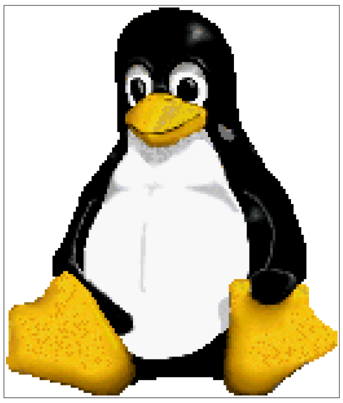


Linux Shell Scripting Basics

I N D E X

Introduction

This tutorial is designed for beginners only and This tutorial explains the basics of shell programming by showing some examples of shell programs. Its not help or manual for the shell. While reading this tutorial you can find manual quite useful (type **man bash** at $ prompt to see manual pages). Manual contains all necessary information you need, but it won't have that much examples, which makes idea more clear. For that reason, this tutorial contains examples rather than all the features of shell. I assumes you have at least working knowledge of Linux i.e. basic commands like how to create, copy, remove files/directories etc or how to use editor like vi or mcedit and login to your system. Before Starting Linux Shell Script Programming you must know

- Kernel
- Shell
- Process
- Redirectors, Pipes, Filters etc.

What's Kernel

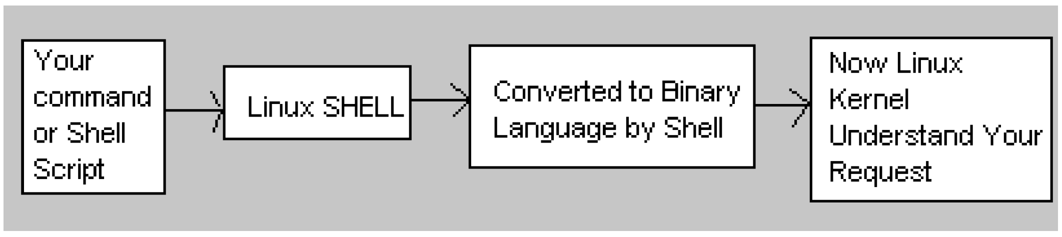
Kernel is hart of Linux O/S. It manages resource of Linux O/S. Resources means facilities available in Linux. For eg. Facility to store data, print data on printer, memory, file management etc . Kernel decides who will use this resource, for how long and when. It runs your programs (or set up to execute binary files) It's Memory resident portion of Linux. It performance following task :-

- I/O management
- Process management
- Device management
- File management
- Memory management

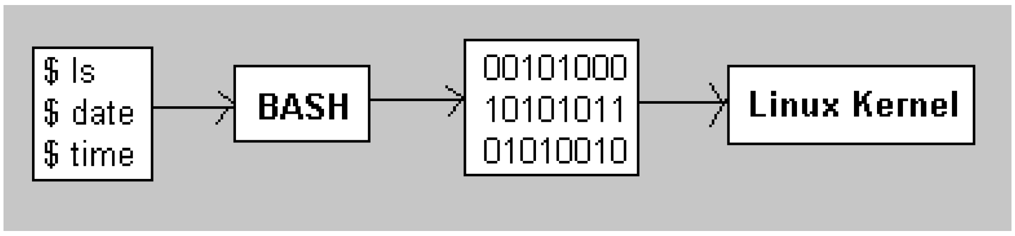
What's Linux Shell

Computer understand the language of 0's and 1's called binary language, In early days of computing, instruction are provided using binary language, which is difficult for all of us, to read and write. So in O/S there is special program called Shell. Shell accepts your instruction or commands in English and translate it into computers native binary language.

This is what Shell Does for US


Your command or Shell Script
Linux SHELL
Converted to Binary Language by Shell
Now Linux Kernel Understand Your Request

You type Your command and shell convert it as



It's environment provided for user interaction. Shell is an command language interpreter that executes commands read from the standard input device (keyboard) or from a file. Linux may use one of the following most popular shells (In MS-DOS, Shell name is COMMAND.COM which is also used for same purpose, but it's not as powerful as our Linux Shells are!)

Shell Name	Developed by	Where	Remark
BASH (Bourne-Again SHell)	Brian Fox and Chet Ramey	Free Software Foundation	Most common shell in Linux. It's Freeware shell.
CSH (C SHell)	Bill Joy	University of California (For BSD)	The C shell's syntax and usage are very similar to the C programming language.
KSH (Korn SHell)	David Korn	AT & T Bell Labs	

Any of the above shell reads command from user (via Keyboard or Mouse) and tells Linux O/s what users want. If we are giving commands from keyboard it is called command line interface (Usually in-front of $ prompt, This prompt is depend upon your shell and Environment that you set or by your System Administrator, therefore you may get different prompt).
NOTE: To find your shell type following command
$ echo $SHELL

How to use Shell

To use shell (You start to use your shell as soon as you log into your system) you have to simply type commands. Following is the list of common commands.

Linux Common Commands

NOTE that following commands are for New users or for Beginners only. The purpose is if you use this command you will be more familiar with your shell and secondly, you need some of these command in your Shell script. If you want to get more information or help for this command try following commands For e.g. To see help or options related with date command try
$ date --help
or To see help or options related with ls command (Here you will screen by screen help, since help of ls command is quite big that can't fit on single screen)

$ ls --help | more

Syntax: *command-name --help*
Syntax: *man command-name*
Syntax: *info command-name*

See what happened when you type following

$ man ls
$ info bash

NOTE: In MS-DOS, you get help by using /? clue or by typing help command as

C:\> dir /?
C:\> date /?
C:\> help time
C:\> help date
C:\> help

Linux Command

For this Purpose	Use this Command Syntax	Example (In front of $ Prompt)
To see date	date	**$ date**
To see who's using system.	who	**$ who**
Print working directory	pwd	**$ pwd**
List name of files in current directory	ls or dirs	**$ ls**
To create text file NOTE: Press and hold **CTRL** key and press **D** to stop or to end file (**CTRL+D**)	cat > { file name }	**$ cat > myfile** *type your text when done press ^D*
To text see files	cat {file name }	**$ cat myfile**
To display file one full screen at a time	more {file name }	**$ more myfile**
To move or rename file/directory	mv {file1} {file2}	**$ mv sales sales.99**
To create multiple file copies with various link. After this both oldfile newfile refers to same name	ln {oldfile} {newfile}	**$ ln Page1 Book1**
To remove file	rm file1	**$ rm myfile**

Remove all files in given directory/subdirectory. Use it very carefully.	rm -rf {dirname}	**$ rm -rf oldfiles**
To change file access permissions u - User who owns the file g - Group file owner o - User classified as other a - All other system user + Set permission - Remove permission r - Read permission w - Write permission x - Execute permission	chmod {u|g|o|a} {+|-} {r|w|x} {filename}	**$ chmod u+x,g+wx,o+x myscript** NOTE: This command set permission for file called 'myscript' as User (Person who creates that file or directory) has execute permission (u+x) Group of file owner can write to this file as well as execute this file (g+wx) Others can only execute file but can not modify it, Since we have not given w (write permission) to them. (o+x).
Read your mail.	mail	**$ mail**
To See more about currently login person (i..e. yourself)	who am i	**$ who am i**
To login out	logout (OR press CTRL+D)	**$ logout** *(Note: It may ask you password type your login password, In some case this feature is disabled by System Administrator)*
Send mail to other person	mail {user-name}	**$ mail ashish**
To count lines, words and characters of given file	wc {file-name}	**$wc myfile**
To searches file for line that match a pattern.	grep {word-to-lookup} {filename}	**$ grep fox myfile**
To sort file in following order -r Reverse normal order -n Sort in numeric order -nr Sort in reverse numeric order	sort -r -n -nr {filename}	**$sort myfile**

To print last | first line of given file	tail - | + { linenumber } {filename}	**$tail +5 myfile**

To Use to compare files	cmp {file1} {file2}
	 OR
	 diff {file1} {file2}
		$cmp myfile myfile.old

To print file	pr {file-name}	**$pr myfile**

What is Processes

Process is any kind of program or task carried out by your PC. For e.g. **$ ls -lR** , is command or a request to list files in a directory and all subdirectory in your current directory. It is a process. A process is program (command given by user) to perform some Job. In Linux when you start process, it gives a number (called PID or process-id), PID starts from 0 to 65535.

Why Process required

Linux is multi-user, multitasking o/s. It means you can run more than two process simultaneously if you wish. For e.g.. To find how many files do you have on your system you may give command like

```
$ ls / -R | wc -l
```

This command will take lot of time to search all files on your system. So you can run such command in Background or simultaneously by giving command like

```
$ ls / -R | wc -l &
```

The ampersand (&) at the end of command tells shells start command (ls / -R | wc -l) and run it in background takes next command immediately. An instance of running command is called process and the number printed by shell is called process-id (PID), this PID can be use to refer specific running process.

Linux Command Related with Process

For this purpose	Use this Command	Example
To see currently running process	ps	**$ ps**
To stop any process i.e. to kill process	kill {PID}	**$ kill 1012**
To get information about all running process	ps -ag	**$ ps -ag**
To stop all process except your shell	kill 0	**$ kill 0**
For background processing (With &, use to put particular command and program in background)	linux-command &	**$ ls / -R | wc -l &**

NOTE that you can only kill process which are created by yourself. A Administrator can almost kill 95-98% processes. But some processes can not be killed, such as VDUProcess.

Redirection of Standard output/input or, Input - Output redirection

Mostly all commands give output on screen or take input from keyboard, but in Linux it's possible to send output to file or to read input from file. For e.g. **$ ls** command gives output to screen; to send output to file of ls give command , **$ ls > filename**. It means put output of ls command to filename. There are three main redirection symbols >,>>,<

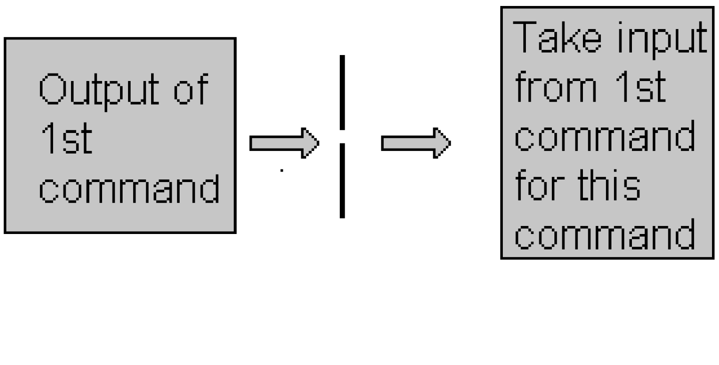
(1) > Redirector Symbol
Syntax: *Linux-command > filename*
To output Linux-commands result to file. Note that If file already exist, it will be overwritten else new file is created. For e.g. To send output of ls command give
$ ls > myfiles
Now if 'myfiles' file exist in your current directory it will be overwritten without any type of warning. (What if I want to send output to file, which is already exist and want to keep information of that file without loosing previous information/data?, For this Read next redirector)

(2) >> Redirector Symbol
Syntax: *Linux-command >> filename*
To output Linux-commands result to END of file. Note that If file exist , it will be opened and new information / data will be written to END of file, without losing previous information/data, And if file is not exist, then new file is created. For e.g. To send output of date command to already exist file give
$ date >> myfiles

(3) < Redirector Symbol
Syntax: *Linux-command < filename*
To take input to Linux-command from file instead of key-board. For e.g. To take input for cat command give
$ cat < myfiles

Pips

A pipe is a way to connect the output of one program to the input of another program without any temporary file.



A pipe is nothing but a temporary storage place where the output of one command is stored and then passed as the input for second command. Pipes are used to run more than two commands (Multiple commands) from same command line.
Syntax: *command1* | *command2*

Command using Pips	Meaning or Use of Pipes
$ ls | more	Here the output of ls command is given as input to more command So that output is printed one screen full page at a time
$ who | sort	Here output of who command is given as input to sort command So that it will print sorted list of users
$ who | wc -l	Here output of who command is given as input to wc command So that it will number of user who logon to system
$ ls -l | wc -l	Here output of ls command is given as input to wc command So that it will print number of files in current directory.
$ who | grep raju	Here output of who command is given as input to grep command So that it will print if particular user name if he is logon or nothing is printed (To see for particular user logon)

Filter

If a Linux command accepts its input from the standard input and produces its output on standard output is know as a filter. A filter performs some kind of process on the input and gives output. For e.g.. Suppose we have file called 'hotel.txt' with 100 lines data, And from 'hotel.txt' we would like to print contains from line number 20 to line number 30 and store this result to file called 'hlist' then give command

```
$ tail +20 < hotel.txt | head -n30 >hlist
```

Here head is filter which takes its input from tail command (tail command start selecting from line number 20 of given file i.e. hotel.txt) and passes this lines to input to head, whose output is redirected to 'hlist' file.

Introduction to Shell Programming

Shell program is series of Linux commands. Shell script is just like batch file is MS-DOS but have more power than the MS-DOS batch file. Shell script can take input from user, file and output them on screen. Useful to create our own commands that can save our lots of time and to automate some task of day today life.

Variables in Linux

Sometimes to process our data/information, it must be kept in computers RAM memory. RAM memory is divided into small locations, and each location had unique number called memory location/address, which is used to hold our data. Programmer can give a unique name to this memory location/address called memory variable or variable (Its a named storage location that may take different values, but only one at a time). In Linux, there are two types of variable
1) System variables - Created and maintained by Linux itself. This type of variable defined in CAPITAL LETTERS.
2) User defined variables (UDV) - Created and maintained by user. This type of variable defined in lower LETTERS.

Some System variables

You can see system variables by giving command like **$ set**, Some of the important System variables are

System Variable	Meaning
BASH=/bin/bash	Our shell name
BASH_VERSION=1.14.7(1)	Our shell version name
COLUMNS=80	No. of columns for our screen
HOME=/home/vivek	Our home directory
LINES=25	No. of columns for our screen
LOGNAME=students	Our logging name
OSTYPE=Linux	Our o/s type : -)
PATH=/usr/bin:/sbin:/bin:/usr/sbin	Our path settings
PS1=[\u@\h \W]\$	Our prompt settings
PWD=/home/students/Common	Our current working directory
SHELL=/bin/bash	Our shell name
USERNAME=vivek	User name who is currently login to this PC

NOTE that Some of the above settings can be different in your PC. You can print any of the above variables contain as follows

```
$ echo $USERNAME
$ echo $HOME
```

Caution: Do not modify System variable this can some time create problems.

How to define User defined variables (UDV)

To define UDV use following syntax
Syntax: *variablename=value*
NOTE: Here 'value' is assigned to given 'variablename' and Value must be on right side = sign For e.g.

```
$ no=10     # this is ok
$ 10=no     # Error, NOT Ok, Value must be on right side of = sign.
```

To define variable called 'vech' having value Bus

$ vech=Bus
To define variable called n having value 10
$ n=10

Rules for Naming variable name (Both UDV and System Variable)

(1) Variable name must begin with Alphanumeric character or underscore character (_), followed by one or more Alphanumeric character. For e.g. Valid shell variable are as follows
HOME
SYSTEM_VERSION
vech
no

(2) Don't put spaces on either side of the equal sign when assigning value to variable. For e.g.. In following variable declaration there will be no error
$ no=10
But here there will be problem for following
```
$ no     =10
$ no=     10
$ no     =     10
```

(3) Variables are case-sensitive, just like filename in Linux. For e.g.
$ no=10
$ No=11
$ NO=20
$ nO=2
Above all are different variable name, so to print value 20 we have to use **$ echo $NO** and Not any of the following
```
$ echo $no       # will print 10 but not 20
$ echo $No       # will print 11 but not 20
$ echo $nO       # will print 2 but not 20
```

(4) You can define NULL variable as follows (NULL variable is variable which has no value at the time of definition) For e.g.
$ vech=
$ vech=""
Try to print it's value **$ echo $vech** , Here nothing will be shown because variable has no value i.e. NULL variable.

(5) Do not use ?,* etc, to name your variable names.

How to print or access value of UDV (User defined variables)

To print or access UDV use following syntax
Syntax: *$variablename*
For eg. To print contains of variable 'vech'
$ echo $vech
It will print 'Bus' (if previously defined as vech=Bus) ,To print contains of variable 'n' **$ echo $n**
It will print '10' (if previously defined as n=10)
Caution: Do not try **$ echo vech** It will print vech instead its value 'Bus' and $ echo n, It will print n instead its value '10', You must use $ followed by variable name.

Q.1.How to Define variable x with value 10 and print it on screen
$ x=10
$ echo $x

Q.2.How to Define variable xn with value Rani and print it on screen

```
$ xn=Rani
$ echo $xn
```

Q.3.How to print sum of two numbers, let's say 6 and 3

```
$ echo 6 + 3
```

This will print 6 + 3, not the sum 9, To do sum or math operations in shell use expr, syntax is as follows Syntax: *expr   op1   operator   op2*
Where, op1 and op2 are any Integer Number (Number without decimal point) and operator can be
+ Addition
- Subtraction
/ Division
% Modular, to find remainder For e.g. 20 / 3 = 6 , to find remainder 20 % 3 = 2, (Remember its integer calculation)
* Multiplication

```
$ expr 6 + 3
```

Now It will print sum as 9 , But

```
$ expr 6+3
```

will not work because space is required between number and operator (See Shell Arithmetic)

Q.4.How to define two variable x=20, y=5 and then to print division of x and y (i.e. x/y)

```
$x=20
$ y=5
$ expr x / y
```

Q.5.Modify above and store division of x and y to variable called z

```
$ x=20
$ y=5
$ z=`expr x / y`
$ echo $z
```

Note : For third statement, read Shell Arithmetic.

How to write shell script

Now we write our first script that will print "Knowledge is Power" on screen. To write shell script you can use in of the Linux's text editor such as vi or mcedit or even you can use cat command. Here we are using cat command you can use any of the above text editor. First type following cat command and rest of text as its

```
$ cat > first
#
# My first shell script
#
clear
echo "Knowledge is Power"
```

Press Ctrl + D to save. Now our script is ready. To execute it type command

```
$ ./first
```

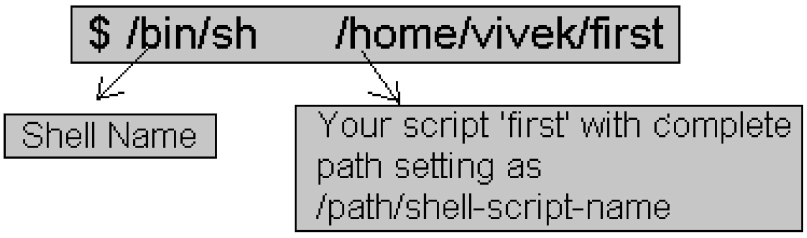
This will give error since we have not set Execute permission for our script first; to do this type command

```
$ chmod +x first
$ ./first
```

First screen will be clear, then Knowledge is Power is printed on screen. To print message of variables contains we user echo command, general form of echo command is as follows
echo "Message"
echo "Message variable1, variable2....variableN"

How to Run Shell Scripts

Because of security of files, in Linux, the creator of Shell Script does not get execution permission by default. So if we wish to run shell script we have to do two things as follows
(1) Use chmod command as follows to give execution permission to our script
Syntax: *chmod +x shell-script-name*
OR Syntax: *chmod 777 shell-script-name*
(2) Run our script as
Syntax: *./your-shell-program-name*
For e.g.
$ ./first
Here '.'(dot) is command, and used in conjunction with shell script. The dot(.) indicates to current shell that the command following the dot(.) has to be executed in the same shell i.e. without the loading of another shell in memory. Or you can also try following syntax to run Shell Script
Syntax: *bash &nbsh;&nbsh; your-shell-program-name*
OR */bin/sh &nbsh;&nbsh; your-shell-program-name*
For e.g.
$ bash first
$ /bin/sh first
Note that to run script, you need to have in same directory where you created your script, if you are in different directory your script will not run (because of path settings), For eg. Your home directory is (use $ pwd to see current working directory) /home/vivek. Then you created one script called 'first', after creation of this script you moved to some other directory lets say /home/vivek/Letters/Personal, Now if you try to execute your script it will not run, since script 'first' is in /home/vivek directory, to Overcome this problem there are two ways First, specify complete path of your script when ever you want to run it from other directories like giving following command
$ /bin/sh /home/vivek/first


$ /bin/sh /home/vivek/first
Shell Name
Your script 'first' with complete path setting as /path/shell-script-name

Now every time you have to give all this detailed as you work in other directory, this take time and you have to remember complete path. There is another way, if you notice that all of our programs (in form of executable files) are marked as executable and can be directly executed from prompt from any directory (To see executables of our normal program give command $ ls -l /bin or ls -l /usr/bin) by typing command like
$ bc
$ cc myprg.c
$ cal
etc, How this happed? All our executables files are installed in directory called /bin and /bin directory is set in your PATH setting, Now when you type name of any command at $ prompt, what shell do is it first look that command in its internal part (called as internal command, which is part of Shell itself, and always available to execute, since they do not need extra executable file), if found as internal command shell will execute it, If not found It will look for current directory, if found shell will execute command from current directory, if not found, then Shell will Look PATH setting, and try to find our requested commands executable file in all of the directories mentioned in PATH settings, if found it will execute it, otherwise it will give message "bash: xxxx :command not found", Still there is one question remain can I run my shell script same as these executables. Yes you can, for

this purpose create bin directory in your home directory and then copy your tested version of shell script to this bin directory. After this you can run you script as executable file without using $ ./shell script-name syntax, Following are steps

```
$ cd
$ mkdir bin
$ cp first ~/bin
$ first
```

Each of above command Explanation

Each of above command	Explanation
$ cd	Go to your home directory
$ mkdir bin	Now created bin directory, to install your own shell script, so that script can be run as independent program or can be accessed from any directory
$ cp first ~/bin	copy your script 'first' to your bin directory
$ first	Test whether script is running or not (It will run)

In shell script comment is given with # character. This comments are ignored by your shell. Comments are used to indicate use of script or person who creates/maintained script, or for some programming explanation etc. Remember always set *Execute permission for you script*.

Commands Related with Shell Programming

(1)*echo [options] [string, variables...]*
Displays text or variables value on screen.
Options
-n Do not output the trailing new line.
-e Enable interpretation of the following backslash escaped characters in the strings:
\a alert (bell)
\b backspace
\c suppress trailing new line
\n new line
\r carriage return
\t horizontal tab
\\ backslash
For eg. **$ echo -e "An apple a day keeps away \a\t\tdoctor\n"**

(2)*More about Quotes*
There are three types of quotes
" i.e. Double Quotes
' i.e. Single quotes
` i.e. Back quote
1."Double Quotes" - Anything enclose in double quotes removed meaning of that characters (except \ and $).
2. 'Single quotes' - Enclosed in single quotes remains unchanged.
3. `Back quote` - To execute command.
For eg.
$ echo "Today is date"
Can't print message with today's date.
$ echo "Today is `date`".
Now it will print today's date as, Today is Tue Jan ....,See the `date` statement uses back quote, (See also Shell Arithmetic NOTE).

(3) *Shell Arithmetic*

Use to perform arithmetic operations For e.g.

```
$ expr 1 + 3
$ expr 2 - 1
$ expr 10 / 2
$ expr 20 % 3 # remainder read as 20 mod 3 and remainder is 2)
$ expr 10 \* 3 # Multiplication use \* not * since its wild card)
$ echo `expr 6 + 3`
```

For the last statement not the following points
1) First, before expr keyword we used ` (back quote) sign not the (single quote i.e. ') sign. Back quote is generally found on the key under tilde (~) on PC keyboards OR To the above of TAB key.
2) Second, expr is also end with ` i.e. back quote.
3) Here expr 6 + 3 is evaluated to 9, then echo command prints 9 as sum
4) Here if you use double quote or single quote, it will NOT work, For eg.

```
$ echo "expr 6 + 3"    # It will print expr 6 + 3
$ echo 'expr 6 + 3'
```

(4) *Command Line Processing*
Now try following command (assumes that the file "grate_stories_of" is not exist on your disk)

```
$ ls grate_stories_of
```

It will print message something like -
grate_stories_of: No such file or directory
Well as it turns out ls was the name of an actual command and shell executed this command when given the command. Now it creates one question What are commands? What happened when you type $ ls grate_stories_of? The first word on command line, ls, is name of the command to be executed. Everything else on command line is taken as arguments to this command. For eg.

```
$ tail +10 myf
```

Here the name of command is tail, and the arguments are +10 and myf.
Now try to determine command and arguments from following commands:

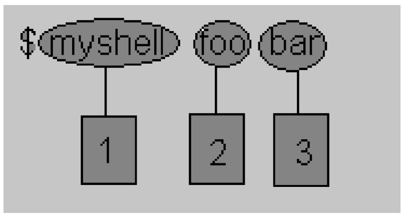
```
$ ls     foo
$ cp    y     y.bak
$ mv    y.bak     y.okay
$ tail    -10     myf
$ mail     raj
$ sort     -r    -n     myf
$ date
$ clear
```

Command	No. of argument to this command	Actual Argument
ls	1	foo
cp	2	y and y.bak
mv	2	y.bak and y.okay
tail	2	-10 and myf
mail	1	raj
sort	3	-r, -n, and myf
date	0	
clear	0	

NOTE: $# holds number of arguments specified on command line. and $* or $@ refer to all arguments in passed to script. Now to obtain total no. of Argument to particular script, your $# variable.

Why Command Line arguments required

Let's take rm command, which is used to remove file, But which file you want to remove and how you will you tail this to rm command (Even rm command does not ask you name of file that would like to remove). So what we do is we write as command as follows
$ rm   {file-name}
Here rm is command and file-name is file which you would like to remove. This way you tail to rm command which file you would like to remove. So we are doing one way communication with our command by specifying file-name. Also you can pass command line arguments to your script to make it more users friendly. But how we address or access command line argument in our script.
Lets take ls command
$ ls -a /*
This command has 2 command line argument -a and /* is another. For shell script,
$ myshell foo bar



1	Shell Script name i.e. myshell
2	First command line argument passed to myshell i.e. foo
3	Second command line argument passed to myshell i.e. bar

In shell if we wish to refer this command line argument we refer above as follows

1	myshell it is $0
2	foo it is $1
3	bar it is $2

Here $# will be 2 (Since foo and bar only two Arguments), Please note At a time such 9 arguments can be used from $0..$9, You can also refer all of them by using $* (which expand to `$0,$1,$2...$9`) Now try to write following for commands, Shell Script Name ($0), No. of Arguments (i.e. $#), And actual argument (i.e. $1,$2 etc)
$ sum   11   20
$ math   4   -   7
$ d
$ bp   -5   myf   +20
$ ls   *
$ cal
$ findBS   4   8   24   BIG

Shell Script Name	No. Of Arguments to script	Actual Argument ($1,..$9)				
$0	$#	$0	$1	$2	$3	$4
sum	2	11	20			
math	3	4	-	7		
d	0					

bp	3	-5	myf	+20		
ls	1	*				
cal	0					
findBS	4	4	8	24	BIG	

For e.g. now will write script to print command ling argument and we will see how to access them

```
$ cat > demo
#!/bin/sh
#
# Script that demos, command line args
#
echo "Total number of command line argument are $#"
echo "$0 is script name"
echo "$1 is first argument"
echo $2 is second argument"
echo "All of them are :- $*"
```

Save the above script by pressing ctrl+d, now make it executable

```
$ chmod +x demo
$ ./demo Hello World
$ cp demo ~/bin
$ demo
```

Note: After this, For any script you have to used above command, in sequence, I am not going to show you all of the above.

(5)*Exit Status*
By default in Linux if particular command is executed, it return two type of values, (Values are used to see whether command is successful or not) if return value is zero (0), command is successful, if return value is nonzero (>0), command is not successful or some sort of error executing command/shell script. This value is know as Exit Status of that command. To determine this exit Status we use $? variable of shell. For eg.

```
$ rm unknow1file
```

It will show error as follows

```
rm: cannot remove `unkowm1file': No such file or directory
```

and after that if you give command **$ echo $?**
it will print nonzero value(>0) to indicate error. Now give command

```
$ ls
$ echo $?
```

It will print 0 to indicate command is successful. Try the following commands and not down there exit status

```
$ expr 1 + 3
$ echo $?

$ echo Welcome
$ echo $?

$ wildwest canwork?
$ echo $?

$ date
$ echo $?

$ echon $?
```

$ echo $?

(6)*<u>if-then-fi for decision making is shell script</u>* Before making any decision in Shell script you must know following things Type **bc** at $ prompt to start Linux calculator program
$ bc
After this command bc is started and waiting for you commands, i.e. give it some calculation as follows type 5 + 2 as
5 + 2
7
7 is response of bc i.e. addition of 5 + 2 you can even try
5 - 2
5 / 2
Now what happened if you type 5 > 2 as follows
5 > 2
0
0 (Zero) is response of bc, How? Here it compare 5 with 2 as, Is 5 is greater then 2, (If I ask same question to you, your answer will be YES) In Linux (bc) gives this 'YES' answer by showing 0 (Zero) value. It means when ever there is any type of comparison in Linux Shell It gives only two answer one is YES and NO is other.

Linux Shell Value	Meaning	Example
Zero Value (0)	Yes/True	0
NON-ZERO Value (> 0)	No/False	-1, 32, 55 anything but not zero

Try following in bc to clear your Idea and not down bc's response
5 > 12
5 == 10
5 != 2
5 == 5
12 < 2

Expression	Meaning to us	Your Answer	BC's Response (i.e. Linux Shell representation in zero & non-zero value)
5 > 12	Is 5 greater than 12	NO	0
5 == 10	Is 5 is equal to 10	NO	0
5 != 2	Is 5 is NOT equal to 2	YES	1
5 == 5	Is 5 is equal to 5	YES	1
1 < 2	Is 1 is less than 2	Yes	1

Now will see, if condition which is used for decision making in shell script, If given condition is true then command1 is executed.
<u>Syntax:</u>

```
    if   condition
    then
            command1 if condition is true or if exit status
            of condition is 0 (zero)
            ...
            ...
    fi
```

Here *condition* is nothing but comparison between two values, for compression we can use test or [expr] statements or even exist status can be also used. An *expression* is nothing but combination of values, relational operator (such as >,<, <> etc) and mathematical operators (such as +, -, / etc).
Following are all examples of expression:
5 > 2
3 + 6
3 * 65
a < b
c > 5
c > 5 + 30 -1
Type following command (assumes you have file called foo)

```
$ cat foo
$ echo $?
```

The cat command return zero(0) on successful, this can be used in if condition as follows, Write shell script as

```
$ cat > showfile
#!/bin/sh
#
#Script to print file
#
if cat $1
then
    echo -e "\n\nFile $1, found and successfully echoed"
fi
```

Now run it.

```
$ chmod +x showfile
$./showfile foo
```

Here

```
$ ./showfile   foo
```

Our shell script name is showfile($0) and foo is argument (which is $1).Now we compare as follows
if cat $1 (i.e. if cat foo)
Now if cat command finds foo file and if its successfully shown on screen, it means our cat command is successful and its exist status is 0 (indicates success) So our if condition is also true and hence statement echo -e "\n\nFile $1, found and successfully echoed" is proceed by shell. Now if cat command is not successful then it returns non-zero value (indicates some sort of failure) and this statement echo -e "\n\nFile $1, found and successfully echoed" is skipped by our shell.
Now try to write answer for following
1) Create following script

```
cat > trmif
#
# Script to test rm command and exist status
#
if rm $1
then
    echo "$1 file deleted"
fi
```

(Press Ctrl + d to save)

```
$ chmod +x trmif
```

Now answer the following
A) There is file called foo, on your disk and you give command, **$ ./trmfi   foo** what will be output.

B) If bar file not present on your disk and you give command, $ **./trmfi bar** what will be output.
C) And if you type **$ ./trmfi**, What will be output.

(7)*test command or [expr]*
test command or [expr] is used to see if an expression is true, and if it is true it return zero(0), otherwise returns nonzero(>0) for false. Syntax: *test expression* OR *[expression]*
Now will write script that determine whether given argument number is positive. Write script as follows

```
$ cat > ispostive
#!/bin/sh
#
# Script to see whether argument is positive
#
if test $1 -gt 0
then
    echo "$1 number is positive"
fi
```

Run it as follows

```
$ chmod +x ispostive
$ ispostive 5
```

Here o/p : 5 number is positive

```
$ispostive -45
```

Here o/p : Nothing is printed

```
$ispostive
```

Here o/p : ./ispostive: test: -gt: unary operator expected
The line, if test $1 -gt 0 , test to see if first command line argument($1) is greater than 0. If it is true(0) then test will return 0 and output will printed as 5 number is positive but for -45 argument there is no output because our condition is not true(0) (no -45 is not greater than 0) hence echo statement is skipped. And for last statement we have not supplied any argument hence error ./ispostive: test: -gt: unary operator expected is generated by shell , to avoid such error we can test whether command line argument is supplied or not. (See command 8 Script example). test or [expr] works with
1.Integer (Number without decimal point)
2.File types
3.Character strings
For Mathematics use following operator in Shell Script

Mathematical Operator in Shell Script	Meaning	Normal Arithmetical/ Mathematical Statements	But in Shell	
			For test statement with if command	For [expr] statement with if command
-eq	is equal to	5 == 6	if test 5 -eq 6	if expr [5 -eq 6]
-ne	is not equal to	5 != 6	if test 5 -ne 6	if expr [5 -ne 6]
-lt	is less than	5 < 6	if test 5 -lt 6	if expr [5 -lt 6]
-le	is less than or equal to	5 <= 6	if test 5 -le 6	if expr [5 -le 6]
-gt	is greater than	5 > 6	if test 5 -gt 6	if expr [5 -gt 6]
-ge	is greater than or equal to	5 >= 6	if test 5 -ge 6	if expr [5 -ge 6]

NOTE: == is equal, != is not equal.
For string Comparisons use

Operator	Meaning
string1 = string2	string1 is equal to string2
string1 != string2	string1 is NOT equal to string2
string1	string1 is NOT NULL or not defined
-n string1	string1 is NOT NULL and does exist
-z string1	string1 is NULL and does exist

Shell also test for file and directory types

Test	Meaning
-s file	Non empty file
-f file	Is File exist or normal file and not a directory
-d dir	Is Directory exist and not a file
-w file	Is writeable file
-r file	Is read-only file
-x file	Is file is executable

Logical Operators
Logical operators are used to combine two or more condition at a time

Operator	Meaning
! expression	Logical NOT
expression1 -a expression2	Logical AND
expression1 -o expression2	Logical OR

(8)*if...else...fi*
If given condition is true then command1 is executed otherwise command2 is executed.
Syntax:

```
if  condition
then
          command1 if condition is true or if exit status
          of condition is 0(zero)
          ...
          ...
else
          command2 if condition is false or if exit status
          of condition is >0 (nonzero)
           ...
           ...
fi
```

For eg. Write Script as follows
$ cat > isnump_n
#!/bin/sh
#

```
# Script to see whether argument is positive or negative
#
if [ $# -eq 0 ]
then
    echo "$0 : You must give/supply one integers"
    exit 1
fi


if test $1 -gt 0
then
    echo "$1 number is positive"
else
    echo "$1 number is negative"
fi
```

Try it as follows

```
$ chmod +x isnump_n
$ isnump_n 5
Here o/p : 5 number is positive
$ isnump_n -45
Here o/p : -45 number is negative
$ isnump_n
Here o/p : ./ispos_n : You must give/supply one integers
$ isnump_n 0
Here o/p : 0 number is negative
```

Here first we see if no command line argument is given then it print error message as "./ispos_n : You must give/supply one integers". if statement checks whether number of argument ($#) passed to script is not equal (-eq) to 0, if we passed any argument to script then this if statement is false and if no command line argument is given then this if statement is true. The echo command i.e.

```
echo "$0 : You must give/supply one integers"
       |           |
       |           |
       1           2
```

1 will print Name of script
2 will print this error message
And finally statement exit 1 causes normal program termination with exit status 1 (nonzero means script is not successfully run), The last sample run $ isnump_n 0 , gives output as "0 number is negative", because given argument is not > 0, hence condition is false and it's taken as negative number. To avoid this replace second if statement with if test $1 -ge 0.

(9)*Multilevel if-then-else*
Syntax:

```
            if condition
            then
                        condition is zero (true - 0)
                        execute all commands up to elif statement
            elif condition1
                        condition1 is zero (true - 0)
                        execute all commands up to elif statement
            elif condition2
                        condition2 is zero (true - 0)
                        execute all commands up to elif statement
```

```
        else
                    None of the above condtion,condtion1,condtion2 are true (i.e.
                    all of the above nonzero or false)
                    execute all commands up to fi
        fi
```

For e.g. Write script as **$ cat > elf #!/bin/sh # # Script to test if..elif...else # # if [$1 -gt 0] then echo "$1 is positive" elif [$1 -lt 0] then echo "$1 is negative" elif [$1 -eq 0] then echo "$1 is zero" else echo "Opps! $1 is not number, give number" fi** Try above script with **$ chmod +x elf $ ./elf 1 $ ./elf -2 $ ./elf 0 $ ./elf a** Here o/p for last sample run: ./elf: [: -gt: unary operator expected ./elf: [: -lt: unary operator expected ./elf: [: -eq: unary operator expected Opps! a is not number, give number Above program gives error for last run, here integer comparison is expected therefore error like "./elf: [: -gt: unary operator expected" occurs, but still our program notify this thing to user by providing message "Opps! a is not number, give number".

(10)*Loops in Shell Scripts*
Computer can repeat particular instruction again and again, until particular condition satisfies. A group of instruction that is executed repeatedly is called a loop.
(a) **for loop** Syntax:

```
        for { variable name } in { list }
        do
                execute one for each item in the list until the list is
                not finished (And repeat all statement between do and done)
        done
```

Suppose,

```
$ cat > testfor
for i in 1 2 3 4 5
do
    echo "Welcome $i times"
done
```

Run it as,

```
$ chmod +x testfor
$ ./testfor
```

The for loop first creates i variable and assigned a number to i from the list of number from 1 to 5, The shell execute echo statement for each assignment of i. (This is usually know as iteration) This process will continue until all the items in the list were not finished, because of this it will repeat 5 echo statements. for e.g. Now try script as follows

```
$ cat > mtable
#!/bin/sh
#
#Script to test for loop
if [ $# -eq 0 ]
then
   echo "Error - Number missing form command line argument"
   echo "Syntax : $0 number"
   echo " Use to print multiplication table for given number"
   exit 1
fi
n=$1
for i in 1 2 3 4 5 6 7 8 9 10
```

```
do
    echo "$n * $i = `expr $i \* $n`"
done
```

Save and Run it as

```
$ chmod +x mtable
$ ./mtable 7
$ ./mtable
```

For first run, Above program print multiplication table of given number where i = 1,2 ... 10 is multiply by given n (here command line argument 7) in order to produce multiplication table as

```
7 * 1 = 7
7 * 2 = 14
...
..
7 * 10 = 70
```

And for Second run, it will print message -

```
Error - Number missing form command line argument
Syntax : ./mtable number
          Use to print multiplication table for given number
```

This happened because we have not supplied given number for which we want multiplication table, Hence we are showing Error message, Syntax and usage of our script. This is good idea if our program takes some argument, let the user know what is use of this script and how to used it. Note that to terminate our script we used 'exit 1' command which takes 1 as argument (1Indicates error and therefore script is terminated)

(b)**while loop**

Syntax:

```
while [ condition ]
do
      command1
      command2
      command3
      ..
      ....
 done
```

Loop is executed as long as given condition is true. For eg. Above for loop program can be written using while loop as

```
$cat > nt1
#!/bin/sh
#
#Script to test while statement
#
#
if [ $# -eq 0 ]
then
   echo "Error - Number missing form command line argument"
   echo "Syntax : $0 number"
   echo " Use to print multiplication table for given number"
   exit 1
fi
n=$1
i=1
while [ $i -le 10 ]
```

```
do
   echo "$n * $i = `expr $i \* $n`"
   i=`expr $i + 1`
done
```

Save it and try as

$ chmod +x nt1
$./nt1 7

Above loop can be explained as follows

n=$1	Set the value of command line argument to variable n. (Here it's set to 7)
i=1	Set variable i to 1
while [$i -le 10]	This is our loop condition, here if value of i is less than 10 then, shell execute all statements between do and done
do	Start loop
echo "$n * $i = `expr $i  \*  $n`"	Print multiplication table as 7 * 1 = 7 7 * 2 = 14 .... 7 * 10 = 70, Here each time value of variable n is multiply be i.
i=`expr $i + 1`	Increment i by 1 and store result to i. (i.e. i=i+1) Caution: If we ignore (remove) this statement than our loop become infinite loop because value of variable i always remain less than 10 and program will only output 7 * 1 = 7 ... ... E (infinite times)
done	Loop stops here if i is not less than 10 i.e. condition of loop is not true. Hence loop is terminated.

From the above discussion not following points about loops
(a) First, the variable used in loop condition must be initialized, Next execution of the loop begins.
(b) A test (condition) is made at the beginning of each iteration.
(c) The body of loop ends with a statement that modifies the value of the test (condition) variable.

(11) *The case Statement*

The case statement is good alternative to Multilevel if-then-else-fi statement. It enable you to match several values against one variable. Its easier to read and write.

Syntax:

```
           case  $variable-name  in
                pattern1)   command
                                ...
                                ..
                                command;;
                pattern2)   command
```

```
                              ...
                              ..
                              command;;
               patternN)   command
                              ...
                              ..
                              command;;
               *)             command
                              ...
                              ..
                              command;;
          esac
```

The $variable-name is compared against the patterns until a match is found. The shell then executes all the statements up to the two semicolons that are next to each other. The default is *) and its executed if no match is found. For eg. Create script as follows

```
$ cat > car
#
# if no vehicle name is given
# i.e. -z $1 is defined and it is NULL
#
# if no command line arg

if [ -z $1 ]
then
   rental="*** Unknown vehicle ***"
elif [ -n $1 ]
then
# otherwise make first arg as rental
   rental=$1
fi

case $rental in
   "car") echo "For $rental Rs.20 per k/m";;
   "van") echo "For $rental Rs.10 per k/m";;
   "jeep") echo "For $rental Rs.5 per k/m";;
   "bicycle") echo "For $rental 20 paisa per k/m";;
   *) echo "Sorry, I can not gat a $rental for you";;
esac
```

Save it by pressing CTRL+D

```
$ chmod +x car
$ car van
$ car car
$ car Maruti-800
```

Here first we will check, that if $1(first command line argument) is not given set value of rental variable to "*** Unknown vehicle ***",if value given then set it to given value. The $rental is compared against the patterns until a match is found. Here for first run its match with van and it will show output For van Rs.10 per k/m. For second run it print, "For car Rs.20 per k/m". And for last run, there is no match for Maruti-800, hence default i.e. *) is executed and it prints, "Sorry, I can not gat a Maruti-800 for you". Note that esac is always required to indicate end of case statement.

(12)*The read Statement*

Use to get input from keyboard and store them to variable.

Syntax: *read varible1, varible2,...varibleN*
Create script as

```
$ cat > sayH
#
#Script to read your name from key-board
#
echo "Your first name please:"
read fname
echo "Hello $fname, Lets be friend!"
```

Run it as follows

```
$ chmod +x sayH
$ ./sayH
```

This script first ask you your name and then waits to enter name from the user, Then user enters name from keyboard (After giving name you have to press ENTER key) and this entered name through keyboard is stored (assigned) to variable fname.

(13)*Filename Shorthand or meta Characters (i.e. wild cards)*
* or ? or [...] is one of such shorthand character.

* Matches any string or group of characters.
For e.g. **$ ls *** , will show all files, **$ ls a*** - will show all files whose first name is starting with letter 'a', **$ ls *.c** ,will show all files having extension .c **$ ls ut*.c**, will show all files having extension .c but first two letters of file name must be 'ut'.

? Matches any single character.
For e.g. **$ ls ?** , will show one single letter file name, **$ ls fo?** , will show all files whose names are 3 character long and file name begin with fo

[...] Matches any one of the enclosed characters.
For e.g. **$ ls [abc]*** - will show all files beginning with letters a,b,c
[..-..] A pair of characters separated by a minus sign denotes a range;
For eg. $ ls /bin/[a-c]* - will show all files name beginning with letter a,b or c like

```
   /bin/arch          /bin/awk           /bin/bsh     /bin/chmod          /bin/cp
   /bin/ash           /bin/basename      /bin/cat     /bin/chown          /bin/cpio
   /bin/ash.static    /bin/bash          /bin/chgrp   /bin/consolechars   /bin/csh
```

But
$ ls /bin/[!a-o]
$ ls /bin/[^a-o]
If the first character following the [is a ! or a ^ then any character not enclosed is matched i.e. do not show us file name that beginning with a,b,c,e...o, like

```
   /bin/ps            /bin/rvi          /bin/sleep  /bin/touch     /bin/view
   /bin/pwd           /bin/rview        /bin/sort   /bin/true      /bin/wcomp
   /bin/red           /bin/sayHello     /bin/stty   /bin/umount    /bin/xconf
   /bin/remadmin      /bin/sed          /bin/su     /bin/uname     /bin/ypdomainname
   /bin/rm            /bin/setserial    /bin/sync   /bin/userconf  /bin/zcat
   /bin/rmdir         /bin/sfxload      /bin/tar    /bin/usleep
   /bin/rpm           /bin/sh           /bin/tcsh   /bin/vi
```

(14)*command1;command2*
To run two command with one command line.For eg. **$ date;who** ,Will print today's date followed

by users who are currently login. Note that You can't use $ date who for same purpose, you must put semicolon in between date and who command.

More Advanced Shell Script Commands

/dev/null - Use to send unwanted output of program

This is special Linux file which is used to send any unwanted output from program/command.
<u>Syntax:</u> *command > /dev/null*
For e.g. **$ ls > /dev/null** , output of this command is not shown on screen its send to this special file. The /dev directory contains other device files. The files in this directory mostly represent peripheral devices such disks liks floppy disk, sound card, line printers etc.

Local and Global Shell variable (export command)

Normally all our variables are local. Local variable can be used in same shell, if you load another copy of shell (by typing the /bin/bash at the $ prompt) then new shell ignored all old shell's variable. For e.g. Consider following example

```
$ vech=Bus
$ echo $vech
Bus
$ /bin/bash
$ echo $vech

NOTE:-Empty line printed
$ vech=Car
$ echo $vech
Car
$ exit
$ echo $vech
Bus
```

Command	Meaning
$ vech=Bus	Create new local variable 'vech' with Bus as value in first shell
$ echo $vech	Print the contains of variable vech
$ /bin/bash	Now load second shell in memory (Which ignores all old shell's variable)
$ echo $vech	Print the contains of variable vech
$ vech=Car	Create new local variable 'vech' with Car as value in second shell
$ echo $vech	Print the contains of variable vech
$ exit	Exit from second shell return to first shell
$ echo $vech	Print the contains of variable vech (Now you can see first shells variable and its value)

We can copy old shell's variable to new shell (i.e. first shells variable to seconds shell), such variable is know as Global Shell variable. To do this use export command
<u>Syntax:</u> *export variable1, variable2,.....variableN*
For e.g.

```
$ vech=Bus
$ echo $vech
Bus
$ export vech
$ /bin/bash
$ echo $vech
Bus
$ exit
$ echo $vech
```

Bus

Command	Meaning
$ vech=Bus	Create new local variable 'vech' with Bus as value in first shell
$ echo $vech	Print the contains of variable vech
$ export vech	Export first shells variable to second shell
$ /bin/bash	Now load second shell in memory (Old shell's variable is accessed from second shell, *if they are exported*)
$ echo $vech	Print the contains of variable vech
$ exit	Exit from second shell return to first shell
$ echo $vech	Print the contains of variable vech

Conditional execution i.e. && and ||

The control operators are && (read as AND) and || (read as OR). An AND list has the
Syntax: command1 && command2
Here command2 is executed if, and only if, command1 returns an exit status of zero. An OR list has the
Syntax: command1 || command2
Here command2 is executed if and only if command1 returns a non-zero exit status. You can use both as follows
command1 && comamnd2 if exist status is zero || command3 if exit status is non-zero
Here if command1 is executed successfully then shell will run command2 and if command1 is not successful then command3 is executed. For e.g.

```
$ rm myf && echo File is removed successfully || echo File is not removed
```

If file (myf) is removed successful (exist status is zero) then "echo File is removed successfully" statement is executed, otherwise "echo File is not removed" statement is executed (since exist status is non-zero)

I/O Redirection and file descriptors

As you know I/O redirectors are used to send output of command to file or to read input from file. (See Input/Output redirection). Now consider following examples

```
$ cat > myf
    This is my file
    ^D
```

Above command send output of cat command to myf file

```
$ cal
```

Above command prints calendar on screen, but if you wish to store this calendar to file then give command

```
$ cal > mycal
```

The cal command send output to mycal file. This is called output redirection

```
$ sort
10
-20
11
2
^D
-20
2
10
11
```

Here sort command takes input from keyboard and then sorts the number, If we wish to take input from file give command as follows

```
$ cat > nos
10
```

```
-20
11
2
^D
$ sort < nos
-20
2
10
11
```

First we have created the file nos, then we have taken input from this file and sort command prints sorted numbers. This is called input redirection. In Linux (And in C programming Language) your keyboard, screen etc are treated as files. Following are name of such files

Standard File	File Descriptors number	Use	Example
stdin	0	as Standard input	Keyboard
stdout	1	as Standard output	Screen
stderr	2	as Standard error	Screen

By default in Linux every program has three files associated with it, (when we start our program these three files are automatically opened by your shell) The use of first two files (i.e. stdin and stdout) , are already seen by us. The last file stderr (numbered as 2) is used by our program to print error on screen. You can redirect the output from a file descriptor directly to file with following
Syntax: file-descriptor-number>filename
For e.g.
$ rm bad_file_name111
rm: cannot remove `bad_file_name111': No such file or directory ,is the output (error) of the above program. Now if we try to redirect this error-output to file, it can not be send to file
$ rm bad_file_name111 > er
Still it prints output on stderr as rm: cannot remove `bad_file_name111': No such file or directory, And if you see er file as $ cat er , This file is empty, since output is send to error device and you can not redirect it to copy this error-output to your file 'er'. To overcome this we have to use following command
$ rm bad_file_name111 2>er
Note that no space are allowed between 2 and >, The 2>er directs the standard error output to file. 2 number is default number of stderr file. Now consider another example, here we are writing shell script as follows

```
$ cat > demoscr
if [ $# -ne 2 ]
then
echo "Error : Number are not supplied"
echo "Usage : $0 number1 number2"
exit 1
fi
ans=`expr $1 + $2`
echo "Sum is $ans"
```

Try it as follows

```
$ chmod +x demoscr
$ ./demoscr
Error : Number are not supplied
Usage : ./demoscr number1 number2
$ ./demoscr > er1
$ ./demoscr 5 7
Sum is 12
```

Here for first sample run , our script prints error message indicating that we have not given two number.
For second sample run, we have redirect output of our script to file, since it's error we have to show it to

user, It means we have to print our error message on stderr not on stdout. To overcome this problem replace above echo statements as follows

```
echo "Error : Number are not supplied" 1>&2
echo "Usage : $0 number1 number2" 1>&2
```

Now if you run as

```
$ ./demoscr > er1
Error : Number are not supplied
Usage : ./demoscr number1 number2
```

It will print error message on stderr and not on stdout. The 1>&2 at the end of echo statement, directs the standard output (stdout) to standard error (stderr) device.

Syntax: *from>&destination*

Functions

Function is series of instruction/commands. Function performs particular activity in shell. To define function use following

Syntax:

```
function-name ( )
{
     command1
     command2
     .....
     ...
     commandN
     return
}
```

Where function-name is name of you function, that executes these commands. A return statement will terminate the function. For e.g. Type SayHello() at $ prompt as follows

```
$ SayHello()
{
echo "Hello $LOGNAME, Have nice computing"
return
}
```

Now to execute this SayHello() function just type it name as follows

```
$ SayHello
Hello xxxxx, Have nice computing
```

This way you can call your function. Note that after restarting your computer you will loss this SayHello() function, since its created for that session only. To overcome this problem and to add you own function to automat some of the day today life task, your function to /etc/bashrc file. Note that to add function to this file you must logon as root. Following is the sample /etc/bashrc file with today() function , which is used to print formatted date. First logon as root or if you already logon with your name (your login is not root), and want to move to root account, then you can type following command , when asked for password type root (administrators) password

```
$ su
password:
```

Now open file as (Note your prompt is changed to # from $ to indicate you are root)

```
# vi /etc/bashrc
OR
# mcedit /etc/bashrc
```

At the end of file add following in /etc/bashrc file

```
#
# today() to print formatted date
#
# To run this function type today at the $ prompt
# Added by Vivek to show function in Linux
#
```

```
today()
{
echo This is a `date +"%A %d in %B of %Y (%r)"`
return
}
```

Save the file and exit it, after all this modification your file may look like as follows

```
# /etc/bashrc

# System wide functions and aliases
# Environment stuff goes in /etc/profile

# For some unknown reason bash refuses to inherit
# PS1 in some circumstances that I can't figure out.
# Putting PS1 here ensures that it gets loaded every time.

PS1="[\u@\h \W]\\$ "

#
# today() to print formatted date
#
# To run this function type today at the $ prompt
# Added by Vivek to show function in Linux
today()
{
echo This is a `date +"%A %d in %B of %Y (%r)"`
return
}
```

To run function first completely logout by typing exit at the $prompt (Or press CTRL + D, Note you may have to type exit (CTRL +D) twice if you login to root account by using su command) ,then login and type **$ today** , this way today() is available to all user in your system, If you want to add particular function to particular user then open .bashrc file in your home directory as follows

```
# vi .bashrc
OR
# mcedit .bashrc
```

At the end of file add following in .bashrc file

```
SayBuy()
{
   echo "Buy $LOGNAME ! Life never be the same, until you log again!"
   echo "Press a key to logout. . ."
   read
   return
}
```

Save the file and exit it, after all this modification your file may look like as follows

```
# .bashrc
#
# User specific aliases and functions
# Source global definitions

if [ -f /etc/bashrc ]; then
. /etc/bashrc
fi

SayBuy()
```

```
{
   echo "Buy $LOGNAME ! Life never be the same, until you log again!"
   echo "Press a key to logout. . ."
   read
   return
}
```

To run function first logout by typing exit at the $ prompt (Or press CTRL + D) ,then logon and type **$ SayBuy** , this way SayBuy() is available to only in your login and not to all user in system, Use .bashrc file in your home directory to add User specific aliases and functions only. (Tip: If you want to show some message or want to perform some action when you logout, Open file .bash_logout in your home directory and add your stuff here For e.g. When ever I logout, I want to show message Buy! Then open your .bash_logout file using text editor such as vi and add statement

```
echo "Buy $LOGNAME, Press a key. . ."
read
```

Save and exit from the file. Then to test this logout from your system by pressing CTRL + D (or type **exit**) immediately you will see message "Buy xxxxx, Press a key. . .", after pressing key you will be exited.)

User Interface

Good program/shell script must interact with users. There are two ways to this one is use command line to script when you want input, second use statement like echo and read to read input into variable from the prompt. For e.g. Write script as

```
$ cat > userinte
#
# Script to demo echo and read command for user interaction
#
echo "Your good name please :"
read na
echo "Your age please :"
read age
neyr=`expr $age + 1`
echo "Hello $na, next year you will be $neyr yrs old."
```

Save it and run as

```
$ chmod +x userinte
$ ./userinte
```

```
Your good name please :
Vivek
Your age please :
25
Hello Vivek, next year you will be 26 yrs old.
```

Even you can create menus to interact with user, first show menu option, then ask user to choose menu item, and take appropriate action according to selected menu item, this technique is show in following script

```
$ cat > menuui
#
# Script to create simple menus and take action according to that selected
# menu item
#
while :
do
     clear
     echo "-------------------------------------"
     echo " Main Menu "
     echo "-------------------------------------"
     echo "[1] Show Todays date/time"
     echo "[2] Show files in current directory"
```

```
    echo "[3] Show calendar"
    echo "[4] Start editor to write letters"
    echo "[5] Exit/Stop"
    echo "======================="
    echo -n "Enter your menu choice [1-5]: "
    read yourch
   case $yourch in
       1) echo "Today is `date` , press a key. . ." ; read ;;
       2) echo "Files in `pwd`" ; ls -l ; echo "Press a key. . ." ; read ;;
       3) cal ; echo "Press a key. . ." ; read ;;
       4) vi ;;
       5) exit 0 ;;
       *) echo "Opps!!! Please select choice 1,2,3,4, or 5";
         echo "Press a key. . ." ; read ;;
   esca

done
```

Above all statement explained in following table

Statement	Explanation
while :	Start infinite loop, this loop will only break if you select 5 (i.e. Exit/Stop menu item) as your menu choice
do	Start loop
clear	Clear the screen, each and every time
echo "-------------------------------------" **echo " Main Menu "** **echo "-------------------------------------"** **echo "[1] Show Todays date/time"** **echo "[2] Show files in current directory"** **echo "[3] Show calendar"** **echo "[4] Start editor to write letters"** **echo "[5] Exit/Stop"** **echo "======================="**	Show menu on screen with menu items
echo -n "Enter your menu choice [1-5]: "	Ask user to enter menu item number
read yourch	Read menu item number from user
case $yourch in **1) echo "Today is `date` , press a key. . ." ; read ;;** **2) echo "Files in `pwd`" ; ls -l ;** **echo "Press a key. . ." ; read ;;** **3) cal ; echo "Press a key. . ." ; read ;;** **4) vi ;;** **5) exit 0 ;;** ***) echo "Opps!!! Please select choice 1,2,3,4, or 5";** **echo "Press a key. . ." ; read ;;** **esca**	Take appropriate action according to selected menu item, If menu item is not between 1 - 5, then show error and ask user to input number between 1-5 again
done	Stop loop , if menu item number is 5 (i.e. Exit/Stop)