin/sh.

```
if test $# -eq 0 ; then
    echo " Usage : $0 pattern file " >/dev/tty
elif test $# -eq 2 ; then
    grep "$1" $2 || echo " not found" >/dev/
else
    echo " you didnt enter 2 arguments" >/de
fi.
```

O/P

① emp3a.sh >foo

② emp3a.sh ## >foo.

③ emp3a.sh henry /etc/passwd

Shorthand for test :-

→ A pair of square brackets enclosing the expressions can replace it.

Eg:- test $x -eq $y

[$x -eq $y]

String Comparison :-

*. test used to compare string

* Test	True if
→ $s_1 = s_2$	String $s_1 = s_2$
→ $s_1 != s_2$	String s_1 is not equal to s_2
→ -n stg	String stg is not a null str
→ -z stg	String stg is a null string.
→ stg	String stg is assigned & not n
→ $s_1 == s_2$	String $s_1 = s_2$ (korn & Bash

Eg:- #!/bin/sh

if [$# -eq 0] ; then

echo "Enter the string to be searched :\c"

read pname.

if [-z "$pname"] ; then

echo "you have not entered the string"

fi

```
echo " Enter the filename to be used: "
read fname
if [ ! -n "$fname" ] ; then
   echo " you have not entered the filename"
fi
emp3.sh "$pname" $fname
else
   emp3.sh $*
fi.
```

File tests :-

→ test can be used to test the various file attributes like its type (file, dir @ symbolic link) @ its permissions (read, write, execute, SUID) etc.

→ test	To test if file
-f file	file exists & is a regular file
-r file	file exists and is readable
-w file	file exists & is writable
-x file	file exists & is executable.
-d file	file exists & is a directory.
-s file	file " —— size greater than zero.

Eg:-

```
#!/bin/sh
if [ ! -e $1 ]; then
    echo "file does not exist"
elif [ ! -r $1 ] ; then
    echo "file is not readable"
elif [ ! -w $1 ]; then
    echo "file is not writable"
else
    echo "file is both readable & writable"
fi.
```

The set and shift commands and handling positron parameters :-

→ The shell has an internal-command called 'set' which shall assigns its arguments to positional parameters $1, $2 and so on.

→ This feature is especially useful for picking up individual fields from the output of a program.

Eg:- $set 9876 2345 6213.

This assigns the value 9876 to positional parameter $1, $2 to 2345 & 6213 to $3 and $# no of arguments

$* Arguments as single string.

$echo "\$1 is $1, \$2 is $2, \$3 is $3"
$1 is 9876, $2 is 2345, $3 is 6213.

$echo "The arguments are $*"
The arguments are 9876 2345 6213.

1-(2)
$set `date`
$echo $*
Wed Jan 8 09:40:35 IST 2003
$echo "The date today is $2 $3, $6"
The date today is Jan 8, 2003.

shift :- Shifting Arguments left :-

* shift statement transfers the contents of a positional parameter to its immediate lower numbered one.
this is done as many times as the statement is called. When called once, $2 becomes $1, $3 becomes $2, and so on.

Eg:-

```
$echo "$@"
Wed Jan 8 09:48:44 IST 2003
$ echo $1 $2 $3
Wed Jan 8
$ shift
$ echo $1 $2 $3
Jan 8 09:48:44
$ shift 2
$ echo $1 $2 $3
09:48:44 IST 2003
```

shift 2 places

Eg:- #!/bin/sh

```
case $# in
0|1) echo "usage: $0 file pattern(s)"; exit 2;;
*) fname=$1 #stores $1 as a variable before it gets lost
shift
for pattern in "$@"; do # starts Iterating with $2
grep "$pattern" $fname ||
echo "pattern $pattern not found"
```

fname stores emp.lst and loop iterates with the three strings: sexena, 1006, 9876.

shift :- Shifting Arguments left :-

* shift statement transfers the contents of a positional parameter to its immediate lower numbered one.
* this is done as many times as the statement is called. When called once, $2 becomes $1, $3 becomes $2, and so on.

Eg:-

```
$echo "$@"
Wed Jan 8 09:48:44 IST 2003
$ echo $1 $2 $3
Wed Jan 8
$ shift
$ echo $1 $2 $3
Jan 8 09:48:44
$ shift 2
$ echo $1 $2 $3
09:48:44 IST 2003
```

shift 2 places.

Eg:- #!/bin/sh

```
case $# in
0|1) echo "usage: $0 file pattern(s)"; exit 2;;
*) fname=$1 #Stores $1 as a variable before it gets lost
shift
for pattern in "$@"; do # starts Iteration with $2
grep "$pattern" $fname ||
echo "pattern $pattern not found"
done;;
```

→ fname stores emp.lst and loop iterates with the three strings: saxena, 1006, 9876.

The Here Document (<<)

→ The shell uses the << symbols to read data from the same file containing the script. This is referred to as a here document, signifying that the data is here rather than in a separate file.

Eg:- mail x sharma << MARK
your program for printing the invoices has been executed on 'date'. Check the print queue
The update file is known as $filename
MARK.

→ The here document symbol (<<) is followed by three lines of data & a delimiter (the string MARK).

→ The shell treats every line following the command and delimited by MARK as input to the command.

→ Sharma at the other end will see the three lines of message text with the date inserted by command substitution and the evaluated filename. The word MARK itself doesn't show.

Using the Here Document with Interactive Programs:-

* We can make the script work non-interactively by supplying the inputs through a here document

```
$emp1.sh << END
>director
>emp.lst
>END.
```

Enter the pattern to be Searched : Enter file to be
Searching for director from file. emp.lst

```
9876 | jai sharma | director | production | 12|03|50
2365 | barun sengupta | director | personnel | 11|05|47
```

trap: Interrupting a Program:-

→ trap statement lets us do the things that we want in case the script receives a signal. The statement is normally placed at the beginning of a shell script & uses two lists:

```
trap 'command_list' signal_list.
```

→ When script is sent any of the signal in signal_list, trap executes the commands in command_list.

→ Eg:- `trap 'rm $$*; echo "Program interrupted" exit' HUP INT TERM.`

here,
- trap is a signal handler.
- It first removes all files expanded from *
- echoes a message and finally terminates the script when the signals SIGHUP(1), SIGINT(2) & SIGTERM(15) are sent to the shell process running the script.

Eg :- (2)

```
trap '' 1 2 15
```

To ignore the signal & continue processing. have to use null command list.

Shell Script Example :- ref document which I have shared.

Smitha.N.
CSE, Asst. Prof.

Module-3 UNIX File APIs:-

Introduction :-

UNIX @ POSIX system, the following functions on any type of files could be performed. Create files, open files, Transfer data to & from files close files, remove files, query file attributes, change file attributes, Truncate file.

General File APIs:-

In UNIX @ POSIX System, file may be of any of the following types:

* Regular file
* Directory file.
* FIFO file
* Character device file
* Block device file
* Symbolic link file.

There are several APIs used to manipulate one/more type of file.

Open :-

* The open function establishes a connection between a process and a file.
* It can be used to create a brand new file.

* After a file is created any process can call the open function to get a file descriptor to refer to the file.

* The prototype of the open function is:

```
#include <sys/types.h>
#include <fcntl.h>
int open (const char *path_name, int access_mode,
          mode_t permission);
```

* Here,
  → first argument path_name is the path-name of a file. It could be absolute / relative path name.
  → second argument access_mode is an integer value that specifies how the file is to be accessed by the calling process.

Access mode flag	Use
1) O_RDONLY	Opens the file for read-only.
2) O_WRONLY	Opens the file for write-only
3) O_RDWR	opens the file for read and write.

One / more of the following modifier flags can be specified by bitwise-ORing them with one of the above access-mode flag to alter the access mechanism of the file:

Access modifier flag	Use
(1) O_APPEND	Appends data to the end of the file
(2) O_CREAT	creates the file if it does not exist. Used with the O_CREAT flag only.
(3) O_EXCL	This flag causes open to fail if the named file already exists.
(4) O_TRUNC	If the file exists, discards the file content & sets the filesize to zero bytes.
(5) O_NONBLOCK	Specifies that any subsequent read (or) write on the file should be nonblocking.
(6) O_NOCTTY	Specifies not to use the named terminal device file as the calling process control terminal.

Eg:- int fdesc = open ("/usr/xyz/textbook", O_RDWR|O_APPEND);

*. If a file is opened for write-only (or) read-write any modifier flags can be specified. However, O_APPEND, O_TRUNC, O_CREAT & O_EXCL are applicable to regular files, O_NONBLOCK is for FIFO & device file only, O_NOCTTY is for terminal

* The O_APPEND flag specifies that data written to a named file will be appended at the end of the file. If this is not specified, data can be written to anywhere in the file.

* The O_TRUNC flag specifies that if a named file exists already, the open function should discard its content. If its not specified, current data in the file will not be altered by the open function.

* The O_CREAT flag specifies
  → If named file does not exist, open function should create it.
  → If doesn't exist, the O_CREAT flag has no effect on the open function.
  → If the named file does not exist & the O_CREAT file is not specified, open will abort with a failure return status.

* The O_EXCL flag should be used with O_CREAT flag. The open function will fail if the named file exists. O_EXCL used to ensure that the open call creates a new file.

* The O_NONBLOCK flag specifies that if the open & any subsequent read/write function calls on a named file will block a calling process, the kernel should abort the functions immediately & return to the process with a