UNIT-III
USING THE SHELL

Using the Shell-Command Line Structure-Met characters-Creating New Commands-Command Arguments and Parameters-Program Output as Arguments-Shell Variables- -More on I/O Redirection-Looping in Shell Programs.

3.1 Command line structure

The simplest command is a single word , usually naming a file for execution

```
$ who                                                    Execute the file /bin/who
you tty2 Sep 28 07:51
jpl tty4 Sep 28 08:32
$
```

A command usually ends with a newline, but a semicolon **;** is also a command terminator :

```
$ date;
Wed Sep 28 09:07:15 EDT 1983
$ date; who
Wed Sep 28 09:07:23 EDT 1983
you tty2 Sep 28 07:51
jpl tty4 Sep 28 08:32
$
```

Although semicolons can be used to terminate commands, as usual nothinghappens until you type RETURN. Notice that the shell only prints one prompt after multiple commands, but except for the prompt,

```
$ date; who
```

is identical to typing the two commands on different lines. In particular, **who**doesn't run until **date** has finished.Try sending the output of "**date ; who**" through a pipe:

```
$ date; who|wc
Wed Sep 28 09:08:48 EDT 1983
2 10 60
$
```

Connecting **who** and **wc** with a pipe forms a single command, called a pipeline , that runs after **date**. The precedence of | is higher than that of **;** as the shell parses your command line.

Parentheses can be used to group commands:

```
$ (date; who)
Wed Sep 28 09:11:09 EDT 1983
you tty2 Sep 28 07:51
jpl tty4 Sep 28 08:32

$ (date; who)|wc
3 16 89
$
```

The outputs of **date** and **who** are concatenated into a single stream that can besent down a pipe.

Data flowing through a pipe can be tapped and placed in a file (but not another pipe) with the t**ee** command,. One use is to save intermediate outputin a file:

```
$ (date; who)|tee save | wc
3 16 89                                                  Output from wc
$ cat save
Wed Sep 28 09:13:22 EDT 1983
you tty2 Sep 28 07:51
jpl tty4 Sep 28 08:32
$ wc<save
3 16 89
$
```

tee copies its input to the named file or files, as well as to its output, so **wc** receives the same data as if **tee** weren't in the pipeline.

Another command terminator is the ampersand **&**. It's exactly like thesemicolon or newline, except that it tells the shell not to wait for the command to complete. Typically, **&**. is used to run a long-running command "in thebackground" while you continue to type interactive commands:

$ long-running-command&.
5273 *Process-id of long-running-command*
$ *Prompt appears immediately*

Given the ability to group commands, there are some more interesting uses ofbackground processes. The command sleep waits the specified number ofseconds before exiting:

```
$ sleep 5
$                                          Five seconds pass before prompt
$ (sleep 5;date) & date
5278
Wed Sep 28 09:18:20 EDT 1983               Output from second date
$ Wed Sep 28 09:18:25 EDT 1983             Prompt appears, then date 5 sec. later
```

The background process starts but immediately sleeps; meanwhile, the second **date** command prints the current time and the shell prompts for a new command. Five seconds later, the `sleep` exits and the first `date` prints the newtime.

```
$ (sleep 300; echo Tea is ready) &          Tea is ready in 5 minutes
5291
$
```

since the precedence of **&** is higher than that of **;**

The &. terminator applies to commands, and since pipelines are commands you don't need parentheses to run pipelines in the background:

```
$ pr file | lpr&
```

arranges to print the file on the line printer without making you wait for thecommand to finish. Parenthesizing the pipeline has the same effect

```
$ (pr file |lpr) &                          Same as last example
```

Most programs accept arguments on the command line, such as file (anargument to pr) in the above example. Arguments are words, separated byblanks and tabs, that typically name files to be processed by the command, but they are strings that may be interpreted any way the program sees fit. Forexample, pr accepts names of files to print, echo echoes its arguments without interpretation, and grep's first argument specifies a text pattern to search for.

The various special characters interpreted by the shell, such as <, >, !, ;

and&, are not arguments to the programs the shell runs. They instead controlhow the shell runs them. For example,

```
$ echo Hello >junk
```

tells the shell to run **echo** with the single argument **Hello**, and place the outputin the file **junk**. The string **> junk** is not an argument to **echo**; it isinterpreted by the shell and never seen by echo. In fact, it need not be thelast string in the command:

```
$ > junk echo Hello
```

3.2 Metacharacters

The shell has a big family of meta characters to which special significance has been given.

All meta characters can be classified as shown ;

a. File name substitution: **? * […] [!...]**
b. I/O redirection : **> < >> << m> m>&n**
c. Process execution : **; () & && ||**
d. Quoting characters : **' ` " " **
e. Positional parameters: **$1 $2 $3 …$9**
f. Special characters: **$0 $* $@ $# #! $$ $_**

a.**Pattern matching or wild cards for File name substitution**: **? * […] [!...]**

wild card	significance
*	Matches any number of characters including none.
?	Matches a single character
[characters]	Matches one occurrence of any of the given characters
[abc]	Matches a single character either a,b,or c
[!characters]	Matches all characters except the set in class.
[!abc]	Matches a single character i.e not a ,b,c

Examples:

1. * Matches any number of characters including none.

$echo * displays all files in your current working directory

$ls exp* It displays all files, that are starting with **exp(exp,exp1, expab,exp.c …)**

$mv * ../d1 moves all files from current working directory to **d1** directory

1. ? Matches a single character.

$ls exp? Lists all files whose name is started with **exp** and after has one haracter.(expa,exp2..)

$ls ch?? Lists all files whose name is started with **ch** and after it has any two characters(ch1a, ch1b..)

$ls ?? Lists all files whose file name has only two characters.(ex,a1,po,…)

3.The character class **[ ]:** can have multiple characters inside this enclosure,but matching takes place for a single character in the class.

$ls exp[34] lists the file **exp3** or **exp 4 but** not **exp34.**

$ls exp[abc] lists the file **expa or expb or expc** but not **expab..**

4.Negating the character class[!]: placing the ! as 1st character of [] reverses the matching criteria. It matches all other characters except the one's in the class.

$[!a-zA-Z]* Matches all filenames where the first character is not alphabetic

$*.[!z] Matches all files having extensions except .z

$ls [!d-m]* Matches all files whose first character is anything other than an alphabet in range **d to m**

b.**I/O redirection meta characters**: **> < >> << 2> 2>&1**

> Input redirection

< output redirection

>> output redirection with append

2> Error redirection

2>&1 both error and output redirection

<< (Input inline redirection or **here document)**: If the character **<<** follows a command in the format **Command** << **word** the shell uses the lines that follows as the standard input for command, until a line that contains just **'word'** is found.

EXAMPLES:

```
$wc -l << END
>here's a line
>and another
>And yet another
>Now you can end
>END
4
```

Here the shell fed every line typed into the standard input of **wc** until it encounters the line containing just **END**. And displayed the no.of lines as output

```
$cat<<foo
>$HOME
>*****
>`date`
>foo
/home/cse501
*****
Sun jan 21 15:23:15 IST 2013
$
```

The shell performs parameter substitution for the redirected input data, executes back-quoted commands and recognizes the back slash character. However any other special characters such as *, " are ignored.

c. Process execution **; () & && ||**

; To run more than one command at same command line, we can run several commands each separated by **;**

```
$date;ls;pwd
Sun jan 21 15:23:15 IST 2013
f1 f2  m  x  d1
/home/cse501
```

1st **date** is executed and displayed its o/p on monitor.
2nd **ls,** next **pwd** is executed.

```
$date;ls;pwd > a.txt
Sun jan 21 15:23:15 IST 2013
f1 f2  m  x  d1
$cat a.txt
/home/cse501
```

date,ls command o/p is dipalyed on terminal. p**wd** command o/p is redirected to a **file a.txt.**

( ) Command can be grouped by placing them between parentheses, which causes them to be executed by a child shell(sub shell). The commands given in a group share the same standard i/p, standard o/p, standard error channels, and the group may be redirected and piped as if it is a single command. The first form ( ) ,causes the commands to be executed by a sub shell,.

```
$(date;ls;pwd) >p.txt
$cat p.txt
Sun jan 21 15:23:15 IST 2013
f1 f2  m  x  d1
/home/cse501
```

The 3 commands o/p is redirected to *p.txt* file

```
$(cd d1; pwd)
/home/cse501/d1
```

& this character causes a process execution in background. If a process takes more time to complete, then we can make that process as background process, and we can proceed with another job in the foreground.
A back ground job is indicated with &
$ sort f1>f2 & /* sort command is executed in back ground*/
&&,|| These characters can be used optionally execute a command depending on the success or failure of the previous command.

&& : **Command1 && Command2** , here if *command1* returns an exit status as zero(true), then only *command2* is executed. if *command1* returns an exit status as nonzero(false), then *command2* gets skipped.

```
$grep rose f1>f2 && echo rose is found
```

in this example if **grep** command find the pattern **rose** in file **f1**, then **echo** command is executed. If the **rose** is not found in **f1,** then the **grep** returns 1(false),**so echo** is not executed.

|| : Command1 || Command2 here if *command1* returns an exit status as zero(true), then *command2* is not executed. if *command1* returns an exit status as nonzero(false), then *command2* is executed.

```
$grep rose f1>f2 || echo rose is not found
```

in this example if **grep** command find the pattern **rose** in file **f1**, then **echo** command is not executed. If the **rose** is not found in **f1,** then the **grep** returns 1(false),**so echo** is executed.

d .Quote characters: The shell recognizes 4 different types of quote characters.
The single quote ‘
The double quote “
The Back slash \
The single quote ‘
1.The meaning of any meta character is lost if they enclosed in single quotes
2. Spaces are preserved in single quotes.

```
$echo one          two          three        four
one two three four
$echo 'one         two          three        four'
one          two          three        four
```

In first **echo** statement the shell removes extra white spaces from line and passes 4 arguments to **echo.**
Ins 2nd **echo** statement because the argument s are enclosed in single quotes all 4 words are considered as single argument and passed to **echo**. So the spaces are preserved when they are in single quote.

```
$a=5
$echo $a
   5
$echo '$a'
   $a
$echo '*'
   *
$echo  '> |; ( ) { } >> " &'
> |; ( ) { } >> " &
```

Generally **echo $a** prints the **a** value. But it is enclosed in single quote it printed **$a** in 2nd **echo** statement.
$echo * prints all files in pwd. But ***** is enclosed in single quote so it lost its meaning and considered as a normal character.
In last echo statement also different special characters are there they all lost their meaning because they were enclosed in single quotes.

```
 $echo 'how are you
 >today John '
 how are you
 today John
```

Even the enter key will be ignored by the shell if it is enclosed in quotes.

The double quote " : double quotes work similar to single quotes, except that they are not as restrictive. The single quote tell the shell to ignore all enclosed characters. Double quote tell the shell to ignore all enclosed characters except *Dollar sign ($), back quote(`), back slash (\)*

```
$a=5
$echo '$a'
   $a
$echo "$a"
    5
```

In first **echo** statement **$a** is in single quote so it is ignored and **$a** was displayed.
In 2nd **echo** statement **$a** is in double quotes,t he **$** is preserved in double quotes so **the** value of **a** is displayed

```
$ x=*
$echo '$x'
    $x
$echo "$x"
  *
$echo $x
 f1 f2  m  x  d1
```

In first **echo** statement **$x** is in single quote so it is ignored and **$x** was displayed.
In 2nd **echo** statement **$x** is in double quotes, the **$** is preserved in double quotes so **the** value of **x** is displayed. i.e ***,** but ***** is not evaluated in double quotes.
In 3rd **echo** statement **$x** is replaced with ***** at **echo, echo *** displays all files in pwd.

```
$echo "one         two          three        four"
 one         two          three        four
```

Spaces are preserved in double quotes.

```
 $echo "how are you
 >today John "
 how are you
 today John
```

Even the enter key will be ignored by the shell if it is enclosed in quotes.

The back slash: The back slash is equivalent to placing a single quote around a single character, with a few minor exceptions. Format is **\c** where **c** is a character that is to be quote. Any special meaning of character **c** is ignored.

```
$echo >
Syntax error: 'newline or:' expected
$echo \>
>
$
```

In 1st **echo >** expects a file name because **>** is a o/p redirection operator.
In **2nd echo** statement the special meaning of **> is** lost because of **** and is considered as normal character.

```
$echo \*
   *
$x=*
$echo \$x
      $x
$
```

Positional parameters: In shell programming, the data can be passed to the shell script at the command line. These are known as command line arguments or positional parameters. All arguments that are passed at command line are stored in a special shell variables. There are 9 such variables that capture and hold values given in command line. They are : **$1 ,$2,$3,…$9**

$1 holds the **1st argument, $2** holds **the 2nd** argument and so on

set command is used to assign the positional parameters.

```
$set Jan  Feb Mar  Apr
$echo $1 $2 $3 $4
Jan  Feb Mar  Apr
```

1st argument is assigned **to $1(jan), 2nd** argument t**o $2(feb)** and so on.

```
$cat l
 Give luck a little time
$set `cat l`
$echo $1 $2 $3 $4 $5
 Give luck a little time
```

Here positional parameters were assigned to the data of **file l.**

If more than 9 arguments are passed at command line, then enclose all 2 digit number with in { } to assign positional parameters.

```
$set a b c d e f g h I j k l
$echo $1 $2 $3 $4 $5 $6 $7 $8 $9 {$10} {$11} {$12}
a b c d e f g h I j k l
```

shift command : shift command shifts the arguments to its left,by default it shifts one position left.

```
$set a b c d e
$shift
$echo $1 $2 $3 $4
b c d e
```

The shift command shifts $2 value to left side, so the $1 lost its value, and $1=$2, $2=$3 and so on.

```
$set a b c d e f g
$shift  3
$echo $1 $2 $3
d e f
```

Special characters:

$1 $2	positional parameters.
$#	number of positional parameters supplied at command line
$*	List of positional parameters.
$@	Same as $*, except when the arguments are enclose in double quotes.
$$	Process ID of the current shell
$!	Process ID of the last background job
$0	Name of the last command being executed
$?	Exit status of the last execute command.

Examples

```
$set do u want credit or results
$echo $#
6
$echo $*
do u want credit or results
```

>

Shell Metacharacters

```
>     prog >file direct standard output to file
>>    prog >>file append standard output to file
<     prog <file take standard input from file
|     P1|p2 connect standard output of p1 to standard input of p2
<<str  here document: standard input follows, up to next str on a line by itself
*      match any string of zero or more characters in filenames
?      match any single character in filenames
[ccc]  match any single character from ccc in filenames;
            ranges like 0-9 or a-z are legal
;     command terminator: p1;p2 does p1,then p 2
&      like ; but doesn't wait for p1 to finish
'...'  run command(s) in ...; output replaces '...'
(...)  run command(s) in ... in a sub-shell
{...}  run command(s) in ... in current shell (rarely used)
$1, $2 etc .    $0...$9 replaced by arguments to shell file
$var      value of shell variable var
${var}     value of var; avoids confusion when concatenated with text;
\c       take character c literally, \newline discarded
'...'     take ... literally
'..."              take ... literally after $, and \ interpreted
#     if # starts word, rest of line is a comment
var-value assign to variable var
P1 && P2 run p1; if successful, run p2
P1||P2   run p1; if unsuccessful, run p2
```

3.3 Creating new commands

Given a sequence of commands that is to be repeated more than a few times, it would be convenient to make it into a "new" command with its own name, so you can use it like a regular command. To be specific, suppose you intend to count users frequently with the pipeline

```
$ who |wc -1
```

you want to make a new program `nu` to do that .

The first step is to create an ordinary file that contains `'who |wc -1'`.

```
 $ echo 'who | wc -1 ' >nu
```

Run the shell with its input coming from the file `nu` instead of the terminal:

```
$ who
you tty2 Sep 28 07:51
rhh tty4 Sep 28 10:02
moh tty5 Sep 28 09:38
ava tty6 Sep 28 10 : 17
$ cat nu
who |wc -l
$ sh<nu
4
$
```

The output is the same as it would have been if you had typed `who | wc -1` at the terminal.

Again like most other programs, the shell takes its input from a file if one is named as an argument;

```
$ sh nu
```

If a file is executable and if it contains text, then the shell assumes it to be a file of shell commands. Such a file is called a **shell file**. All you have to do is to make `nu` executable, once:

```
$ chmod +x nu
```

and thereafter you can invoke it with

```
$ nu
```

From now on, users of `nu` cannot tell, just by running it, that you implemented it in this easy way.

The way the shell actually runs `nu` is to create a new shell process exactly as if you had typed

```
$ sh nu
```

This child shell is called a sub-shell — a shell process invoked by your currentshell, **`sh nu`** is not the same as **`sh<nu`**, because its standard input is still connected to the terminal.

As it stands, nu works only if it's in your current directory (provided, ofcourse, that the current directory is in your PATH, which we will assume from now on). To make **`nu`** part of your repertoire regardless of what directory you're in, move it to your private bin directory, and add **`/usr/you/bin`** toyour search path:

```
$ pwd
/usr/you
$ mkdir bin                                   Make a bin if you haven't already
$ echo $PATH                                  Check PATH for sure
: /usr/you/bin : /bin : /usr/bin              Should look like this
$ mv nu bin                                   Install nu
$ nu
4                                             But it's found by the shell
$
```

3.4 Command arguments and parameters

Suppose we want to make a program called cx to change the mode of a fileto executable, so

```
$ cx nu
```

is a shorthand for

```
$ chmod +x nu
```

We already know almost enough to do this. We need a file called cx whose contents are

```
chmod +x filename
```

The only new thing we need to know is how to tell cx what the name of the file is, since it will be different each time cx is run. When the shell executes a file of commands, each occurrence of $1 isreplaced by the first argument, each $2 is replaced by the second argument, and so on through $9, So if the file **`cx`** contains

```
chmod +x $1
```

when the command

```
$ cx nu
```

is run, the sub-shell replaces "$1" by its first argument, "nu."Let's look at the whole sequence of operations:

```
$ echo 'chmod +x $1' >cx                      Create cx originally
$ sh cx cx                                    Make cx itself executable
$ echo `who|cut -f1 -d" "'>na                 Make a test program
$ na                                          Try it
$ mv cx  /usr/you/bin                         Install cx
$ mv na /usr/you/na                           Install cx
$ cx na
$ na
you
rhh
moh
ava
$
```

Notice that we said

```
$ sh cx cx
```

exactly as the shell would have automatically done if **`cx`** were already executable and we typed

```
$ cx cx
```

If you want to handle more than one argument, for example to makea program like **`cx`** handle several files at once then we can use command line arguments or positional parameters.

```
chmod +x $1 $2 $3 $4 $5 $6 $7 $8 $9
```

the effect is that only the arguments that were actually provided are passed to chmod by the sub-shell. The proper way to define cx, then, is

```
chmod +x $*
```

which works regardless of how many arguments are provided.With $* added to your repertoire, you can make some convenient shellfiles, such as 1c or m:
Create a command lc that counts the no.of lines of specified files.$ cd /usr/you/bin

```
$ cat 1c
wc -1 $*
$chmod +x lc
$mv lc /usr/you/bin
$lc f1 f2 f3 a b c
4   f1
5   f2
3   f3
10  a
1   b
4   c
$
```

For example, consider searching a personal telephone directory. If you have a file named *phone-book* that contains lines like

```
dial-a- joke 212-976-3838
dial -a -prayer 212-246-4200
dialsanta 212-976-3636
dow jones report 212-976-4141
```

then the grep command can be used to search it. (Your own lib directory is a good place to store such personal data bases.). Let's make a directory assistance program, which we'll call 411 in honor of the telephone directory assistance number where welive:

```
$ echo ' grep $* phone-book ' >411
$ cx 411
$ 411 joke
dial-a- joke 212-976-3838
$411 dial
dial-a- joke 212-976-3838
dial-a-prayer 212-246-4200
dialsanta 212-976-3636
$
```

3.5 Program output as arguments

The output of any program can be placed in a command line by enclosing the invocation in backquotes` `

```
$ echo At the tone the time will he 'date'.
At the tone the time will be Thu Sep 29 00:02: 15 EDT 1983.
$
```

A small change illustrates that **`...`** is interpreted inside double quotes **"…"**

```
$ echo "At the tone
> the time will be `date`."
At the tone
the time will be Thu Sep 29 00:03:07 EDT 1983.
$
```

As another example, suppose you want to send mail to a list of people whose login names are in the file **mailinglist.** A clumsy way to handle thisis to edit **mailinglist** into a suitable mail command and present it to the shell, but it's far easier to say

```
$ mail `cat mailinglist` <letter
```

This runs cat to produce the list of user names, and those become the arguments to mail. (When interpreting output in backquotes as arguments, theshell treats newlines as word separators, not command-line terminators; Backquotes are easy enough to usethat there's really no need for a separate mailing-list option to the mail command.
A slightly different approach is to convert the file mailinglist from just a list of names into a program that prints the list of names:

```
$ cat mailinglist New version
echo don whr ejs mb
$ cx mailinglist
$ mailinglist
```

```
don whr ejs mb
$
```
Now mailing the letter to the people on the list becomes
```
$ mail `mailinglist ` <letter
```
With the addition of one more program, it's even possible to modify the user list interactively. The program is called **pick:**
```
$ pick arguments ...
```
presents the arguments one at a time and waits after each for a response. The output of pick is those arguments selected by y (for "yes") responses; any other response causes the argument to be discarded. For example,
```
$ pr `pick *.c`|lpr
```
presents each filename that ends in .c; those selected are printed with **pr** and **lpr**.
```
 $ mail  `pick \ `mailinglist\ `` <letter
don? y
whr?
ejs?
mb? y
$
```
sends the letter to **don and mb**. Notice that there are nested backquotes; the back slashes prevent the interpretation of the inner during the parsing ofthe outer one.

3.6 Shell variables

SHELL VARIABLES: Variables are "words" that hold a *value*.

Shell variables are of two types: 1. User define variables 2. Environment or System variables.

1. Environment variables: system variables or environment variables are defined by the system itself. System variables are set either during the boot sequence or intermediately after login.

- The working environment, under which a user works, depends entirely upon the values of these variables. When a new shell is started by user, some of these variables are inherited by the sub shell from its parent.
- Environment variables are different in scope from simple shell variables. These are similar to global variables . they visible in the users total environment.
- Environment variables are: **HOME, PATH, USER or LOGNAME, MAIL, MAILCHECK, HISTSIZE, HISTFILE, HISTFILESIZE, TERM, PWD, CDPATH, PS1, PS2, SHELL, IFS, TZ.**
- The set statement displays a complete list of all variables. The *env* command statement shows only the environment variables.

```
$env
CDPATH=.:..:$HOME
HOME=/home/cse501
LOGNAME=cse501
MAIL=/var/mail/cse501
MAILCHECK=60
PAGER=/usr/bin/more
PATH=/bin:/usr/bin:/usr/dt/bin:/home/cse501/d1
PWD= home/cse501
PS1=$
SHELL=/bin/bash
TERM=ansi
…..
```

1.HOME: Indicates the home directory of the current user. When a user is login , UNIX places him in a directory called as home directory. A user's home directory is specified in the line pertaining to that user in */etc/passwd* file. This file contains a line for every user with 7 fields' per line.

cse501:xxx:208:50::/home/cse501/bin/sh the homedirectory is specified in the 6th field.

A user can change the value of home as **HOME=/home/John.**

2.LOGNAME: stores login name of user.

```
 $echo $LOGNAME
 /home/cse501
```

3.MAIL: the mail variable determines the path of the file where all incoming mail addressed to the user is stored.

Normally the mail variable holds **/usr/spool/mail/cse501 or /usr/spool/mail or /var/mail or /var/spool/mail.** The login shell searches this path every time when a user logs into the system. If the file contains any mail, the shell informs the user with a message **"you have a mail".**

```
$echo $MAIL
 /var/spool/mail
```

4. MAILCHECK: This variable stores the positive number in seconds that specifies how often the shell should check for the arrival of the mail in the path specified by the **MAIL** variable. The default value of **MAILCHECK v**ariable is **600** Seconds. If the **MAILCHECK v**ariable is **0 ,**the checks for mail every time a user logs into the system.

```
$echo $MAILCHECK
600
```

5.PATH: Indicates search path for commands. It is a colon-separated list of directories in which the shell looks for commands. It is a list of directories, that are to be searched to execute a command.

A common value is

```
$echo $PATH
/bin:/sbin:/usr/bin:/usr/sbin:/usr/local/bin:/usr/ucb
```

The user can add or remove the search path to the existing contents of the PATH variable.

```
PATH=$PATH:/home/cse501
```

After this statement, to execute a command, the shell searches your own directory also.

6.PS1: It contains the login shell prompt. The default value of the **PS1** variable is **'$'.**

```
$echo PS1
   $
```

User can change the shell prompt by changing the value of PS1

```
$PS1="UNIX"
UNIX mkdir d1
UNIX rmdir d1  /* here the shell prompt is changed from $ to UNIX*/
```

7. PS2: It contains the sub shell prompt. The default value of the **PS2** variable is **'>'.**

```
  $echo PS2
   >
```

User can change the shell prompt by changing the value of PS2

```
$PS2="OS"
OS for a in 1 2
OS do
OS echo $a
OS done
o/p:
1
2
```

8.IFS: Indicates the Internal Field Separator that is used by the parser for word splitting after expansion. **$IFS** is also used to split lines into words with the read built-in command. The default value is the string, **" \t\n ",** where **" "** is the space character, **\t** is the tab character, and **\n** is the newline character.

User can assign a character as a field separator value. **$IFS=:** after this statement the field separator is not space or tab but it is **:**

9.HZ: the **HZ(hertz)** variable stores the no.of clock interrupts per second. The default value of **HZ** is **100.**the value of **HZ** is stored in the /etc/default/login

```
$echo $HZ
100
```

10.SHELL: The **SHELL** variable stores the pathname of the shell in which the user is currently working. The default value of **SHELL** is **/bin/bash.** The system administrator sets the value of the **SHELL** variable when he creates an account for the user.

```
$echo $SHELL
/bin/bash
```

11.TERM : the **TERM** variable indicates the terminal type being used. Several programs like *vi* are terminal dependent and they require to know the type of the terminal you are using. If the **TERM** variable is not set correctly *vi* won't work properly and display will be faulty.

12. PWD: Indicates the current working directory as set by the *cd* command.

13.TZ(Time zone): It stores the time zone specification, used by commands like `date, at, batch.` To set the time appropriate to your time zone. The value stored in the TZ variable will be in the format XXXYZZZ where XXX stores the standard local time zone abbreviation like IST,EST,PST etc.. , Y is the difference b/n the local time zone and Green wich Mean Time(GMT) and ZZZ is the abbreviation for the summer daylight savings time zone.

```
$echo $TZ
IST5TDT
```

2.User defined variables: Variables are "words" that hold a value. The shell enables you to create, assign, and delete variables. Although the shell manages some variables, it is mostly up to the programmer to manage variables in shell scripts. By using variables, you are able to make your scripts flexible and maintainable.

The value of a variable is associated with the shell that creates it, and is not automatically passed to the shell's children.

```
$ x=Hello                          Create x
$ sh                               New shell
$ echo $x                          Newline only: x undefined in the sub-shell
$ ctl-d                            Leave this shell
$                                  Back in original shell
$ echo $x                          x still defined
Hello
$
```

This means that a shell file cannot change the value of a variable, because the shell file is run by a sub-shell:

```
$ echo 'x= "Good Bye"              Make a two-line shell file ...
>echo $x' >setx                    ...to set and print x
$ cat setx
x="Good Bye"
echo $x
$ echo $x
Hello                                  x is Hello in original shell
$ sh setx
Good Bye                           x is Good Bye in sub-shell...
$ echo $x
Hello                              ...but still Hello in this shell
$
```

The other way to set the value of a variable in a sub-shell is to assign to it explicitly on the command line before the command itself:

```
$ echo 'echo $x ' >ec
$ cx ec
$ echo $x
Hello                                  As before
$ ec
                                       x not set in sub-shell
$ x=Hi ec
Hi                                     Value of x passed to sub-shell
$
```

When you want to make the value of a variable accessible in sub-shells, thes hell's export command should be used.

```
$ x=Hello
$ export x
$ sh                                   New shell
$ echo $x
Hello                                   x known in sub-shell
$x="Good Bye "                         Change its value
$ echo $x
Good Bye
$ ctl-d                                Leave this shell
$                                      Back in original shell
$ echo $x
Hello                                   x still Hello
$
```

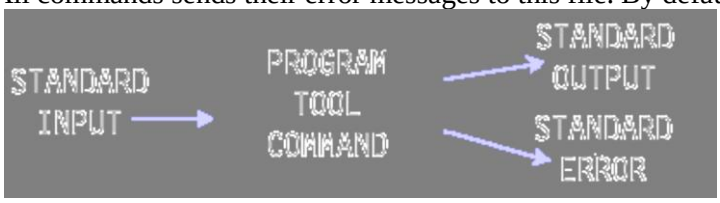
3.7 More on I/O redirection

In UNIX the files are divided into 3 types: standard input file, standard output file, standard error file

standard input file(stdin): From this file all commands accepts the input, by default commands are entered at the keyboard, so keyboard is a standard input file:

***standard output file(stdout*)**: all commands sends their output to this standard output file. By default it is monitor.

***standard error file(stderr*)**: All commands sends their error messages to this file. By default it is monitor.



Every file in Unix is associated with an unique number called as file descriptor value.

The file descriptor for standard input is **0 (zero)**

The file descriptor for standard output is **1**

The file descriptor for standard error is **2.**

***Input devices*:** The input devices in Unix are Keyboard ,file, pipe. i.e a command can accepts its i/p from Keyboard or, file or , pipe

***Out put devices*:** The output devices in Unix are monitor ,file, pipe. i.e a command can send its O/P to monitor or, file or , pipe.

Redirection: this is a process of reading input from a file instead of keyboard, pipe and writing output to file instead of monitor or pipe.

I/P redirection: If a command accepts its input from a file it is called as input redirection. and indicated with the symbol < It can be write as: **command < file**

```
$cat<f1  [Enter]
   foo
   bar
   fred
   barney
   dino
```

Here **cat<f1** or **cat f1** both are same. In these 2 cases **cat a**ccepts its input from file **f1.** And displays that on monitor

The **wc (word count)** command counts the number of bytes, word and lines in a file.

```
$ wc<f1 [Enter]
6       7       39    f1
```

wc<f1 here **wc** command accepts input from file **f1** and displays the no. of lines 6, words 7, characters 39 followed by the name of the file **wc** opened.

wc<f1 or wc f1 both are same**. wc f1** assumes that there is an i/p redirection operator between **wc and f1**

O/P redirection: If a command writes its output to a file it is called output redirection. And indicated with the symbol >

It can be write as:**command > file**

```
$ echo "Hello World!" > s [Enter]
$ cat s [Enter]
  Hello World!
```

Here the echo command written it's o/p to a file *s* ,because of o/p redirection.

2.$ ls > m [Enter] /* send the out put of **ls** command to a file called **m */**

3.

```
$ wc<f1>x[Enter]
$cat x
6       7       39
```

/* here the **wc f1** command writes its output to file **x. w**e can view that by **cat x*/**

The same command can be written as : **wc<f1>x or wc>x<f1 or >x<f1 wc** in these all commands the **wc** accepts its i/p from **f1** and writes its o/p to **x.**

When **<, >** appears in the same line **<(I**/P redirection) has the higher priority than **>(**O/P redirection).

Note: If the file mentioned at output redirection is already exists, it is overwritten.

In case you want to append to an existing file, then instead of the '**> '** operator you should use the **' >> '** operator. This would append to the file if it already exists, else it would create a new file by that name and then add the output to that newly created file.

```
$ echo "Hello World!" > s [Enter]
$ cat s [Enter]
   Hello World!
$wc  f1>s [Enter]
$cat s [Enter]
  6        7       39
```

In this example first the output of **echo** is redirected to a file called s.
Second time when **wc** command o/p is redirected to the same file **s** ,then the previous contents of s are overwritten. So we can find only **wc** o/p in file **s**. we can avoid this with **>>** symbol.

```
$ echo "Hello World!" > s [Enter]
$ cat s [Enter]
   Hello World!
$wc  f1>>s [Enter]
$cat s [Enter]
   Hello World!
  6        7       39
```

In this example first the output of **echo** is redirected to a file called s.
Second time when **wc** command o/p is redirected to the same file **s** using the operator **>>** , we can find both outputs **(echo, wc) in s** file.

Standard error devices are indicated with the FD 2. This is a device where all error messages are redirected. It may a terminal or a file.

```
$cat> y
This is pen
[ctrl-d]
$cat   z
This is rose
[ctrl-d]
```

```
$ls [Enter]
f1    m    s    x    y    z
$cat  a  [Enter]
cat: cannot open a: No such file or directory
```

In this example file **a** is not existed so it displayed error message on terminal.

```
$ls [Enter]
f1    m    s   x  y    z
$cat  y z a >f5  [Enter]
cat: cannot open a: No such file or directory
$ cat  f5  [Enter]
This is pen
This is rose
```

By using o/p redirection operator also the error is not redirected to a file **f5** . instead it was displayed on terminal. Only **x y** files o/p was redirected to **f5** file.

To redirect error messages to a file use O/P redirection operator with the file descriptor value of standard error device i.e 2. So error redirection is possible with (2>).

```
$ls [Enter]
f1    m    s   x  y    z
$cat  a  2> f6
```

In this example the error message regarding the file **a** is redirected to a file **f6**

```
$cat  y  z  a  >p   2>q [Enter]
$cat p [Enter]
This is pen
This is rose
$cat q [Enter]
cat: cannot open a: No such file or directory
```

In this example the output is redirected to a file **p** and the error message is redirected to a file **q**.

To send both standard output and standard error to the same file use the following format:

command > file 2>> file or **command > file 2>&1**

```
$cat  y  z  a  >k   2>&1 [Enter]
$cat k [Enter]
This is pen
This is rose
cat: cannot open a: No such file or directory
$
```

In this example the output , error messages two are redirected to a same **file k.** the following command also produces the same output:

Shell I/O Redirections

>file direct standard output to file
>>file append standard output to file
<file take standard input from file
P1|P2 connect standard output of program p1 to input of p2
^ obsolete synonym for !
n>file direct output from file descriptor n to file
n>>file append output from file descriptor n to file
n>&m merge output from file descriptor n with file descriptor m
n<&m merge input from file descriptor n with file descriptor m
<<s here document: take standard input until next *s* at beginning of a line;
<<\s here document with no substitution
<< 's' here document with no substitution

Pipes: A Pipe is a general mechanism by using which the output of one command is connected or redirected as the input to another command directly without using any temporary files.

To send the output of one command as input for another, the two commands must be joined using a **|** operator.

We can pipe or connect any no.of commands on the pipeline. The generalize form is

Command1|command2|command3|…..|commandn

In the above format

a. Command1 should support the o/p redirection. i.e it should produce some output ex: **ls,who,wc ..**
b. Command 2,command3,…command n-1 all these commands should support both i/p,o/p redirection. Ex: **cmp,cut,head, wc**…
c. Command n should support i/p redirection. **Ex:bc,wc,cut**…

The pipes, redirection operators several commands can be combined together to perform complex tasks

2. Write a command to display the no. of users that are logged in.

To get the no.of users that are currently logged in we don't have a direct command. But we can solve this using either redirection or piping.

```
$who
 root     console    Feb12:00
cse501    tty01      Feb 14:09
cse502    tty03      Feb 14:13
cse503    tty07      Feb 15:02
```

Assume this is the o/p of **who** command

```
$who|wc  -l
4
```

Here first **who** command writes its output to a pipe (|). Next **wc** command reads the input from pipe. (Which is the o/p **of who**) and counts the no. of lines from it.

2.**Write a command to display only login names of users who are currently logged n.**

```
$who|cut –d " "  -f 1
root
cse501
cse502
cse503
```

Here first **who** command writes its output to a pipe (|). Next **cut** command reads the input from pipe. (Which is the o/p **of who**) and cuts the first field(which is login name of users.)

3. **.Write a command to display login names, login time of users who are currently logged n.**

```
$who|cut –d " "  -f 1,3
root        Feb12:00
cse501      Feb 14:09
cse502      Feb 14:13
cse503      Feb 15:02
```

Here first **who** command writes its output to a pipe (|). Next **cut** command reads the input from pipe. (Which is the o/p **of who**) and cuts the first ,third fields(which is login name, login time of users.)

5. **Display no.of files and directories in current working directory .**

```
$ls|wc  -l
12
```

ls command displays all files in pwd to pipe. ***wc*** reads from pipe & counts the no.of lines from it.

7.**Display the files which have write permission for the group.**

```
$ls -l|grep "-----w" | cut -f9
```

8.Display Both modification and access time of files

```
$ls -l| tr -s ' '|cut -d" " f 6-9  >temp
$ls -lu| tr -s ' '|cut -d" " f 6-9|paste temp
```

`ls -l` displays long listing of files with modification time. **`ls -lu`** displays long listing of files with access time. **`tr`** command squeezes multiple spaces to single space. **`cut`** command cuts the field number 6 to 9 which is the time, file name ,**`paste`** command pastes both timings in single row.

3.8 Looping in shell programs

The shell is actually a programming language: it has variables, loops,decision-making, and so on. the shell's for statement is the only shell control-flow statement that you might commonly type at the terminal rather than potting in a file for later execution. The syntax is:

```
for var in list of words
do
commands
done
```

For example, a for statement to echo filenames one per line is just

```
$ for i in *
> do
>echo $i
> done
```

The "i" can be any shell variable, although i is traditional. Note that the variable's value is accessed by $i, but that the for loop refers to the variable as i. We used * to pick up all the files in the current directory, but any other list of arguments can be used. Normally you want to do something moreinteresting than merely printing filenames

```
$ ls ch2 .* | 5
ch2 . 1 ch2 . 2 ch2 . 3 ch2 . 4 ch2 .5
ch2 . 6 ch2 .7
$ for i in ch2 .*
> do
>echo $i:
>diff -b old/$i $i
>echo                                          Add a blank line for readability
> done | pr -h "diff `pwd`/old `pwd `" lpr&
3712                                           Process-id
$
```

We piped the output into pr and lpr just to illustrate that it's possible: the standard output of the programs within a for goes to the standard output of the for itself.

`for i in *` which loops over all filenames in the current directory, and

`for i in $*` which loops over all arguments to the shell file.)

The argument list for a for most often comes from pattern matching on filenames, but it can come from anything. It could be

```
$ for i in 'cat . . .'
```

or arguments could just be typed.

1. SHELL:

- The shell is a program (usually stored in the file 'sh' in the 'bin' directory) that acts as an interface b/w the user and the OS (kernel).
- As the name suggests the shell envelops the kernel.
- The shell is a program that acts as a command interpreter reads lines (i.e., commands) typed by the user and translates the requests into actions passing the commands directly to the kernel.

- All communications b/w the kernel and the user pass only through the shell. Shell also works as a programming language. The shell programs are known as shell scripts.
- The shell is not a part of the kernel but an application program that you see and use.
- Whenever the user logs into the system, a UNIX shell is automatically invoked and starts running as a separate, personal copy for each user i.e., at a particular instance there may be several copies of shell running on the system simultaneously with a minimum of one shell per user.
- Once authorized by the kernel to enter the login session, the shell displays the shell prompt ($ for ordinary users and # for super users).
- When the user types a command, the shell decodes the command line and searches for a program with the same name as the command in the list of directories containing program files as specified by the environmental variable PATH.
- In UNIX, there are only error messages and no success messages. If you type a command, and press the 'enter' key wait for the execution of the command. If there are no error messages, then, the meaning of that is the command has been successfully executed.
- Some of the most popular shells used in UNIX systems are.
  a) Bourne shell
  b) C shell
  c) Korn shell

a) Bourne Shell: The oldest UNIX shell, and most used one, was originally developed by Steve Bourne of AT&T Bell labs in the late 1970's.

- The Bourne shell is stored in the program file *sh.*

Features*:*

- A built in command set for writing shell scripts
- A set of shell variables for configuring your environment.
- Background execution of commands.
- Supports both I/O redirection.
- Supports use of wild card characters.

b) C shell: The C shell also referred as the Berkeley C shell was designed and developed by Bill Joy at the university of California at Berkeley for BSD UNIX distributions is now a part of system V.

- The C shell is stored in the program file *csh*

Features*:*

- A built in command set for writing shell scripts
- A set of shell variables for configuring your environment.
- Supports both I/O redirection .
- Supports command aliasing of frequently used commands.
- Supports rerunning of previously used commands
- Supports command substitution & command history.
- Supports job control

c)Korn Shell: The Korn shell was developed by David Korn of AT&T Bell labs in the late 1980's. The Korn shell is the superset of the Bourne shell and hence everything that works with the Bourne shell also works in the Korn shell.

- The Korn shell is stored in the program file *Ksh.*

Features*:*

- Supports command line editing with editors like vi, ed or emacs.
- Supports command aliasing and command history.
- Supports job control.
- Supports extensive pattern matching.

All the Shells are most similar, their syntax, standard features, and notations are extremely close to one another.

The default prompts of shells are:

(i) Bourne Shell(**$**) (ii) C Shell(**%**) (iii) Korn Shell(**$**)

Other Shells used in UNIX system includes.

a)Bash shell **b)** pd-Ksh shell **c)** dt-Ksh shell **d)** Tc Shell **e)** POSIX shell **f)**z shell **g)**Restricted Shell

a) Bash Shell(Bourne Again Shell): is an expanded version of the Bourne shell was developed by the free s/w foundation the bash shell has an extended shell scripting language that includes advanced loop constructs and other functions. The bash shell is stored in the program file */bin/bash.*

b)pd- Ksh Shell(public domain Korn Shell): is an expanded version of the Korn Shell developed at AT & T Bells labs.

The pd-Ksh is available in both binary and source form.

For a shell programmer there is no difference b/n the korn shell and pd-ksh shell.

All scripts that run in one version will also run in other version.

c)dt-Ksh shell(desktop Korn Shell) : is also an expanded version of the Korn shell, which provides the capability to create and display GUI's using the Korn shell syntax. It is stored in /*bin/dt-ksh*

***d*)TC Shell(Turbo C Shell):** is an expanded version of the C shell developed at the BSD UNIX distribution systems.

It has many advanced features like the command line editing, spelling correction, and easy retrieval of previously executed commands, immediate documentation access as the user types a command, ability to schedule for periodic execution.

e**)POSIX Shell(Portable Operating System Interface Shell)** is a superset of the Bourne shell. But it is much like the

Korn shell .The POSIX shell also offers aliases, path searches, command history, filename completion and job control.

f) Z- Shell: Developed by Paul Falstand in the early 90's as expanded version of the Bourne Shell, C shell and Korn shell with its own features. It is stored in a file */bin/zsh*

h) Restricted Shell: the restricted shell is almost same as the regular shell. But it is designed to restrict user's capabilities by restricting some of the standard actions that an ordinary shell allows.

A user working in the restricted shell

- Cannot change from his HOME directory to any system directories.
- Cannot create new files or append to existing files by redirecting the O/P using > or >> operators.
- Cannot change the PATH shell variable.
- Cannot use a command containing the symbol /