UNIT - II

Shell Responsibilities

SHELL Programming with Bourne again Shell (Bash)

1. Shell Responsibilities :-

A shell is the only medium between the users and operating system to run the program.

1. Built-in commands :-

A shell contains several preset commands.

Ex:- $ ls ↵

The shell searcher "/bin" directory for the executable program called "ls". If it founds it executes the commands

"ls" command displays the list of files and directories in the current directory.

2. Redirection :-

The shell is associated with 3 files with the terminal. They perform all terminal-related activity with the 3 files that the shell makes available to every command.

1. Standard Input (0) :- The file (or stream) representing input, which is connected to the keyboard
2. Standard Output (1) :- The file (or stream) representing output, which is connected to the display
3. Standard Error (2) :- The file (or stream) representing error messages that emanate from the command or shell. This is also connected to the display.

Standard Input :- This file can represent three i/p sources
1. The keyboard, the default source.
2. file using redirection with < symbol (ametacharacter)
3. Another using a pipeline.

Ex: 1 wc ↵
Standard input can be redirected
It can come from a file
or a pipe line
[ctrl-d]
3 14 17 → output

Ex 2:- wc < sample.txt
3 14 71 → output

here On seeing <, the shell opens the disk file, sample.txt, wc reads that data from sample.txt.

Ex 3: cat sample.txt | wc ↵
3 14 71 → output
pipe symbol.

here cat command output will be transferred to wc then it counts the lines, words, characters.

Standard output :-
There are 3 possible destinations of this stream
1. The terminal (the default destination)
2. The file using the redirection symbol > and >>
3. As an input another program using pipe line.

Ex:- 1 $ wc sample.txt > newfile

```
$ cat newfile ↵
3 14 71 sample.txt
```
→ Output.

here wc command sends the output to the newfile. So nothing appear on the terminal. If the newfile not exists then create that file. If it exists then it overwrite the previous datas.

Ex:2 :- $ wc sample.txt >> newfile.

```
$ cat newfile ↵
  Abcdef.
  3  14  17  $
```

output

e.g. > → overwrite
>> → appends

here the out put of the wc command will be appended to the new file. The Abcdef ... data already there in the newfile. The count will be added to this file.

3. Standard Error :-

The standard Input, output, error file is represented by a number called a file descriptor.

0 - Standard Input
1 - Standard Output
2 - Standard Error.

Ex 1: cat foo ↵

```
cat: cannot open foo
```

error → output

Ex 2:- $ cat foo > errorfile.

```
$ cat errorfile
cat: cannot open a file.
```

→ output

Ex 3:- $ cat foo 2>> error file.

↳ the error will be appended in error file.

PIPES :- Standard Input and standard output Constitute 2 seperate streams that can be Individually manipulated by the Shell.

The shell connects standard Input and standard output using a special operator, the | (Pipe)

* for Example if want to count no. of users under that shell Then we require wc and who.

1) First we have to copy the list of users to another file i.e

```
$ who > user.txt
```

2) second we can count the users list by using

```
$ wc -l < user.txt
5 → counts the number of lines
```

* The above Example can be replaced with Pipe operator.

```
$ who | wc -l
5 → output
```

→ here who transferes the output to the wc command
→ wc consider the who command as its input then it displays the no. of lines count.

Command substitution :- The shell enables one or more Command arguments to be obtained from the standard output of another command. This feature is called Command substitution.

Consider the Example If I want to print todays date with a statement like this :-

The date today is Sat Sep 7 19:01:16 IST 2016

To get this output use expression 'date' as an argument to echo.

Exp:- $ echo The date today is 'date'

The date today is sat sep 7 19:01:16 IST 2016.

Output

Ex2:-

$ echo 'There are `ls | wc -l` files in the directory'

There are 56 files in the current directory

Output.

* Command substitution has intersting application possibilities in shell script. It speed up work by letting you combine a number of instructions is one.
* Command substitution is enabled when backquotes are used with double quotes. If you use singlequotes, it won't work.

SHELL Variable :- The shell supports variables that are use-ful both in the command and shell scripts

A variable assignment is of the form

Variable = Value.

but its evaluation requires the $ as prefix to the variable name:

Ex:- $ count = 5

$ echo $count

-) There must not be any whitespace on either side of the = symbol.

→ Variable names begin with a letter. but can contain numerals and the _ as the other characters.
→ Names are case sensitive ; x and X are 2 different variable
→ Shell variables are not typed ; you don't need to use a char, int etc when you define them.
→ All shell variable are of the string type.
→ All the shell variables are initialized to null strings by default.
→ A variable can be removed with **unset** and protected from reassignment by **readonly**. Both are shell internal commands.

Ex:- unset x
readonly x ;

Shell variables are of 2 types :-

1. local variables
2. Environment variable.

local variables :- are more restricted in scope :

Ex:- $ DOWNLOAD_DIR = /home/kumar/download

here DOWNLOAD_DIR is a local variable ; its value is not available to child processes.

Ex:- i) $ DOWNLOAD_DIR = /home/kumar/download.
2) $ echo $DOWNLOAD_DIR
/home/kumar/DOWNLOAD → output
3) $ sh → it creates a child.
4) $echo $DOWNLOAD_DIR
$ - → it won't display because it won't be visible to child.
5) $ exit → terminate child.
6) $ -

→ **Set** :- The statement displays all variable in the current Shell

→ **Env** :- command displays only environment variables.

The common Environment Variables :- These variable are visible in user's total Environment.

list of Environment Variables.

1) PATH :- The PATH variable instructs the shell about route it should follow to locate any executable command.

2) HOME :- When you log in, LINUX normally places you in a directory named after your login name. This directory is called the home or login directory:

```
$ echo $HOME
/home/henry.
```

3) Mailbox Location and checking (MAIL and MAILCHECK) :-

4) Mail :- Absolute path name of current Directory

5) PS1 :- Primary prompt string

6) PS2 :- Secondary prompt string.

7) SHELL :- User's login shell and one invoked by programs having shell escapes.

Aliases :- Bash and Korn support the use of aliases that let you assign shorthand names to frequently used commands.

Example:- The most frequently used ls -l command we can change the name as 'll'

$ alias 'll = 'ls -l' ↵

→ Here multiple words we should keep in single Quoles

→ After execution of above command we can use ll instead of ls -l.

→ An alias is recursive. which means that
if a is aliased to b
b is aliased to c
Now a should run c

→ we can unset the alias using unalias command

Command Substitution :-

The shell enables the connecting of 2 commands in yet another way.

1) A pipe enables a command to obtained its standard input from the standard output of another command

2) The shell enables one or more command arguments to be obtained from the standard out of another command.

This 2nd feature is called command substitution

Ex:- You need to display the date with a statement then

① echo The date today is `date`

output → The date today is Thu 19:01:56 IST 2016

② echo "There are `ls|wc -l` files in the directory"

There are 58 files in the directory → output

→ command substitution speeds up work by combining a number of instructions in one.

→ The back quote (`) is one of the characters interpreted by the shell when we placed within double quotes.

→ If you want to echo a literal `, then you have to use single quotes.

Ex3:- echo "There are `ls|wc -l` files in the directory"

(There are `ls|wc -l` files in the directory) → output

→ Ex 2:- Executes the sub command but

Ex 3 - won't executes the sub command.

Quoting :- Single quote (' '), Double Quote (" "), Back quote (` `)

Effects of Quoting (', ", \) and Escaping (\)

→ To assign a multiword string to a variable, you can use any of these mentioned below.

1) escape each space character
2) ~~You~~ using of single ~~code~~ or double quote is preferred solution.

Ex:- 1) $ Message = You \didn \' t \ enter \ the \ filename.
2) $ Message = "you didn't enter the filename."

→ If you have a special character ($) you dont want to evaluate the $ then we can use

1) escape $ symbol.
2) using of single quote

Ex: 1) $ echo the average pay is \$1000
2) $ echo 'the average pay is $1000'

→ like the backquote the $ is also evaluated by the shell when it is double-quoted.

Ex:- 1) $ echo "the PATH is $PATH and the current directory is `pwd`"

The PATH is /bin:/usr/bin: and the current directory is /root ← Output

2) $ echo "The average pay is $1000" ↵
↳ (the average pay is 000) → output

→ whenever you should use double quote or single quote depends on whether you want command

substitution and variable evaluation to be enabled or not.

→ Double quotes permit their evaluation but single quotes not don't

Test commands :-

Using test and [ ] to evaluate expressions

When you use it to evaluate expressions, you need the **test** statement because the true or false values returned by expressions

test work in 3 ways :-

1) Compare 2 numbers
2) compare 2 strings or a single one for a null value.
3) Checks a file's attributes.

1) Number Comparision :-

Numerical comparison operators used by test

operator	Meaning
-eq	Equal to
-ne	Not equal to
-gt	Grater than.
-ge	Grater than or equal
-lt	less than
-le	less than or equal to

Examples :-

1. $ x=5 ; y=7 ; z=7.2

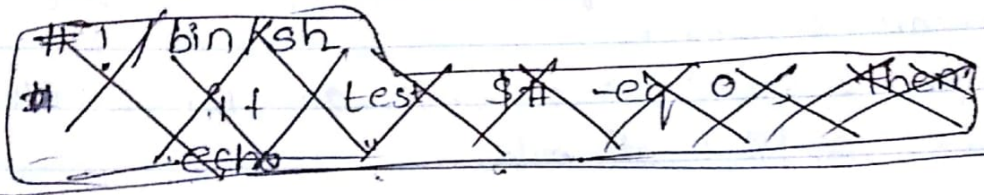
$ test $x -eq $y ; echo $?
(1) → output Not equal.

$ test $x -lt $y ; echo $?
(0) → output True.

$ test $z -gt $y ; echo $?
(1) → output False.

$ test $z -eq $y ; echo $?
(0) → output True.

-) Uses test in an if-elif-else-fi construct:-




```
x = 5
y = 6
if test $x -eq $y then
    echo " Both are Equal"
else
    if test $x -lt $y the
        echo " y is big"
else
    echo " x is big"
```

ex1.sh.

$ sh ex1.sh
y is big.

Shorthand for test :- A pair of rectangular brackets enclosing the expression can replace it.

```
test $x -eq $y
      ⇓
[ $x  -eq  $y ]
```

if (x) if x is greater than 0 it is true.
if x is '0' then it is false.

∴ if [$x] ⟺ if [$x -gt 0]

String Comparision

String Test used by Test

Test	True if
S1 = S2	String S1 = S2
S1 != S2	String S1 is not equal to S2
-n Stg	String stg is ~~assigned~~ ~~and~~ not null
-z Stg	String Stg is a null string
Stg	String stg is assigned and not null.
S1 == S2	String s1 = S2 (Korn and Bash only)

Ex:- if [-n "$pname" -a -n "$fname"]
then
~~emp3~~sh "$fname" "$fname"
echo "$pname and $fname"
else
echo "At least one input was a null"
fi

File Tests :- test can be used to test the various file attributes like its type or its permissions.

File Related Test wit test

Test	True if file
-f file	file exists and is a regular file.
-r file	file exists and it is readable.
-w file	file exists and it is writable.
-x file	file exists and it is executable.
-d file	file exists and is a directory.
-s file	file exists and it has a size grater than zero.
-e file	file exists (Korn and Bash only)
-u file	file exists and has SUID bit set
-k file	file exists and has sticky bit set
-L file	file exists and is a symbolic link (Korn & Bash)
f1 -nt f2	f1 is newer than f2 (Korn & Bash)
f1 -ot f2	f1 is older than f2 (Korn & Bash)
f1 -ef f2	f1 is linked to f2 (Korn & Bash)

Ex :-

```
$ ls -l emp.lst
-rw-rw-rw- 1 kumar group 870 Jun 8 15:52 emp.lst
$ [-f emp.lst ] ; echo $?        → An ordinary file?
  0                                Yes
$ [-x emp.lst ] ; echo $?        → An executable file?
  1                                NO.
```

Interrupt Processing functions:-

trap :- Interrupting a Program.

→ By default shell scripts terminates whenever Interrupt key is passed.

→ It is not good always to terminate shell scripts in this way because that can leave a lot of temporary files on disk.

→ Trap statement lets you do the thing you want in case the script receives a signal.

→ The statement is normally placed at the beginning of a shell script and uses two list:

```
trap 'command - list' signal-list
```

trap is a signal handler.

Ex:-

```
trap 'rm $$*; echo "Program Interrupted"; exit' HUP INT TERM.
```

In the above example trap is a signal handler. Here if first removes all files expanded from $$*,

→ and echos a message and finally terminates the script when the Signals SIGHUP (1), SIGINT (2) (^C), SIGTERM (15) are sent to the shell process running the script

∴ $$* process id of current shell. Process ID Under which they are working

Debugging shell script with set -x :-

→ Set serves as a useful debugging tool with its -x option.

→ When used inside a script, it echos each statement on the terminal, proceeded by a (+) as it is executed.

Example :-

```
$ vi empish
Set -x
   echo "enter search pattern"
   read pname
   echo "enter filename"
   read filename.
   grep "$pname" $filename
   echo " *** THANK YOU *** "
Set +x
```

Output :- $ sh empish

```
+ echo 'enter search pattern'
  enter search pattern
+ read pname.
  devi
+ echo 'enter filename'
  enter file name
+ read file name
  emp.lst
+ grep devi emp.lst
  106 | devi | faculty |IT
+ echo '*** THANK YOU ***'
  *** THANK YOU ***
```

Using command line Arguments

Shell scripts also accepts arguments from the command line When arguments are specified with a shell script, they are assigned to certain special "variables" -

The first argument is read by shell into the perameter $1, 2nd argument into $2, and so on.

→ In addition to these positional perameters there are few other special perameters used by the shell. Thoes are

$* → It store the complete set of positional perameter as a single string

$# → It is set to the number arguments specified.

$0 => Hold the command name itself.

Example

```
echo "program : $0 The number of
arguments specified is $#
The arguments are $*"
```

com.sh.

Output

```
$ com.sh director emp.lst
program : com.sh
The number of arguments specified is 2
The arguments are director Emp.lst
```

→ $0 → program name called com.sh
$1 → 1st argument director
$2 → 2nd argument emp.lst
$# → counting of perameters except $0, is 2
$* → except filenam it gives director & emp.lst

The logical operators && AND || - Conditional Execution

The shell provides & operators

1) &&
2) ||

1) The && delimits two commands: ~~the command~~

Ex:-

cmd1 && cmd2

Here cmd2 is executed only when cmd1 succeeds.

Ex:- grep 'director' emp.ls && echo "pattern found"

```
101 | director | --- | --- | ---
102 | director | --- | --- | ---
pattern found.
```
→ output

2) The || operator plays an inverse role

cmd1 || cmd2

Here The second command is executed only when the first one fails.

Ex:-

grep "manager' emp.lst || echo 'pattern not found"

```
patter not found
```
→ output.

In shell script test ([ ]) also permits the checking of more than one condition in the same line.

1) using -a (AND) operation can be performed
2) using -o (OR) operation can be performed

The if condition:-

The if statement makes two-way decision depending on fulfillment of a certain condition.

Patter 1

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

Pattern 2:-

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

Pattern 3:-

```
if command is successful
then
        execute commands
elif    command is successful
then    ....
else    ---
fi
```

Example programs:-

1) find even or odd number checking (Refer class notes)

The case conditional:-

The case statement is the second conditional offered by the shell.

Syntax:

```
case expression in
    pattern 1) Commands 1 ;;
    pattern 2) commands 2 ;;
    pattern 3) commands 3 ;;
    .....
    pattern x) commands
esac
```

→ x is an optional case. for that no need of ;; symbols.

→ case first matches the pattern 1, If match succeeds, then it executes Commands 1, If fails then pattern 2 is matched and so forth.

→ The entire construct is closed with esac.

Example:- refer class notes

Shell Program to perform arithmatic operations

```
let a=5+4
echo $a
let "a=5+4"
echo $a
```

```
let "a=$1+$2"
z=$((x+y))
```

even or odd :-

```
echo "Enter number"
read n
rem = $(($n % 2))
if [ $rem -eq 0 ]
then
    echo "$n is even number"
else
    echo "$n is odd number"
fi
```