
# SHELL SCRIPTING
# REFERENCE CUM LAB MANUAL

***Version 3.0***

***By***

***Musabuddin Syed (Redhat Certified)***

VirtualPath Techno Solutions
#101, Namdev Block, Balaji Towers
Prime Hospital Lane, Ameerpet Hyd-TS
Contact: +91 799 309 6092, 040-6666 6092
Email: info@virtualpathtech.com
Website: www.virtualpathtech.com, www.musab.in

# Table of Contents

# INTRODUCTION

**What's Kernel**

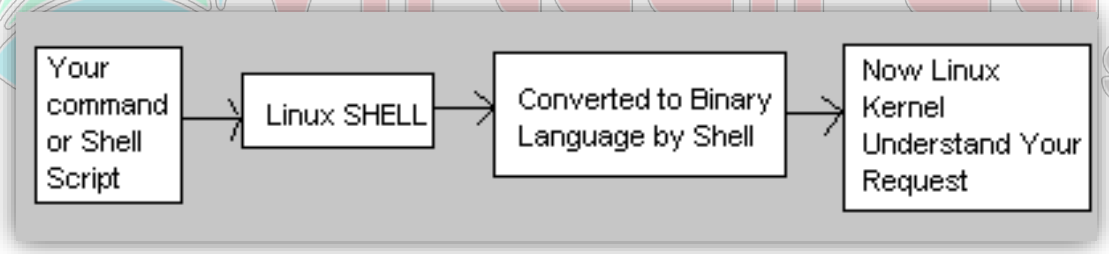
Kernel is the heart of Linux O/S. It manages resource of Linux O/S. Resources means facilities available in Linux. For e.g. Facility to store data, print data on printer, memory, file management etc. Kernel decides who will use this resource, for how long and when. It runs your programs (or set up to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

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

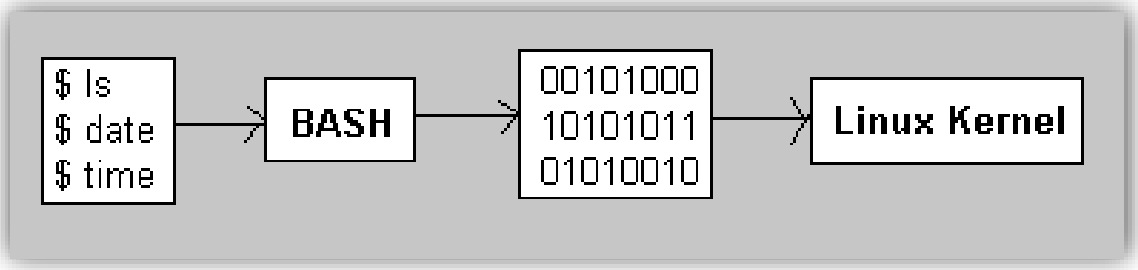
**What's Linux Shell?**

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 computer's native binary language.

**This is what Shell Does for US**




**You type Your command and shell convert it as**




- → Its environment provided for user interaction.
- → Shell is a command language interpreter that executes commands read from the standard input device (keyboard) or from a file.

**Types of Shells in Linux:**

- → Shell is not part of the system kernel, but it uses system kernel to execute the programs.
- → There are 13 types of shells widely used across, Among them SH, BASH, KSH are more popular

| TYPES OF SHELLS | DEVELOPED BY | WHERE |
|---|---|---|
| BSH/ SH (Bourne SHell) | Stephen Bourne | AT & T Bell Labs |
| BASH ( Bourne-Again SHell ) | Brian Fox and Chet Ramey | Free Software Foundation |
| CSH (C SHell) | Bill Joy | University of California (For BSD) |
| KSH (Korn SHell) | David Korn | AT & T Bell Labs |

**Accessing a shell in Linux:**

As soon as we login into the system successfully, we would be landing into the home directory and by default we would be given a program that would be running in the background call shell. To use a shell you just need to login into the system and you would be able to access it.



**What is a Shell Script?**

Shell scripting is writing a series of command for the shell to execute. It can combine lengthy and repetitive sequences of commands into a single and simple script, which can be stored and executed anytime. This reduces the effort required by the end user.

**Why Shell Scripting?**

- → Shell script can take input from user, file and output them on screen.
- → Useful to create our own commands.
- → Save lots of time.
- → To automate some task of day today life.
- → System Administration part can be also automated.

**Limitation of Shell Script:**

- → Compatibility problems between different platforms.
- → Slow execution speed.
- → A new process launched for almost every shell command executed.

**Creating a Shell Script:**

→ To write shell script we can use any of the Linux's text editor such as vi, vim, nano, emacs etc., even you can use cat command.

→ It is preferable to use vi or vim editor as it is widely used and known by many Linux admins.

**Basics of Unix/Linux:**

→ Before we can go to shell scripting, let's review some important basics of Unix/Linux administration.

**Getting help in Unix/Linux:**

To get help in Unix/Linux command line, we can make use of following commands after command name.

→ **help**

→ **man**

→ **info**

**File Types supported in Unix/Linux:**

→ UNIX Supports variety of files among them the basic types of files are.

| Character | File Type |
|---|---|
| - | Regular file. |
| l | Symbolic Link |
| c | Character Special |
| b | Block Special |
| p | Named Pipe |
| s | Socket file. |
| d | Directory |

→ To determine the type of the file we can get it from ls –l command. The first character defines the type.

```
[root@mlinux7 ~]# ls -l myfile
-rw-r--r--. 1 root root 12 Oct  1 19:00 myfile
[root@mlinux7 ~]#
```

## File Permissions:

**Permissions are applied on three levels:**

- Owner or User level
- Group level
- Others level

**Access modes are of three types:**

- **r** read only
- **w** write/edit/delete/append
- **x** execute/run a command

**Access modes are different on file and directory:**

| Permissions | Files | Directory |
|---|---|---|
| R | Open the file | 'ls'/list the contents of directory |
| W | Write, edit, append, delete file | Add/Del/Rename contents of directory |
| X | To run a command/shell script | To enter into directory using 'cd' |

```
[root@musab1 ~]# ls -l myfile
-rw-r--r--. 2 root root 0 Feb 13 13:55 myfile
[root@musab1 ~]# ls -ld mydir
drwxr-xr-x. 2 root root 4096 Feb 13 16:43 mydir
```

**Filetype+permission, links, owner, group name of owner, size in bytes, date of modification, file name**

→ To change the ownership permissions of file using ***chown*** command and this command can only be executed by root user.

→ To change the file actions permissions of a file using ***chmod*** command and this can be given by the owner of the file.

## Let's Get Started:

### Shebang:

The **#!** Syntax used in scripts to indicate an interpreter for execution under UNIX / Linux operating systems. Most Linux shell and perl / python script starts with the following line:

**#!/bin/bash**
**OR**
**#!/usr/bin/perl**
**OR**
**#!/usr/bin/python**

- → It is called a shebang or a "bang" line.
- → It is nothing but the absolute path to the Bash interpreter (shell).
- → It consists of a number sign and an exclamation point character (#!), followed by the full path to the interpreter such as /bin/bash.
- → All scripts under Linux execute using the interpreter specified on a first line.
- → Almost all bash scripts often begin with #!/bin/bash (assuming that Bash has been installed in /bin)
- → This ensures that Bash will be used to interpret the script, even if it is executed under another shell.
- → The shebang was introduced by Dennis Ritchie between Version 7 Unix and 8 at Bell Laboratories. It was then also added to the BSD line at Berkeley
- → If no shebang interpreter line is added, then it would consider **/bin/sh** as default. It is recommended to give **#!/bin/bash** for best results.



***Do you know?***

- → *In musical notation, a **"#"** is called a **sharp** and an exclamation point - **"!"** - is sometimes referred to as a **bang**. Thus, **shebang** becomes a shortening of **sharp-bang**. The term is mentioned in Elizabeth Castro's Perl and CGI for the World Wide Web.*

### Comment:

- → You should be aware of the fact that you might not be the only person reading your code. A lot of users and system administrators run scripts that were written by other people. If they want to see how you did it, comments are useful to enlighten the reader.

- → Comments also make your own life easier. Say that you had to read a lot of man pages in order to achieve a particular result with some command that you used in your script. You won't remember how it worked if you need to change your script after a few weeks or months, unless you have commented what you did, how you did it and/or why you did it.

```
#!/bin/bash
echo "Hello World" #printing a message --> this is a comment
```

**Executing a Shell Script:**

→ The most simplest and the easiest way to run a script in Unix/Linux is as follows

```
[root@mlinux71 ~]# sh script1.sh
Or
[root@mlinux71 ~]# bash script1.sh
```

→ Another widely used method is;

```
[root@mlinux71 ~]# ./script1.sh
```

→ *Do you know that almost 90% of first time writers of shell script will get below error*

```
[root@mlinux71 ~]# ./script1.sh
-bash: ./script1.sh: Permission denied
[root@mlinux71 ~]#
```

*This error is because every script needs execute permission which might be missing*

So, let's first apply the execute "x" permission on the file

```
[root@mlinux71 ~]# chmod +x script1.sh
[root@mlinux71 ~]# ls -l script1.sh
-rwxr-xr-x. 1 root root 73 Oct  3 18:51 script1.sh
[root@mlinux71 ~]#
```



## ECHO COMMAND

The built-in ***echo*** command is an older form of ***printf*** command in Linux/Unix systems.
It used to display the text or variables on the output screen. The following the example narrates the usage of echo command.

```
[root@mlinux71 ~]# cat script1.sh
#!/bin/bash
echo WELCOME TO SHELL SCRIPTING
```

**Output**

```
[root@mlinux71 ~]# ./script1.sh
WELCOME TO SHELL SCRIPTING
```

**Options:**

→ **-n: (No New Line) a line field is automatically added after the string is displayed, It can be suppressed with the - n option.**

→ **-e : (enable escape sequence) If the - e option is enabled then echo command will enable escape sequences**

**\n - New line escape sequence.**

**\t - New horizontal tap escape sequence.**

**ECHO -n Option:**

- → ECHO Command is used to print the text on the output screen and by default it creates a new line feed automatically after the string is displayed.
- → We can suppress ·t using -n option and the following commands will show the difference in using -n option.

***Without –n Option:***

```
[root@mlinux71 ~]# echo hello world
hello world
[root@mlinux71 ~]#
```

***With –n Option:***

```
[root@mlinux71 ~]# echo -n hello world
hello world[root@mlinux71 ~]#
```

In the above example you can see the command prompt is in the same line instead of new line if we are using -n option.

**Echo –e option:**

- → Option -e stands for enabling escape sequence in ***echo*** command. In some cases you may require to print multiple lines of output using ***echo*** command. Usually we have to use multiple ***echo*** commands to do so, but we can do the same *in* ***echo*** command by enabling -e option in it.



- → The following are the most used option in echo escape sequence.
  **\n -** New line escape sequence.
  **\t -** New horizontal tab escape sequence.

**Without –e Option:**

```
[root@mlinux7 ~]# echo Hello; echo Welcome to VPTS
Hello
Welcome to VPTS
```

In this output to get two lines we have to use two echo commands separated by a ";" semicolon.

**With –e Option:**

***With \n***

```
[root@mlinux7 ~]# echo -e "Hello\nWelcome to VPTS"
Hello
Welcome to VPTS
```

***With \t***

```
[root@mlinux7 ~]# echo -e "Hello\tWelcome to VPTS"
Hello   Welcome to VPTS
```

If the escape sequence like **\n \t** is used without ***–e*** it won't be treated as special characters.

## ***Lab Exercises:***

**1.** Script to print the message on the output screen.

**$vim script1.sh**

```
#!/bin/bash
echo Hello World ### echo command is used print
                 ### the message on the screen
```

***Output:***

```
[root@mlinux7 scripts]# ./script1.sh
-bash: ./script1.sh: Permission denied
[root@mlinux7 scripts]# chmod +x script1.sh
[root@mlinux7 scripts]# ./script1.sh
Hello World
```

**2.** Script to print the message on the screen using escape sequence

**$vim script2.sh**

```
#!/bin/bash
echo -e "Hello world\nWelcome to VPTS." ### echo command will print new line
                                        ### \n is mentioned in the message.
```

***Output:***

```
[root@mlinux7 scripts]# chmod +x script2.sh
[root@mlinux7 scripts]# ./script2.sh
Hello world
Welcome to VPTS.
```

**3. Script to print the output and execute the commands using Double quotes.**

**$vim script3.sh**

```
#!/bin/bash
### To execute commands in message of echo command
### Calling date command
echo "Date Command Output is = `date`"
### Calling who am i command
echo "Logged in as = `who am i`"
```

***Output:***

```
[root@mlinux7 scripts]# chmod +x script3.sh
[root@mlinux7 scripts]# ./script3.sh
Date Command Output is = Mon Oct 23 17:07:18 IST 2017
Logged in as = root     pts/0        2017-10-23 16:51
```

**Colorizing the Output:**

Shell scripts commonly used ANSI escape codes for color output. Following table shows Numbers representing colors in Escape Sequences.

| Color | Foreground | Background |
|---|---|---|
| **Black** | 30 | 40 |
| **Red** | 31 | 41 |
| **Green** | 32 | 42 |
| **Yellow** | 33 | 43 |
| **Blue** | 34 | 44 |
| **Magenta** | 35 | 45 |
| **Cyan** | 36 | 46 |
| **White** | 37 | 47 |

The following is the syntax for getting colored text

```
#echo –e "\033[COLOR(CODE)m  TEXT"
```

The **"\033["** begins the escape sequence. You can also use **"\e["** instead of **"\033[", COLOR** specifies a foreground color, according to table above. The **"m"** terminates escape sequence, and text begins immediately after that.



To print Magenta colored text:

```
[root@mlinux71 ~]# echo -e "\033[35m sample text"
 sample text
[root@mlinux71 ~]#
[root@mlinux71 ~]#
```

**Or**

```
[root@mlinux71 ~]# echo -e "\e[35m sample text"
 sample text
[root@mlinux71 ~]#
[root@mlinux71 ~]#
```

The problem with above statement is that the magenta color that starts with the 35 color code is never switched back to the regular color, so any text you type after the prompt and even prompt also is still in the Magenta color.

**To return to the plain, normal mode, we have to use another sequence**.

```
[root@mlinux71 ~]# echo -e "\033[0m"

[root@mlinux71 ~]#
[root@mlinux71 ~]#
```

Now you won't see anything new on the screen, as this echo statement was not passed any string to display. But it has done its job, which was to restore the normal viewing mode.

→ Escape sequence also allow you to control the manner in which characters are displayed on the screen.

→ The following table summarizes numbers representing text attributes in Escape Sequences.

| ANSI CODE | Meaning |
|---|---|
| 0 | Normal Characters |
| 1 | Bold Characters |
| 4 | Underlined Characters |
| 5 | Blinking Characters |
| 7 | Reverse video Characters |

Combining all these Escape Sequences, you can get more fancy effect. Use the following template for writing colored text on a colored background.

```
echo -e "\033[COLOR1;COLOR2m sample text\033[0m"
```

Example for yellow color text on blue background

```
[root@mlinux71 ~]# echo -e "\033[33;44m Yellow text on blue background\033[0m"
 Yellow text on blue background
```

Bold Yellow Text on Blue background

```
[root@mlinux71 ~]# echo -e "\033[1;33;44m Bold Yellow text on blue background\033[0m"
 Bold Yellow text on blue background
```

Bold Yellow Underlined text on Blue Background

```
[root@mlinux71 ~]# echo -e "\033[1;4;33;44m Bold Yellow underline text on blue background\033[0m"
 Bold Yellow underline text on blue background
[root@mlinux71 ~]#
```

## ***Lab Exercises:***

**1.** Using Colors in Shell scripting.

**$vim script66.sh**

```
#!/bin/bash
# This script echoes colors and codes
echo -e "\n\033[1;4;31mLight Colors\033[0m  \t\t\033[4;31mDark Colors\033[0m"

echo -e "\e[0;30;47m Black     \e[0m 0;30m \t\e[1;30;40m Dark Gray  \e[0m 1;30m"
echo -e "\e[0;31;47m Red       \e[0m 0;31m \t\e[1;31;40m Dark Red   \e[0m 1;31m"
echo -e "\e[0;32;47m Green     \e[0m 0;32m \t\e[1;32;40m Dark Green \e[0m 1;32m"
echo -e "\e[0;33;47m Yellow    \e[0m 0;33m \t\e[1;33;40m Dark Yellow\e[0m 1;33m"
echo -e "\e[0;34;47m Blue      \e[0m 0;34m \t\e[1;34;40m Dark Blue  \e[0m 1;34m"
echo -e "\e[0;35;47m Magenta   \e[0m 0;35m \t\e[1;35;40m DarkMagenta\e[0m 1;35m"
echo -e "\e[0;36;47m Cyan      \e[0m 0;36m \t\e[1;36;40m Dark Cyan  \e[0m 1;36m"
echo -e "\e[0;37;47m LightGray\e[0m 0;37m \t\e[1;37;40m White      \e[0m 1;37m"
```

***Output:***

```
[root@mlinux7 scripts]# chmod +x script66.sh
[root@mlinux7 scripts]# ./script66.sh
```

| Light Colors | | Dark Colors | |
|---|---|---|---|
| Black | 0;30m | Dark Gray | 1;30m |
| Red | 0;31m | Dark Red | 1;31m |
| Green | 0;32m | Dark Green | 1;32m |
| Yellow | 0;33m | Dark Yellow | 1;33m |
| Blue | 0;34m | Dark Blue | 1;34m |
| Magenta | 0;35m | DarkMagenta | 1;35m |
| Cyan | 0;36m | Dark Cyan | 1;36m |
| | 0;37m | White | 1;37m |

# MISCELLANEOUS COMMANDS

There are few commands which can be used only in scripting and which may make our job easy in certain situations.

**EVAL Command:**

The ***eval*** command line, performs all shell substitutions, and then executes the command line and returns the exit status of the executed command.

The following example narrates the usage of ***eval*** command.

Situation of executing the commands without ***eval*** command.

```
[root@mlinux7 scripts]# a=100
[root@mlinux7 scripts]# b='$a'
[root@mlinux7 scripts]# echo $b
$a
```

In the above example if we are expecting a value of 10 as the output but we are not getting the exact value why because the value is in single quote which is making our variable to treat-a normal character. In such situation if we need to exactly substitute the values and then the command is to be executed we have to use ***eval*** command.

The same example if you see using the ***eval*** command.

```
[root@mlinux7 scripts]# a=100
[root@mlinux7 scripts]# b='$a'
[root@mlinux7 scripts]# eval echo $b
100
```

**Colon Command (:):**

The ***colon*** command is a shell initialized command and it will have only single command that exit with a status zero. That is nothing but if you execute ***colon*** command it simply returns a status zero and does nothing other than that.

```
[root@mlinux7 ~]# :
[root@mlinux7 ~]# echo $?
0
```

The colon command is usually used in ***if*** statements which requires at least one statement.

**Type Command:**

The ***type*** shell built-in command will make you know whether the command you are executing is an alias, function, shell built-in or the command installed on the server. The syntax of ***type*** command is as follows

```
type <Command>
```

The following examples describe the usage of type command

```
[root@mlinux7 ~]# type ls
ls is aliased to `ls --color=auto'
[root@mlinux7 ~]# type cat
cat is /usr/bin/cat
[root@mlinux7 ~]# type vim
vim is /usr/bin/vim
[root@mlinux7 ~]# type history
history is a shell builtin
```

**Sleep Command:**

The ***sleep*** command pauses for a given number of seconds. The basic syntax is

```
sleep n
```

Where n is the number of seconds to sleep or pause. Some types of UNIX enable other time units to be specified. It is usually recommended that n not exceed 65,534.
***Sleep*** can be used to give a user time to read an output message before clearing the screen

**Trap Command:**

The trap command is used to catch a signal that is sent to a process created due to execution of script. An action is taken based on the signal by using the action which is define in the trap command instead of taking the default effect on the process. The basic syntax is as follows.

```
trap  '' <Signal Number>
```

In the single quotes we can give the commands when appropriate signal is given trap command will execute those commands. We can get all the signals available in Linux using ***Kill -l*** command.



```
[root@mlinux7 ~]# kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

Among all these signals we usually trap SIGINT, SIGKILL signals.

**Note:** In single quotes if we don't give any commands then ***trap*** command will not allow the shell to pass those signals to that particular process.

## ***Lab Exercises:***

1. Script for the usage of ***eval*** command.

```
$vim script63.sh
#!/bin/bash
a='10'
b='$a'
echo "Without using eval command"
echo '$b='$b
echo "With eval command"
eval echo '$b='$b
```

***Output***

```
[root@mlinux7 scripts]# chmod +x script63.sh
[root@mlinux7 scripts]# ./script63.sh
Without using eval command
$b=$a
With eval command
$a=10
```

2. Usage of xargs command.

```
$ vim script64.sh
#!/bin/bash
### Usage of xargs command
echo '$echo -e "1\n2\n3\n4\n5"'
echo -e "1\n2\n3\n4\n5"
echo Using xargs command we can convert this column into row
echo '$echo -e "1\n2\n3\n4\n5" |xargs'
echo -e "1 \n2\n3\n4\n5" |xargs
echo We can specify number of columns also in xargs using -n option
echo '$echo -e "1\n2\n3\n4\nS"|xargs -n 2'
echo -e "1\n2\n3\n4\n5"|xargs -n 2
```

***Output:***

```
[root@mlinux7 scripts]# chmod +x script64.sh
[root@mlinux7 scripts]# ./script64.sh
$echo -e "1\n2\n3\n4\n5"
1
2
3
4
5
Using xargs command we can convert this column into row
$echo -e "1\n2\n3\n4\n5" |xargs
1 2 3 4 5
We can specify number of columns also in xargs using -n option
$echo -e "1\n2\n3\n4\nS"|xargs -n 2
1 2
3 4
5
```

3. Usage of ***expr*** command

**$vim script65.sh**

```
#!/bin/bash
### expr command explained.
echo "To compare two words and get a count that how many characters are matched"
echo '$ expr Techno : Tech'
expr Techno : Tech
echo "We can compare two strings and get the status 0= Failure, 1=Success"
echo '$ expr 10 = 20'
expr 10 = 20
echo '$ expr 10 \!= 20'
expr 10 \!= 20
echo "Doing a numerical operations"
echo '$ expr 10 / 2'
echo 10 / 5
```

***Output:***

```
[root@mlinux7 scripts]# chmod +x script65.sh
[root@mlinux7 scripts]# ./script65.sh
To compare two words and get a count that how many characters are matched
$ expr Techno : Tech
4
We can compare two strings and get the status 0= Failure, 1=Success
$ expr 10 = 20
0
$ expr 10 \!= 20
1
Doing a numerical operations
$ expr 10 / 2
10 / 5
```



4. Basic usage of the trap command.

**$ vim script67.sh**

```
#!/bin/bash
### Script to use trap command and disable the CTRL + C keys.
echo "Try stopping the script using ctrl+c"
trap '' 2
sleep 10
echo "!!!Task Completed,now exiting!!!"
```

***Output:***

```
[root@mlinux7 scripts]# chmod +x script67.sh
[root@mlinux7 scripts]# ./script67.sh
Try stopping the script using ctrl+c
^C^C^C
!!!Task Completed,now exiting!!!
```

5. Usage of trap command using functions.

**$vim script68.sh**

```
#!/bin/bash
## trap using functions.
trapf() {
        echo "You pressed CTRL-C"
        echo "Removing temporary files if used"
        sleep 1
        ### Execute any command(s) if you want to do anything before exiting
        exit
        }
trap 'trapf' 2
sleep 10
echo  "Task Completed Exiting"
```

***Output:***

```
[root@mlinux7 scripts]# chmod +x script68.sh
[root@mlinux7 scripts]# ./script68.sh
^CYou pressed CTRL-C
Removing temporary files if used
```

# QUOTES

**Wild Cards:**

Wildcards are a shell feature that makes the command line much more powerful than any GUI file managers. If you want to select a big group of files in a graphical file manager, you usually have to select them with your mouse. This may seem simple, but in some cases it can be very frustrating. For, example, you have a directory with a huge amount of all kinds of files and subdirectories, and you have decided to move all the LOG files, which have the word "Linux" somewhere in the middle of their names, from that big directory into another directory. What's a simple way to do this? If the directory contains a huge amount of differently named LOG files, your task is everything but simple!

In the Linux CLI that task is just as simple to perform as moving only one LOG file, and it's so easy because of the shell wildcards. Wildcards are special characters that allow you to select filenames that match certain patterns of characters. This helps you to select even a big group of files with typing just a few characters, and in most cases it's easier than selecting the files with a mouse.

**Here's a list of the most commonly used wildcards in bash:**

| Wild Card | Description | Example |
|---|---|---|
| * | Zero or More Characters | Is *.sh, Is log*.log |
| ? | Exactly only one character | Is file? .txt |
| - | Range of Characters | Is fil e[1-3].txt |
| [ ] | Enclose a set of characters | Is fil e[1,2,3].txt |
| ~ | Denotes user home path | cd ~/Desktop |
| {} | exactly one entire word in the options given | Is {file* .txt,log*.log) |

In wild cards we looked at shell substitution, which occurs automatically whenever you enter a command containing a wildcard character or a $ parameter. The way the shell interprets these and other special characters is generally useful, but sometimes it is necessary to turn off shell substitution and let each character stand for itself. Usually we use echo command to quote the text and to turn off the special meaning of a character is called quoting, and it can be done in three ways:

- → Using Backslash
- → Using Single Quotes
- → Using Double Quotes

**Meta Characters:**

Here is a list of most of the shell special characters also called Meta characters.

*** ? [ ] ' " \ $ ; & ( ) I ^ < > new-line space tab**

**Using Backslash:**

By using, backslash we can quote or nullify the specialty of only one character at a time. We can prevent the ***echo*** command or shell from interpreting a character by placing a backslash **("\")** in front of a special character.

```
[root@mlinux7 ~]# echo Hello ; World
Hello
bash: World: command not found...
[root@mlinux7 ~]# echo Hello \; World
Hello ; World
```

In the above example to nullify the specialty of the **semicolon (;)** character we used backslash.

```
[root@mlinux7 ~]# echo You owe me $1500
You owe me 500
[root@mlinux7 ~]# echo You owe me \$1500
You owe me $1500
```

In the above example we expected the output as **$1500** which shown as **500** because of having dollar **($1)** special character initialized by shell as a variable. So after quoting it using backslash the output is as expected.

**Using Single Quotes:**

In single quotes all the characters including the special characters are treated as normal characters. So characters within single quotes are quoted just as if a backslash is in front of each character.



```
[root@mlinux7 ~]# echo <-$1500.**>; (update?) [y/n]
-bash: syntax error near unexpected token `;'
[root@mlinux7 ~]# echo \<-\$1500.\*\*\>\; \(update\?\) \[y\|n\]
<-$1500.**>; (update?) [y|n]
[root@mlinux7 ~]# echo '<-$1500.**>; (update?) [y/n]'
<-$1500.**>; (update?) [y/n]
```

**Using Double Quotes:**

The double quote **( " " )** protects everything enclosed between two double quote marks except **$, `, "** and ****. Use the double quotes when you want only variables and command substitution.

```
[root@mlinux7 ~]# echo "$SHELL"
/bin/bash
[root@mlinux7 ~]# echo "/etc/*.conf"
/etc/*.conf
[root@mlinux7 ~]# echo "Today is $(date)"
Today is Thu Oct  5 19:46:14 IST 2017
[root@mlinux7 ~]# echo "$1=Rs 60"
=Rs 60
[root@mlinux7 ~]# echo "\$1=Rs 60"
$1=Rs 60
```

**The comparisons table for all the three quotes is as follows.**

| S.No | Parameter | Backlash | Single Quote | Double Quote |
|---|---|---|---|---|
| 1. | Variable | Use Backslash to change the meaning of the characters to escape special characters. | No | Yes |
| 2. | Wild Card | | No | No |
| 3. | Command Substitute | | No | Yes |

**Quoting Rules and Situations:**

Following are the additional options that help in quoting for various scenario.

| S.No | Situation | Example | |
|---|---|---|---|
| 1. | Quoting Ignores Word Boundaries | echo Hello “;” World | |
| 2. | Combining Quoting in Commands | echo "$USER" owe me '$10' | |
| 3. | Quoting Newlines to Continue on the Next Line | echo 'Line1 >Line2’ | echo Line1 \ >Line2 |
| 4. | Embedding Spaces in a Single Argument | Echo ‘Hi Bye’ | |
| 5. | Quoting to Access Filenames Containing Special Characters | rmdir New\ Folder | |
| 6. | Quoting the Backslash to Enable echo Escape Sequences | echo -e "Linel\nline2" | |



***Lab Exercises:***

1. Script to read password from the users.

```
$vim script14.sh
#!/bin/bash
# -s option is used to read a password without displaying on the screen.
read -p 'Pl Enter your Name: ' name
read -s -p 'Pl Enter your Password: ' pass
echo -e "\nYour Name= $name\nYour Password= $pass"
```

***Output:***

```
[root@mlinux7 scripts]# chmod +x script14.sh
[root@mlinux7 scripts]# ./script14.sh
Pl Enter your Name: Musab
Pl Enter your Password:
Your Name= Musab
Your Password= abcd
```

2. Script to define two values to do basic numerical operations.

**$vim script15.sh**

```
#!/bin/bash
## Script to do numerical operation using variables .
a=20
b=10
ADD=$(($a+$b))
SUB=$(($a-$b))
MUL=$(($a*$b))
DIV=$(($a/$b))
echo -e "Addition=$ADD\nSubtraction=$SUB\nMultiply=$MUL\nDivide=$DIV"
```

***Output:***

```
[root@mlinux7 scripts]# chmod +x script15.sh
[root@mlinux7 scripts]# ./script15.sh
Addition=30
Subtraction=10
Multiply=200
Divide=2
```

3. Script to read the input values from the user and do numerical operations for the given values.

**$vim script16.sh**

```
#!/bin/bash
##Script to do numerical operations using input from user
echo "Please Enter the inputs to continue."
read -p 'Enter the 1st value: ' a
read -p 'Enter the 2nd value: ' b
ADD=$(($a+$b))
SUB=$(($a-$b))
MUL=$(($a*$b))
DIV=$(($a/$b))
echo -e "Addition=$ADD\nSubtraction=$SUB\nMultiply=$MUL\nDivide=$DIV"
```

***Output:***

```
[root@mlinux7 scripts]# chmod +x script16.sh
[root@mlinux7 scripts]# ./script16.sh
Please Enter the inputs to continue.
Enter the 1st value: 10
Enter the 2nd value: 5
Addition=15
Subtraction=5
Multiply=50
Divide=2
```

4. Script for the basic usage of pipes to get one command output and the same given as input to another command.

```
$vim script17.sh
#!/bin/bash
### Using Pipes instead of STDIN
who |wc -l ### To get how many users logged into server
echo "Body Message"|mail -s "Hey!!" admin@vpts.com
## Send an email using the echo command message as body message.
```

***Output:***

```
[root@mlinux7 scripts]# chmod +x script17.sh
[root@mlinux7 scripts]# ./script17.sh
2
```

5. Script to show the usage of basic calculator.

```
$vim script18.sh
#!/bin/bash
### bc is basic calculator in Linux
echo 1 + 2 |bc
echo 1/2 |bc
echo "scale=3; 1/2"|bc ## Using scale to print decimal values
```



***Output***:

```
[root@mlinux7 scripts]# chmod +x script18.sh
[root@mlinux7 scripts]# ./script18.sh
3
0
.500
```

6. Script to do numerical operations using ***bc*** calculator and also read the input form the user.

```
#!/bin/bash
##Script to do numerical operations using bc calculator.
echo "Please Enter the inputs to continue."
read -p 'Enter the 1st value: ' a
read -p 'Enter the 2nd value: ' b
ADD=`echo $a+$b|bc`
SUB=`echo $a-$b|bc`
MUL=`echo $a*$b|bc`
DIV=`echo "scale=5;$a/$b"|bc`
echo -e "Addition=$ADD\nSubtraction=$SUB\nMultiply=$MUL\nDivide=$DIV"
```

***Output:***

```
[root@mlinux7 scripts]# chmod +x script19.sh
[root@mlinux7 scripts]# ./script19.sh
Please Enter the inputs to continue.
Enter the 1st value: 10
Enter the 2nd value: 20
Addition=30
Subtraction=-10
Multiply=200
Divide=.50000
```

## INPUT/ OUTPUT:

**Redirectors:**

- The shell and many UNIX commands take their input from standard input (STDIN), write output to standard output (STDOUT), and write error output to standard error (STDERR). By default, standard input is connected to the terminal keyboard and standard output and error to the terminal screen.
- Redirection of I/O to a file, is accomplished by specifying the destination on the command line using a redirection Meta characters followed by the desired destination.
- The BASH uses a format for redirection which includes numbers. The numbers refers to the file descriptor numbers (0 standard input, 1 standard output, 2 standard error).

| Characters | Actions |
|---|---|
| > | Redirect standard output |
| 2> | Redirect standard error |
| 2>&1 | Redirect standard error to standard output |
| < | Redirect standard input |
| \| | Pipe standard output to another command |
| >> | Appends to standard output |
| 2>&1\| | Pipe standard output and standard error to another command |

| Example | Details |
|---|---|
| **$ls >list.out** | Redirects the output of ls command to a file list.out<br>Note: If there is a file list.out this particular command removes the old data and add the out of the latest command. |
| **$lss 2>list.err** | Redirects the STDERR to list.err file. Whereas the normal output will be printed on the screen. |
| **$ls 2>&1 >list.out** | Redirects both STDOUT and STDERR > to list.out file |
| **$ls >>list.out** | Appends output of ls command to list.out file. It will append the latest data to the file and wont erases the earlier data. |
| **mail -s TEST myuser@vpts.com <list.out** | This command will Send an email to the user but whereas the body message will be taken from a file list.out instead of STDIN from keyboard. |

**Pipes:**

A pipe is used to connect output of one program or command to the input of other command/program. It is symbolized by **"|"**

```
[root@mlinux7 ~]# cat list.err |wc -l
[root@mlinux7 ~]# cat list.err |sort
```

**Filters:**

Shell scripts are often called on to manipulate and reformat the output from commands that they execute. Sometimes this task is as simple as displaying only part of the output by filtering out certain lines. In most instances, the processing required is much more sophisticated. Now we see few basic text filtering commands with example.

**Head:**

*head* is a program on UNIX and Unix-like systems used to display the first few lines of a text file or piped data. By default it will show the first 10 lines of a file.

**Syntax: head [-number I -n number] filename**

```
$ head list .out
$ head -n 20 list. out
$ head -20 list.out
```

**Tail:**

*Tail* is a program on UNIX and Unix-like systems used to display the last few lines of a text file or piped data. By default it will show the last 10 lines of a file.

**Syntax: tail [options] filename**

```
$ tail –n 5 /etc/passwd          ### Prints the last five lines of  the file
$ tail –f  /var/log/messages   ### To see the continuous output of appended lines
```



**Grep:**

The word grep stands for globally regular expression print. *The* ***grep*** command allows you to search one file or multiple files for lines that contain a pattern.

**Syntax: grep [options] pattern [files]**

The most regular options we use in grep command are as follows

| Option | Description |
|---|---|
| **-b** | Display the block number at the beginning of each line. |
| **-c** | Display the number of matched lines. |
| **-h** | Display the matched lines, but do not display the filenames. |
| **-i** | Ignore case sensitivity. |
| **-l** | Display the filenames, but do not display the matched lines. |
| **-n** | Display the matched lines and their line numbers. |
| **-s** | Silent mode. |
| **-v** | Display all lines that do NOT match. |
| **-w** | Match whole word. |

The following are the examples for *grep* command with different options.
To search for a word **root** in */etc/passwd* file

```
[root@mlinux7 ~]# grep root /etc/passwd
root:x:0:0:root:/root:/bin/bash
operator:x:11:0:operator:/root:/sbin/nologin
```

To find the block number of search result pattern, it will list the lines of pattern found along with the block number of the line.

```
[root@mlinux7 ~]# grep -b root /etc/passwd
0:root:x:0:0:root:/root:/bin/bash
340:operator:x:11:0:operator:/root:/sbin/nologin
```

To count the number of lines pattern was found in search result

```
[root@mlinux7 ~]# grep -c root /etc/passwd
2
```

While searching for the multiple files for the same pattern it will display the lines along with the file names. For example we are searching for a pattern ***root*** from files ***/etc/passwd & /etc/shadow.***

```
# grep root /etc/passwd /etc/shadow
/etc/passwd:root:x:0:0:root:/root:/bin/bash
/etc/passwd:operator:x:11:0:operator:/root:/sbin/nologin
/etc/shadow:root:$6$AvTUk/9qKoMKOWin$hdxz5DZ5vGBvdLsedA0Bv
b41b8tU7EAuqmWvY5uXLfsbRrSGxcJWwF0xCA7h1xA/::0:99999:7:::
```

Same thing without displaying file names.

```
[root@mlinux7 ~]# grep -h root /etc/passwd /etc/shadow
root:x:0:0:root:/root:/bin/bash
operator:x:11:0:operator:/root:/sbin/nologin
root:$6$AvTUk/9qKoMKOWin$hdxz5DZ5vGBvdLsedA0BvR6BNaaIDHD
mWvY5uXLfsbRrSGxcJWwF0xCA7h1xA/::0:99999:7:::
```

Displaying only filenames without the search strings.

```
[root@mlinux7 ~]# grep -l root /etc/passwd /etc/shadow
/etc/passwd
/etc/shadow
```

Displaying all the lines except a particular word/string in the files.

```
[root@mlinux7 ~]# grep -v root /etc/passwd
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
```

Highlighting the words searched by displaying in colors in the output

```
[root@mlinux7 ~]# grep --color root /etc/passwd
root:x:0:0:root:/root:/bin/bash
operator:x:11:0:operator:/root:/sbin/nologin
```

Grep can be combined with other commands by taking input from them.

```
#cat /etc/passwd |grep root
#lvdisplay |grep mylv
```

## Cut Command:

The cut command in UNIX (or Linux) is used to select sections of text from each line of files. You can use the cut command to select fields or columns from a line by specifying a delimiter or you can select a portion of the text by specifying the range or characters. Basically the cut command slices a line and extracts the text.

The most commonly used options in cut command are as follows.

| Option | Description |
|---|---|
| **-c** | Cuts the input file using list of characters specified by this option |
| **-f** | Cuts the input file using list of field. The default field to be used TAB. |
| **-d** | Specifies a delimiter to use as a field. As mentioned previously default field is TAB and this option overwrites this default behavior. |



In the following examples we would be using following file

```
[root@mlinux7 ~]# cat cut.txt
Welcome to VPTS
```

Example for using **–c** option

```
[root@mlinux7 ~]# cut -c 2 cut.txt
e
[root@mlinux7 ~]# cut -c -3 cut.txt
Wel
[root@mlinux7 ~]# cut -c 1-3 cut.txt
Wel
[root@mlinux7 ~]# cut -c 4- cut.txt
come to VPTS
```

Example to use **-d** option

```
[root@mlinux7 ~]# cat cut.txt
Welcome:to:VPTS
[root@mlinux7 ~]# cut -d ":" -f2 cut.txt
to
[root@mlinux7 ~]# cut -d ":" -f-2 cut.txt
Welcome:to
```

Example for *–f* option with **–d**

```
[root@mlinux7 ~]# cat cut.txt
Welcome to VPTS
[root@mlinux7 ~]# cut -d " " -f1 cut.txt
Welcome
[root@mlinux7 ~]# cut -d " " -f3 cut.txt
VPTS
[root@mlinux7 ~]# cut -d " " -f1,3 cut.txt
Welcome VPTS
[root@mlinux7 ~]# cut -d " " -f-2 cut.txt
Welcome to
[root@mlinux7 ~]# cut -d " " -f2- cut.txt
to VPTS
```

## Exit Status:

Any program completes execution under the UNIX or Linux system, it returns a status back to the system. This status is a number that usually indicates whether the program successfully ran or not, and hence called as exit status. By convention, an exit status of zero indicates that a program succeeded, and non-zero (1-255) indicates that it failed. Failures can be caused by invalid arguments passed to the program, or by an error condition detected by the program.

The shell variable $? is automatically set by the shell to the exit status of the last command executed. Naturally, you can use echo to display its value at the terminal.

```
[root@mlinux7 ~]# cp myfile new_myfile
[root@mlinux7 ~]# echo $?
0
[root@mlinux7 ~]# cp vpts new_vpts
cp: cannot stat 'vpts': No such file or directory
[root@mlinux7 ~]# echo $?
1
```

Note that the numeric result of a "failure" for some commands can vary from one UNIX version to the next, but success is always signified by a zero exit status.

The following are the few exit status which will be generated regularly while executing the scripts or commands.

| Exit Code | Description |
|---|---|
| **1** | General Input Error |
| **2** | Misuse of Command and very rare |
| **126** | Cannot invoke the requested command, For example executing a script which doesn't have execute command |
| **127** | Command not found error |
| **128 + n** | Fatal error signal "n" (for example-;'kill -9 = 137). |
| **130** | Script terminated by Ctrl-C |

In shell scripting we can make our scripts generic using the exit codes. We can exit our script using exit command followed by a number.
Following examples narrates Exit States:

```
#!/bin/bash
exit 1 ##Mention the value to exit with the same status.
### To exit with the value 1.
```

```
#!/bin/bash
echo VPTS
exit 0 ### To exit with the exit value 0.
```

# AWK COMMAND

**Awk** is a powerful language to manipulate and process text files. It is especially helpful when the lines in a text files are in a record format, i.e., when each line (record) contains multiple fields separated by a delimiter.

Even when the input file is not in a record format, you can still use awk to do some basic file and data processing. You can also write programming logic using awk even when there are no input files that needs to be processed. In short, AWK is a powerful language that can come in handy to do daily routine Job.

**The *employee.txt* sample file:**

The *employee.txt* is a comma delimited file that contains 5 employee records in the following format

```
employee-number, employee-name, employee-title
```

We will be using the following file in below examples.

```
[root@mlinux7 ~]# cat employee.txt
101,John Doe,CEO
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
```



**Basic Awk Syntax:**

```
awk -Fs '/pattern/ {action}' input-file
(or)
awk -Fs '{action}' intput-file
```

In the above syntax:

- → -Fis the field separator. If you don't specify it will use an empty space as field delimiter.
- → The /pattern/ and the {action} should be enclosed inside single quotes.
- → /pattern/ is optional. If you don't provide it, awk will process all the records from the input-file. If you specify a pattern, it will process only those records from the input-file that match the given pattern.
- → {action} - These are the awk programming commands, which can be one or multiple awk commands. The whole action block (includiflg all the awk commands together) should be closed between {and}
- → Input-file - The input file that needs to be processed.

**Awk program structure (BEGIN, body, END block).**

**1. BEGIN Block**

Syntax for BEGIN block:

```
BEGIN { awk commands }
```

The begin Block gets executed only once at the beginning, before awk starts executing the body block for all the lines in the input file.

- → The begin block is a good place to print report headers, and initialize variables.
- → You can have one or more awk commands in the begin block.
- → The keyword BEGIN should be specified in upper case.
- → Begin block is optional.

**2. Body Block**

Syntax for body block:

```
/pattern/ {action}
```

The body block gets executed once for every line in the input file.

- → If the input file has 10 records, the commands in the body block will be executed 10 times (once for each record in the input file).
- → There is no keyword for the body block. We discussed pattern and action previously.

**3. END Block**

Syntax of end block:

```
END { awk-commands }
```

The end block gets executed only once at the end, after awk completes executing the body block for all the lines in the input-file.

- The end block is a good place to print report footer and do any clean-up activities.
- You can have one or more awk commands in the end block.
- The keyword **END** should be specified in upper case.



**Examples of Awk Command.**

**1.** To filter the columns in the employee.txt file using a delimiter.

```
[root@mlinux7 ~]# awk -F, '{print $1}' employee.txt
101
102
103
104
105
[root@mlinux7 ~]# awk -F, '{print $2}' employee.txt
John Doe
Jason Smith
Raj Reddy
Anand Ram
Jane Miller
```

**Note:** By default the delimiter taken by awk is either ***tab space or a single space***, To specify custom delimiter we have to use ***–F*** option.

**2.** To filter the columns along with enabling search as well.

```
[root@mlinux7 ~]# awk -F, '/Manager/ {print $2}' employee.txt
Jason Smith
Jane Miller
```

In the above example along with the filtering column we are also filtering based on text

**3.** To use the **BEGIN** Command to print the headers.

```
[root@mlinux7 ~]# awk 'BEGIN { FS=",";print "--Employee Name--"}/Manager/ {print $2}' employee.txt
--Employee Name--
Jason Smith
Jane Miller
```

**4.** To use **END** Command to print the footers.

```
[root@mlinux7 ~]# awk -F, '/Manager/ {print $2} END { print "---EOF---"}' employee.txt
Jason Smith
Jane Miller
---EOF---
```

**5.** To use **BEGIN** and **END** blocks in a single awk command.

```
[root@mlinux7 ~]# awk -F, 'BEGIN { print "EMP_ID   EMP_NAME    EMP_DESG"} {print $1"    "$2"    "$3}
END { print "---EOF---"}' employee.txt
EMP_ID   EMP_NAME    EMP_DESG
101    John Doe    CEO
102    Jason Smith    IT Manager
103    Raj Reddy    Sysadmin
104    Anand Ram    Developer
105    Jane Miller    Sales Manager
---EOF---
```

**6.** To print or read a file using awk.

```
[root@mlinux7 ~]# awk '{print}' employee.txt
101,John Doe,CEO
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
```

**7.** To see the usage of AWK Built-In Filed Separator (FS).

```
[root@mlinux7 ~]# awk -F, 'BEGIN {FS=","}{ print $1,$2}' employee.txt
101 John Doe
102 Jason Smith
103 Raj Reddy
104 Anand Ram
105 Jane Miller
```

**8.** To see the usage of Output Field Separator (OFS).

OFS is for input field separator. OFS is for output field separator. OFS is printed between consecutive fields in the output. By default, awk prints the output fields with space between the fields.

```
[root@mlinux7 ~]# awk -F ',' '{print $2, ":" $3}' employee.txt
John Doe :CEO
Jason Smith :IT Manager
Raj Reddy :Sysadmin
Anand Ram :Developer
Jane Miller :Sales Manager
[root@mlinux7 ~]#
[root@mlinux7 ~]# awk -F ',' 'BEGIN { OFS=":"}{ print $2, $3}' employee.txt
John Doe:CEO
Jason Smith:IT Manager
Raj Reddy:Sysadmin
Anand Ram:Developer
Jane Miller:Sales Manager
```

**9.** To see the Record Separator (RS) usage.

```
[root@mlinux7 ~]# cat employee-one-line.txt
101,John Doe:102,Jason Smith:103,Raj Reddy:104,Aslam khan:105,David Miller
[root@mlinux7 ~]#
[root@mlinux7 ~]# awk -F, 'BEGIN { RS=":" } { print $0 }' employee-one-line.txt
101,John Doe
102,Jason Smith
103,Raj Reddy
104,Aslam khan
105,David Miller
```

**10.** Combining all IFS, OFS, RS usage.

```
[root@mlinux7 ~]# cat employee-change-fs-ofs.txt
101
John Doe
CEO
-
102
Jason Smith
IT Manager
-
103
Raj Reddy
Sysadmin
104
Aslam khan
Developer
-
105
David Miller
Sales Manager
[root@mlinux7 ~]# awk 'BEGIN { FS="\n"; RS="-\n"; OFS=":" }{print $2, $3}' employee-change-fs-ofs.txt
John Doe:CEO
Jason Smith:IT Manager
Raj Reddy:Sysadmin
David Miller:Sales Manager
```

**11.** To check the number of records

→ ***NR*** is very helpful. When used inside the loop, this gives the line number. When used in the ***END block***, this gives the total number of records in the file.

→ Even though **NR** stands for ***"Number of Records"***, it might be appropriate to call this as ***"Number of the Record"***, as it actually gives you the line number of the current record.

```
[root@mlinux7 ~]# awk 'BEGIN {FS=","} {print "Emp Id of record number",NR,"is",$1;} END {print "Total
number of records:",NR}' employee.txt
Emp Id of record number 1 is 101
Emp Id of record number 2 is 102
Emp Id of record number 3 is 103
Emp Id of record number 4 is 104
Emp Id of record number 5 is 105
Total number of records: 5
```

**12.** To see the Auto-Increment and Auto-Decrement using Unary Operators.

```
[root@mlinux7 ~]# cat employee-sal.txt
101,John Doe,CEO,85000
102,Jason Smith,IT Manager,50000
103,Raj Reddy,Sysadmin,30000
104,Anand Ram,Developer,30000
105,Jane Miller,Sales Manager,25000
[root@mlinux7 ~]# awk -F, '{print ++$4}' employee-sal.txt
85001
50001
30001
30001
25001
[root@mlinux7 ~]# awk -F, '{print --$4}' employee-sal.txt
84999
49999
29999
29999
24999
```

**13.** To print the last column of a file or last second column.

```
[root@mlinux7 ~]# awk -F, '{print $NF}' employee.txt
CEO
IT Manager
Sysadmin
Developer
Sales Manager
[root@mlinux7 ~]#
[root@mlinux7 ~]# awk -F, '{print $(NF-1)}' employee.txt
John Doe
Jason Smith
Raj Reddy
Anand Ram
Jane Miller
```

# SED COMMAND

**SED** stands for **Stream Editor.** It is very powerful tool to manipulate, filter, and transform text. Sed can take input from a file, or from a pipe. You might even have several sed one line commands in your bash startup file that you use for various scenarios without exactly understanding the sed scripts. For beginners, sed script might look cryptic. Once you understand the sed commands in detail, you'll be able to solve a lot of complex text manipulation problems by writing a quick sed script.

**SED Command Syntax:**
The following is the basic SED command syntax.

```
sed (options] {sed-commands} {input-file}
```

- → Sed reads one line at a time from the {input-file} and executes the {sed-commands} on that particular line.
- → It reads the 1st line from the {input-file} and executes the {sed-commands} on the 1st line. Then it reads the 2nd line from the {input-file} and executes the {sed-commands} on the 2nd line. Sed repeats this process until it reaches the end of the {input-file}.
- → There are also a few optional command line options that can be assed to sed as indicated by [options].

For all sed examples, we will be using the following employee.txt file. Please create this text file to try out the commands given in this book.

```
[root@mlinux7 ~]# cat employee.txt
101,John Doe,CEO
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
106,Stuart Gant,HR Manager
107,Tedd Stones,Practice Director
```

The above employee database contains the following fields for every record:

- → **Employee Id**
- → **Employee Name**
- → **Title**

Let's see the different ways of editing the file using SED editor with different options**.**

**SED Print Command (p):**
The syntax for printing lines in sed command is as follows.

```
sed  'p'  file
```

**7.** To print a file we use 'p' command in sed.

```
[root@mlinux7 ~]# sed p employee.txt
101,John Doe,CEO
101,John Doe,CEO
102,Jason Smith,IT Manager
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
105,Jane Miller,Sales Manager
106,Stuart Gant,HR Manager
106,Stuart Gant,HR Manager
107,Tedd Stones,Practice Director
107,Tedd Stones,Practice Director
```

In the above you can see the double lines of output. That is due to the sed work behavior the first entry is due to the backup of the line and whereas the other line is the exact output.

**8.** To print a file without any duplicate lines we use ***-n*** option in print command of sed.

```
[root@mlinux7 ~]# sed -n 'p' employee.txt
101,John Doe,CEO
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
106,Stuart Gant,HR Manager
107,Tedd Stones,Practice Director
```

**9.** To search a particular string in the file. Ex., jane in employee.txt file

```
[root@mlinux7 ~]# sed -n '/Jane/p' employee.txt
105,Jane Miller,Sales Manager
```

**10.** To search multiple expressions and print the matched lines we have to use ***-e*** option in sed command.

```
[root@mlinux7 ~]# sed -n -e '/Jane/ p' -e '/Stuart/ p' employee.txt
105,Jane Miller,Sales Manager
106,Stuart Gant,HR Manager
```

**11.** To execute multiple commands from a file we have to use - f option.

```
[root@mlinux7 ~]# cat mycomm.sed
/Jane/ p
/Stuart/ p
[root@mlinux7 ~]# sed -n -f mycomm.sed employee.txt
105,Jane Miller,Sales Manager
106,Stuart Gant,HR Manager
```

In the above example we are passing multiple commands which are listed in mycomm.sed file. Usually we use this option in such an environment where we need commonly used options each and every time, then we can go with creating a file with all the option and use that particular file in the sed command.

**12.** To print only 2nd line of a file.

```
[root@mlinux7 ~]#
[root@mlinux7 ~]# sed -n '2 p' employee.txt
102,Jason Smith,IT Manager
```

In the above example we are specifying before print that to print only the 2nd line

**13.** To print from line 1 through line 4.

```
[root@mlinux7 ~]# sed -n '1,4 p' employee.txt
101,John Doe,CEO
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
```



**14.** To print from line 3 through the last line.

```
[root@mlinux7 ~]# sed -n '3,$ p' employee.txt
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
106,Stuart Gant,HR Manager
107,Tedd Stones,Practice Director
```

**15.** To print only odd numbered lines

```
[root@mlinux7 ~]# sed -n '1~2 p' employee.txt
101,John Doe,CEO
103,Raj Reddy,Sysadmin
105,Jane Miller,Sales Manager
107,Tedd Stones,Practice Director
```

**16.** To print lines starting from the 1st match of "Jason" until the 4th line.

```
[root@mlinux7 ~]# sed -n '/Jason/,4 p' employee.txt
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
```

**17.** To print lines starting from the 1st match of "Raj" until the last line.

```
[root@mlinux7 ~]# sed -n '/Raj/,$ p' employee.txt
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
106,Stuart Gant,HR Manager
107,Tedd Stones,Practice Director
```

**18.** To print lines starting from the line matching "Raj" until the line matching "Jane"

```
[root@mlinux7 ~]# sed -n '/Raj/,/Jane/ p' employee.txt
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
```

**19.** To print the line matching "Jason" and 2 lines immediately after that.

```
[root@mlinux7 ~]# sed -n '/Jason/,+2 p' employee.txt
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
```

**SED Delete Command (d):**

The syntax for deleting lines in sed command is as follows.

```
sed 'd' file
```

The following are the different ways of using the delete (d) command.

**1.** To delete only the 2nd line from a file.

```
[root@mlinux7 ~]# sed '2 d' employee.txt
101,John Doe,CEO
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
106,Stuart Gant,HR Manager
107,Tedd Stones,Practice Director
```

**2.** To delete from line 1 to 4

```
[root@mlinux7 ~]# sed '1,4 d' employee.txt
105,Jane Miller,Sales Manager
106,Stuart Gant,HR Manager
107,Tedd Stones,Practice Director
```

**3.** To delete from line 2 through last line.

```
[root@mlinux7 ~]# sed '2,$ d' employee.txt
101,John Doe,CEO
```

**4.** To delete only odd numbered lines.

```
[root@mlinux7 ~]# sed '1~2 d' employee.txt
102,Jason Smith,IT Manager
104,Anand Ram,Developer
106,Stuart Gant,HR Manager
```

**5.** To delete lines matching the pattern Manager.

```
[root@mlinux7 ~]# sed '/Manager/ d' employee.txt
101,John Doe,CEO
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
107,Tedd Stones,Practice Director
```

**6.** To delete lines starting from the 1st match of Jason until the 4th line.

```
[root@mlinux7 ~]# sed '/Jason/,4 d' employee.txt
101,John Doe,CEO
105,Jane Miller,Sales Manager
106,Stuart Gant,HR Manager
107,Tedd Stones,Practice Director
```

**7.** To delete lines starting from the 1st match of "Raj" until the last line.

```
[root@mlinux7 ~]# sed '/Raj/,$ d' employee.txt
101,John Doe,CEO
102,Jason Smith,IT Manager
```

**8.** To delete lines starting from the line matching "Raj" until the line matching.

```
[root@mlinux7 ~]# sed '/Raj/,/Jane/ d' employee.txt
101,John Doe,CEO
102,Jason Smith,IT Manager
106,Stuart Gant,HR Manager
107,Tedd Stones,Practice Director
```

**9.** To delete lines starting from the line matching "Jason" and 2 lines immediately after that.

```
[root@mlinux7 ~]# sed '/Jason/,+2 d' employee.txt
101,John Doe,CEO
105,Jane Miller,Sales Manager
106,Stuart Gant,HR Manager
107,Tedd Stones,Practice Director
```

**10.** To delete all empty lines in a file

```
[root@mlinux7 ~]# sed '/^$/ d' employee.txt
101,John Doe,CEO
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
106,Stuart Gant,HR Manager
107,Tedd Stones,Practice Director
```

**Write Command (W):**

**1.** To Write the content of employee.txt file to file output.txt

```
[root@mlinux7 ~]# sed 'w output.txt' employee.txt
101,John Doe,CEO
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
106,Stuart Gant,HR Manager
107,Tedd Stones,Practice Director
[root@mlinux7 ~]#
[root@mlinux7 ~]# cat output.txt
101,John Doe,CEO
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
106,Stuart Gant,HR Manager
107,Tedd Stones,Practice Director
```

In the above example you can observe that the content on employee.txt file is copied to a new file output.txt and also you can see the duplicate lines while writing to other file. We can use - n option to not print the original lines on the screen. Usually in general people use Print command to print the content and use redirectors to write in a new file.

**2.** To Write the content of sed.txt file to output.txt-file without displaying the output

```
# sed -n w output .txt sed .txt
```

**3.** To write only the 2nd line to output.txt file

```
[root@mlinux7 ~]# sed -n '2 w output.txt' employee.txt
[root@mlinux7 ~]#
[root@mlinux7 ~]# cat output.txt
102,Jason Smith,IT Manager
```

**4.** To write lines from 1 to 4

```
[root@mlinux7 ~]# sed -n '1,4 w output.txt' employee.txt
[root@mlinux7 ~]#
[root@mlinux7 ~]# cat output.txt
101,John Doe,CEO
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
```

**5.** To write lines from 2 till the last line

```
[root@mlinux7 ~]# sed -n '2,$ w output.txt' employee.txt
[root@mlinux7 ~]#
[root@mlinux7 ~]# cat output.txt
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
106,Stuart Gant,HR Manager
107,Tedd Stones,Practice Director
```

**6.** To Write only odd numbered lines.

```
[root@mlinux7 ~]# sed -n '1~2 w output.txt' employee.txt
[root@mlinux7 ~]#
[root@mlinux7 ~]# cat output.txt
101,John Doe,CEO
103,Raj Reddy,Sysadmin
105,Jane Miller,Sales Manager
107,Tedd Stones,Practice Director
```

**7.** Write lines matching the pattern "Jane"

```
[root@mlinux7 ~]# sed -n '/Jane/ w output.txt' employee.txt
[root@mlinux7 ~]#
[root@mlinux7 ~]# cat output.txt
105,Jane Miller,Sales Manager
```

**8.** To write lines starting from the 1st match of "Jason" until the 4th line.

```
[root@mlinux7 ~]# sed -n '/Jason/,4 w output.txt' employee.txt
[root@mlinux7 ~]#
[root@mlinux7 ~]# cat output.txt
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
```

**9.** To write the line matching "Jason" and the next 2 lines immediately after that.

```
[root@mlinux7 ~]# sed -n '/Jason/,+2 w output.txt' employee.txt
[root@mlinux7 ~]#
[root@mlinux7 ~]# cat output.txt
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
```

**Substitute Command (s)**:

The following is the ***sed*** substitute command syntax.

```
sed '[address-range|pattern] s/originalstring/replacement-string/(substitute-flags]' inputfile
```

In the above sed substitute command syntax:

- → Address-rang or pattern -range is optional. If you don't specify one, sed will execute the substitute command on all lines.
- → s -tells sed to execute the substitute command
- → Original-string - this is the string to be searched for in the input file. The original-string can also be a regular expression.
- → Replacement-string- Sed will replace original-string with this string.
- → Substitute-flags are optional. More on this in the next section.



**1.** To replace all occurrences of Manager with Director

```
[root@mlinux7 ~]# sed 's/Manager/Director/' employee.txt
101,John Doe,CEO
102,Jason Smith,IT Director
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Director
106,Stuart Gant,HR Director
107,Tedd Stones,Practice Director
```

**2.** To Replace Manager with Director only on lines that contains the keyword "Sales"

```
[root@mlinux7 ~]# sed '/Sales/s/Manager/Director/' employee.txt
101,John Doe,CEO
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Director
106,Stuart Gant,HR Manager
107,Tedd Stones,Practice Director
```

**3.** To Replace the 1st occurrence of lower case a with upper case A.

```
[root@mlinux7 ~]# echo apple ant |sed -e 's/a/A/'
Apple ant
[root@mlinux7 ~]# echo apple ant |sed -e 's/a/A/g'
Apple Ant
```

**4.** To Replace the 2nd occurrence of lower case ***a*** to upper case ***A.***

```
[root@mlinux7 ~]# echo apple ant|sed -e 's/a/A/2'
apple Ant
```

**5.** To write only the line that was changed by the substitute command to output.txt.

```
[root@mlinux7 ~]# sed -n 's/John/Johnny/w output.txt' employee.txt
[root@mlinux7 ~]# cat output.txt
101,Johnny Doe,CEO
```

**6.** To replace John with Johnny using ***i*** command to enable case insensitivity.

```
[root@mlinux7 ~]# sed 's/john/Johnny/' employee.txt
101,John Doe,CEO
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
106,Stuart Gant,HR Manager
107,Tedd Stones,Practice Director
[root@mlinux7 ~]# sed 's/john/Johnny/i' employee.txt
101,Johnny Doe,CEO
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
106,Stuart Gant,HR Manager
107,Tedd Stones,Practice Director
```

**Append line after (a command} :**

You can insert a new line after specific locations by using the sed append command (a). The following is the syntax.

```
sed  '[address] a the -line-to-append '  input-file
```

**7.** To add a new record to the employee.txt file after line number

```
[root@mlinux7 ~]# sed '2 a 203,Rock Johnson,Engineer' employee.txt
101,John Doe,CEO
102,Jason Smith,IT Manager
203,Rock Johnson,Engineer
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
106,Stuart Gant,HR Manager
107,Tedd Stones,Practice Director
```

**8.** To add a new record to the end of the employee.txt file.

```
[root@mlinux7 ~]# sed '$ a 108,Rock Johnson,Engineer' employee.txt
101,John Doe,CEO
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
106,Stuart Gant,HR Manager
107,Tedd Stones,Practice Director
108,Rock Johnson,Engineer
[root@mlinux7 ~]#
```

**Insert Line Before (i) Command:**

The sed insert command (i) works just like the append command except that it inserts a line before a specific location instead of after the location. The following is the syntax.

```
sed ' [address] i the-line-to-insert' input-file
```

**1.** To insert a new record before line number 2 of the employee.txt file.

```
[root@mlinux7 ~]# sed '2 i 203,Rock Johnson,Engineer' employee.txt
101,John Doe,CEO
203,Rock Johnson,Engineer
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
106,Stuart Gant,HR Manager
107,Tedd Stones,Practice Director
```

**2.** To insert a new record before the last line of the employee.txt file.

```
[root@mlinux7 ~]# sed '$ i 203,Rock Johnson,Engineer' employee.txt
101,John Doe,CEO
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
106,Stuart Gant,HR Manager
203,Rock Johnson,Engineer
107,Tedd Stones,Practice Director
```

**3.** To insert two lines before the line that matches 'Jason'.

```
[root@mlinux7 ~]# sed '/Jason/i\
> 203,Rock Johnson,Engineer\
> 204,Steve Austin,Sales Engineer' employee.txt
101,John Doe,CEO
203,Rock Johnson,Engineer
204,Steve Austin,Sales Engineer
102,Jason Smith,IT Manager
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
106,Stuart Gant,HR Manager
107,Tedd Stones,Practice Director
```

## Change Command (c):

The sed change command (c) lets you replace an existing line with new text. The syntax for change command is as follows.

```
sed '[address] c the-line-to-insert' input-file
```

**1.** To delete the record at line number 2 and replace it with a new record

```
[root@mlinux7 ~]# sed '2 c 202,Rock Johnson,Engineer' employee.txt
101,John Doe,CEO
202,Rock Johnson,Engineer
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
106,Stuart Gant,HR Manager
107,Tedd Stones,Practice Director
```

**2.** To delete the line that matches 'Raj' and replaces it with two new lines.

```
[root@mlinux7 ~]# sed '/Raj/c\
> 203,Rock Johnson,Engineer\
> 204,Mark Wahlberg,Sales Engineer' employee.txt
101,John Doe,CEO
102,Jason Smith,IT Manager
203,Rock Johnson,Engineer
204,Mark Wahlberg,Sales Engineer
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
106,Stuart Gant,HR Manager
107,Tedd Stones,Practice Director
```

You can also combine the "**a, i, and c**" commands. The following sed example does all these three things:

- → a - Append 'Rock Johnson' after 'Jason'
- → i - Insert 'Mark Wahlberg' before 'Jason'
- → c - Change 'Jason' to 'Joe Mason'

```
[root@mlinux7 ~]# sed '/Jason/ {
> a\
> 204,Rock Johnson,Engineer
> i\
> 202,Mark Wahlberg,Sales Engineer
> c\
> 203,Joe Mason, Sysadmin
> }' employee.txt
101,John Doe,CEO
202,Mark Wahlberg,Sales Engineer
203,Joe Mason, Sysadmin
204,Rock Johnson,Engineer
103,Raj Reddy,Sysadmin
104,Anand Ram,Developer
105,Jane Miller,Sales Manager
106,Stuart Gant,HR Manager
107,Tedd Stones,Practice Director
```

# VARIABLE

A variable in a shell script is a means of referencing a numeric or character value. And unlike formal programming languages, a shell script doesn't require you to declare a type for your variables. Thus, you could assign a number to the variable stuff and then use it again in the same script to hold a string of characters. To access the value (contents) of a variable, prefix it with a dollar **($)** sign. Usually the variables are used to store the path of the files & output of commands.

**Setting a Variable:**
The shell enables us to set and unset a variable

**Syntax: name=value  Ex: NAME=Musab**

- → Here **'name'** is the name of the variable and **'value'** is the data that variable name hold.
- → In the above example, **NAME** is the variable and **Musab** is the data named to the variable.
- → Variables of this type are called scalar variables, which mean it can hold only one value at a time.

**Accessing the Variable:**
To access a variable, prefix its name with the dollar sign.
Ex: echo $NAME

```
[root@mlinux71 ~]# NAME=Musab
[root@mlinux71 ~]#
[root@mlinux71 ~]# echo $NAME
Musab
```

In order to store multiple words or multiple lines in a variable you need to use either single quote or double quote.

```
[root@mlinux71 ~]# NAME="Musab Syed"
[root@mlinux71 ~]# echo $NAME
Musab Syed
[root@mlinux71 ~]# NAME='Musab Syed'
[root@mlinux71 ~]# echo $NAME
Musab Syed
[root@mlinux71 ~]#
```

**Rules to define a Variable:**
**T**here are certain limitation for defining the name of the variable.

- → The name of the variable can contain letters (a to z & A to Z)
- → It can have numbers ( 0 to9 )
- → No special characters can be used in the name of a variable other than underscore character "_".
- → A variable name can either start with letter or underscore and it cannot be started with a number.
- → Variable names starting with numbers are used for shell which will be discussed in coming topics.

## Command Substitution:

The Bourne shell can redirect a command's standard output back to the shell's own command line. That is, you can use a command's output as an argument to another command, or you can store the command output in a shell variable. We can do this by enclosing a command in back quotes (') or in braces (()).

**Syntax: VAR=`COMMAND` OR VAR=$(COMMAND)**

```
[root@mlinux71 ~]# STATUS=`passwd -S myuser`
[root@mlinux71 ~]# echo $STATUS
myuser PS 1969-12-31 0 99999 7 -1 (Password set, SHA512 crypt.)
```

**Arithmetic Substitution:**

In ksh and bash, the shell enables integer arithmetic to be performed. This avoids having to run an extra program such as expr or be to do math in a shell script. This feature is not available in sh.

**Syntax: VAR=$((expression))**

```
[root@mlinux71 ~]#
[root@mlinux71 ~]# cal=$(( ((10 + 5*2) -4) /2 ))
[root@mlinux71 ~]# echo $cal
8
```

**The following are the operators could be used in the arithmetic substitution.**

| OPERATOR | DESCRIPTION |
|---|---|
| + | Addition operator, Add two Numbers |
| - | Subtraction Operator, Subtract two Numbers |
| * | Multiplication Operator, Multiply two Numbers |
| / | Division Operator, Division |
| () | The parentheses clarify which expressions should be evaluated before others. |

**Note: Arithmetic Substitution usually won't give the value in decimals, to get the values in decimal way we have to go with other command "*bc*".**

***Lab Exercises:***

**4.** Script to

a. Declare a variable
b. Access the variable
c. Unset the variable
d. Include the comment for every command.

**$vim script4.sh**

```
#!/bin/bash
###Script to set and unset a variable.
a=10     #Declaring a variable.
echo $a #Accessing the variable.
unset a #Unsettling the variable.
```

***Output:***

```
[root@mlinux7 scripts]# chmod +x script4.sh
[root@mlinux7 scripts]# ./script4.sh
10
```

## ARRAY VARIABLES

The variable in Bash is a scalar. From Bash 2.0 it can support a type of variable called Array Variable which can handle multiple values at a same time. All the naming rules discussed for Shell Variables would be applicable while naming arrays. Shell does not create a bunch of blank array items to fill the space between indexes. It will keep track of an array index that contains values. In KSH numerical indices for arrays must be between 0 and 1023. In bash there is no limitation. The index supports only integers and we cannot use floating and decimal values. If an array variable with the same name as scalar variable is defined, the value of scalar variable becomes the value of the element of the array at index 0.

**Syntax**:

```
ArrayName[index]=Value  Ex: VAR[x]=value
```

Here, VAR is the variable name
[x] Refer the index value
Value is the data that the array variable would be holding.

**Example:**

```
Fruits[0]="Apple"
Fruits[1]="Mango"
Fruits[2]="Orange"
Fruits[3]="Banana"
```

Another way with which you can initialize the Array variables are as follows:

**Syntax:**

**ArrayName=(Value1 Value2 Value3 .. ValueN)**

**Example:**

**Fruits=(Apple Mango Orange Banana)**

**The Following examples shows us how to use array:**

```
#!/bin/bash
########## Demo Script for array variables

### Defining an array individually
fruits[0]="Apple"
fruits[1]="Mango"
fruits[2]="Orange"
fruits[3]="Banana"

### Accessing the variable

echo 'echo ${fruits[0]}='${fruits[0]}
echo 'echo ${fruits[1]}='${fruits[1]}
echo 'echo ${fruits[2]}='${fruits[2]}
echo 'echo ${fruits[3]}='${fruits[3]}

### Accessing all values at a time

echo 'echo ${fruits[*]}='${fruits[*]}
echo 'echo ${fruits[@]}='${fruits[@]}

### To get the array size

echo 'echo ${#fruits[*]}='${#fruits[*]}
echo 'echo ${#fruits[@]}='${#fruits[@]}
```

**Output**

```
[root@mlinux7 ~]# ./array.sh
echo ${fruits[0]}=Apple
echo ${fruits[1]}=Mango
echo ${fruits[2]}=Orange
echo ${fruits[3]}=Banana
echo ${fruits[*]}=Apple Mango Orange Banana
echo ${fruits[@]}=Apple Mango Orange Banana
echo ${#fruits[*]}=4
echo ${#fruits[@]}=4
```

**Unset Variable:**

The shell built -in unset command is used for unsetting the data that an initialized variable is holding.

**Syntax: unset <Variable>  Ex: unset name**

The above example will unset the data for the variable name is holding.

**Read-only Variables:**

The Shell allows us to make or mark a variable as read-only; once the value is marked as read-only it cannot be changed or manipulated.

**Syntax: readonly <Variable Name>  Ex: readonly name**

The above example will set the variable "name" as readonly.

```
[root@mlinux7 ~]# name="VPTS"
[root@mlinux7 ~]# readonly name
[root@mlinux7 ~]# echo $name
VPTS
[root@mlinux7 ~]# unset name
-bash: unset: name: cannot unset: readonly variable
[root@mlinux7 ~]# name=hello
-bash: name: readonly variable
[root@mlinux7 ~]#
```

**Note:** Once the variable is marked as **"readonly"**, it can't be reversed or undone.

**Environment Variables:**

Generally when you create a variable it is only valid on your shell, it is not available for others on different terminal/shell. Environmental variables are such variables which are available globally for everyone who are using the shell, although they may be connected to different session of terminal.

- → Some programs need environment variables in order to function correctly.
- → Usually a shell script defines only those environment variables that are needed by the programs that it runs.

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 will ignore all old shell's variable.

```
[root@mlinux7 ~]# name=Musab
[root@mlinux7 ~]# echo $name
Musab
[root@mlinux7 ~]# /bin/bash
[root@mlinux7 ~]# echo $name

[root@mlinux7 ~]# exit
exit
[root@mlinux7 ~]# echo $name
Musab
[root@mlinux7 ~]#
```

*An empty line is printed*

To set an environment variable we have to use the ***export*** command to use across the environment.

**Syntax: export <Variable Name>  export name**

```
[root@mlinux7 ~]# name=Musab
[root@mlinux7 ~]# echo $name
Musab
[root@mlinux7 ~]# export name
[root@mlinux7 ~]# /bin/bash
[root@mlinux7 ~]# echo $name
Musab
[root@mlinux7 ~]# exit
exit
[root@mlinux7 ~]# echo $name
Musab
```

In Bash Shell, with the ***set*** command we can check whether the variable is set to some value or not. If we want to check only the variables which are exported then ***export -p*** is the command.

```
[root@mlinux7 ~]# set
[root@mlinux7 ~]# export -p
```

***Lab Exercises:***

**5.** Script to declare an environment variable and try to access them in local shell.

**$vim script5.sh**

```
#!/bin/bash
## Script to declare environment variables.
MYNAME='VPTS'
export MYNAME
### OR ###
export MYNAME='VPTS'
```

***Output:***

```
[root@mlinux7 scripts]# chmod +x script5.sh
[root@mlinux7 scripts]# ./script5.sh
[root@mlinux7 scripts]# echo $MYNAME

[root@mlinux7 scripts]# . script5.sh
[root@mlinux7 scripts]# echo $MYNAME
VPTS
```

**Note:** If you are declaring any environment variables those scripts has to be executed with '**. <Space>** ' not with**". /**".

6. Define an environment variable in shell and write a script to access the same variable.

**$vim script6.sh**

```
#!/bin/bash
### Accessing an environmental variable
### defined in a shell
echo $MYNAME1
```

***Output:***

```
[root@mlinux7 scripts]# chmod +x script6.sh
[root@mlinux7 scripts]# ./script6.sh

[root@mlinux7 scripts]# export MYNAME1=MUSAB
[root@mlinux7 scripts]# ./script6.sh
MUSAB
```

7. Script to print the variable defined in 5th example.

**$vim script7.sh**

```
#!/bin/bash
### Accessing the variable of other script
. script5.sh
echo $MYNAME
```

***Output:***

```
[root@mlinux7 scripts]# chmod +x script7.sh
[root@mlinux7 scripts]# ./script7.sh
VPTS
```

8. Script to
   a) Declare an array variable.
   b) Access the values of an array individually.
   c) Access all the values at a time.
   d) Get the size of an array.

**$vim script8.sh**

```
#!/bin/bash
## Declaring array variables individually.

FRUIT[0]=Banana
FRUIT[1]=Mango
FRUIT[2]=Apple

##Accessing the array variables individually.
echo "###Accessing the array values individually"
echo ${FRUIT[0]}
echo ${FRUIT[1]}
echo ${FRUIT[2]}

## Declaring an array values all at a time.

FRUIT=(Banana Mango Apple)

### Accessing all values in an array at a time.
echo "###Accessing all values in array at a time."
echo ${FRUIT[*]}
### OR ###
echo ${FRUIT[@]}
### Get the size of an array
echo "###Size of an array"
echo ${#FRUIT[@]}
```



***Output:***

```
[root@mlinux7 scripts]# chmod +x script8.sh
[root@mlinux7 scripts]# ./script8.sh
###Accessing the array values individually
Banana
Mango
Apple
###Accessing all values in array at a time.
Banana Mango Apple
Banana Mango Apple
###Size of an array
3
```

9. Script to declare a readonly variable.

**$vim script9.sh**

```
#!/bin/bash
MYVAR="READONLY"
readonly MYVAR ### making a variable as readonly
echo $MYVAR
MYVAR="CHANGE"
echo $MYVAR
```

***Output:***

```
[root@mlinux7 scripts]# chmod +x script9.sh
[root@mlinux7 scripts]# ./script9.sh
READONLY
./script9.sh: line 5: MYVAR: readonly variable
READONLY
```

**10.** Script to define a variable from command output.

**$vim script10.sh**

```
#!/bin/bash
## Defining a variable from Command Output
DATE=`date +%F` ##Date command with custom option
echo "Today's Date is --> $DATE"
```

***Output:***

```
[root@mlinux7 scripts]# chmod +x script10.sh
[root@mlinux7 scripts]# ./script10.sh
Today's Date is --> 2017-10-24
```

**11.** Script to define a variable using arithmetic substitution for Addition, Subtraction, Multiplication, Division and printing the output values.

**$vim script11.sh**

```
#!/bin/bash
### Declaring Arithmetic Substitution of variables.
ADD=$((2+1))
SUB=$((2-1))
MUL=$((2*1))
DIV=$((2/1))
echo -e "Addition(2+1)=$ADD\nSubtraction(2-1)=$SUB
Multiply(2*1)=$MUL\nDivide(2/1)=$DIV"
```

***Output***:

```
[root@mlinux7 scripts]# chmod +x script11.sh
[root@mlinux7 scripts]# ./script11.sh
Addition(2+1)=3
Subtraction(2-1)=1
Multiply(2*1)=2
Divide(2/1)=2
```

# READ COMMAND

To get input from the keyboard, you use the ***read*** command. The ***read*** command takes input from the keyboard and assigns it to a variable. It is mainly used at the time taking confirmation from the user to do the jobs that seems to be little dangerous.

**Syntax: read <Variable Name> Ex: read a**

It reads the input from the user and stores in a variable. In the above example it will store in variable named "a''.

**Read - p Option:**

If you would like to print the output while reading a variable then you have use ***- p*** option. This is helpful to reduce one echo command to display the output.

**Syntax: read -p "Message to be printed" variable**

**Ex: read -p "Enter You Name" name**

**Read -s Option:**

If you would like to go with silent read we have to go with ***- s*** option in read command. Usually for reading passwords you have to mention the **- s option**; **read -s** option disables the feature of showing the text that you enter on the screen.

**Syntax: read -p -s "Message to be displayed" variable**

**Ex: read -p -s "Enter your password" pass**



**The following is the script narrates the usage of Read Command.**

```
#!/bin/bash

## Basic read command
echo -n "Enter your name: "
read name
echo "Your name is $name"

## Using -p option
read -p 'Enter your name: ' name
echo "Your name is $name"

## Using -s option to read
read -s -p 'Enter Password: ' pass
echo "Password is $pass"
```

**Output:**

```
[root@mlinux7 ~]# ./read.sh
Enter your name: Musab
Your name is Musab
Enter your name: Musab Syed
Your name is Musab Syed
Enter Password: Password is vpts
```

## ***Lab Exercises:***

**12.** Script to read the input from the user using read command and print the output of the given input.

**$ vim script12.sh**

```
#!/bin/bash
### Script to read the input from the user.
echo "Please Enter Your Name:"
read name
echo "Please Enter your course:"
read course
echo -e "Your Name: $name\nYour Course: $course"
```

***Output:***

```
[root@mlinux7 scripts]# chmod +x script12.sh
[root@mlinux7 scripts]# ./script12.sh
Please Enter Your Name:
Musab
Please Enter your course:
Shell Scripting
Your Name: Musab
Your Course: Shell Scripting
```



**13.** Script to read the input from the user and use the same read command to print the output while reading.

**$vim script13.sh**

```
#!/bin/bash
### -p option in read helps you to print the message while reading.
read -p 'Pl Enter you Name: ' name
read -p 'Pl Enter your course:'  course
echo -e "Your Name: $name\nYour Course: $course"
```

***Output:***

```
[root@mlinux7 scripts]# vim script13.sh
[root@mlinux7 scripts]# chmod +x script13.sh
[root@mlinux7 scripts]# ./script13.sh
Pl Enter you Name: Musab
Pl Enter your course:Shell Scripting
Your Name: Musab
Your Course: Shell Scripting
```

# CONDITIONS

The order in which commands execute in a shell script is called the *flow* of the script. In the scripts that you have looked at so far, the flow is always the same because the same set of commands executes every time.

In most scripts, you need to change the commands that execute depending on some condition provided by the user or detected by the script itself. When you change the commands that execute based on a condition, you change the flow of the script

Two powerful flow control mechanics are available in the shell:

- → The **case** statement
- → The **if** statement

The case statement is the other major form of flow control available in the shell. Let's see a syntax to use it.

```
case WORD in
pattern1)
list1
;;
pattern2)
list2
;;
esac
```

The case statement can execute commands based on a pattern matching decision. The word <WORD> is matched against every pattern <PATTERN n> and on a match, the associated list <LIST n> is executed. Every command list is terminated by ;;, this rule is optional for the very last command list (i.e. you can omit the ;; before the esac). In the place of PATTERN if we give "*" then if the WORD is not matched with any of the PATTERN then the LIST will be executed.

Basic Example of case statement:

```
#!/bin/bash
read -p 'Enter the OS: ' name
case $name in
        linux) cat /etc/redhat-release ;;
        unix) uname -a ;;
esac
```

***Output:***

```
[root@mlinux7 ~]# ./case1.sh
Enter the OS: linux
Red Hat Enterprise Linux Server release 7.2 (Maipo)
[root@mlinux7 ~]# ./case1.sh
Enter the OS: unix
Linux mlinux7.mb.com 3.10.0-327.el7.x86_64 #1 SMP Thu
86_64 x86_64 x86_64 GNU/Linux
```

Same thing with multiple patterns, each pattern must be delimit using pipe symbol

```
case <word> in
        <Pattern1>|<Pattern2>)
                List1
                ;;
        <Pattern3>)
                List2
                ;;
esac
```

An example for multiple pattern

```
#!/bin/bash
read -p 'Enter the OS:' name
case $name in
        linux|centos|redhat|oel) cat /etc/redhat-release ;;
        aix|hpux) uname -a ;;
esac
```

**Output**

```
[root@mlinux7 ~]# sh case2.sh
Enter the OS:redhat
Red Hat Enterprise Linux Server release 7.2 (Maipo)
[root@mlinux7 ~]# sh case2.sh
Enter the OS:aix
Linux mlinux7.mb.com 3.10.0-327.el7.x86_64 #1 SMP Thu
86_64 x86_64 x86_64 GNU/Linux
```



***Lab Exercises:***

**20.** Script for simple case statements.

```
#!/bin/bash
### Simple case statement
read -p 'Pl Enter your Choice[L/U]: ' choice
case $choice in
        L) echo "You Have Selected Linux" ;;
        U) echo "You Have Selected Unix" ;;
        *) echo "Invalid option selected" ;;
esac
```

***Output:***

```
[root@mlinux7 scripts]# chmod +x script20.sh
[root@mlinux7 scripts]# ./script20.sh
Pl Enter your Choice[L/U]: L
You Have Selected Linux
[root@mlinux7 scripts]# ./script20.sh
Pl Enter your Choice[L/U]: U
You Have Selected Unix
[root@mlinux7 scripts]# ./script20.sh
Pl Enter your Choice[L/U]: R
Invalid option selected
```

**21.** Script for numerical operation using case statements.

**$vim script21.sh**

```
#!/bin/bash
## Numerical operation using case statements.
read -p 'Pl Enter 1st Value: ' a
read -p 'Pl Enter 2nd Value: ' b
read -p 'Enter the operator[ADD|SUB|MUL|DIV]: ' op
case $op in
        ADD) ADD=$(($a+$b))
                echo "Addition = $ADD" ;;
        SUB) SUB=$(($a-$b))
                echo "Subtract = $SUB" ;;
        MUL) MUL=$(($a*$b))
                echo "Multiply =$MUL" ;;
        DIV) DIV=$(($a/$b))
                echo "Divide =$DIV" ;;
        *) echo "Invalid Operator !!" ;;
esac
```

***Output:***

```
[root@mlinux7 scripts]# vim script21.sh
[root@mlinux7 scripts]# chmod +x script21.sh
[root@mlinux7 scripts]# ./script21.sh
Pl Enter 1st Value: 24
Pl Enter 2nd Value: 26
Enter the operator[ADD|SUB|MUL|DIV]: ADD
Addition = 50
```

**Test Command:**

*test* command or [ expr ] is used to see if an expression is true, and if it is true it return zero(O), otherwise returns nonzero for false.

**Syntax:**

**test expression**

Here expression is constructed using one of the special options to the test command. The test command returns either 0 (true) or a 1 (false) after evaluating an expression.

Shorthand for the test command is placing the expression in [ ].

**[ expression ]**

Here *expression* is any valid expression that the test command understands. This shorthand form is the most common form of test that you can encounter.

The types of expressions understood by test can be broken into three types:

- → File tests
- → String comparisons
- → Numerical comparisons

**File Tests:**

File test expressions test whether a file fits some particular criteria. The general syntax for a file test is

| **test option file** **or** **[ option file ]** |
|---|

The following are the options available in file tests.

| Option | Description |
|---|---|
| **-b** | True if file exists and is a block special file. |
| **-c** | True if file exists and is a character special file. |
| **-d** | True if file exists and is a directory. |
| **-e** | True if file exists. |
| **-f** | True if file exists and is a regular file. |
| **-g** | True if file exists and has its SGID bit set. |
| **-h** | True if file exists and is a symbolic link. |
| **-k** | True if file exists and has its "sticky" bit set. |
| **-p** | True if file exists and is a named pipe. |
| **-r** | True if file exists and is readable. |
| **-s** | True if file exists and has a size greater than zero. |
| **-u** | True if file exists and has its SUID bit set. |
| **-w** | True if file exists and is writable. |
| **-x** | True if file exists and is executable. |
| **-O** | True if file exists and is owned by the effective user ID. |



**Example for file test**

```
[root@mlinux7 ~]# test -f case2.sh
[root@mlinux7 ~]# echo $?
0
[root@mlinux7 ~]# test -b case2.sh
[root@mlinux7 ~]# echo $?
1
```

In the above example we are checking a file ***case2.sh*** as regular file or 0 using ***–f*** option and it returned a value 0 (true) which means it is true that particular file is a regular file. Then we are checking the same file whether it is block special file or not using ***-b*** option and it returned a value 1 (false) which means that particular file is not a block special file. Likewise we can check the files with all the options mentioned in the above table.

The ***test*** command also supports string comparisons. We can perform the following operations in string comparisons.

- → Checking whether a string is empty or not.
- → Checking whether two strings are equal or not.

The syntax for string comparison is

| **test *option string*** **or** **[ *option string* ]** |
|---|

The following are the available options in String Comparisons.

| Option | Description |
|---|---|
| **-z string** | True if string has zero length. |
| **-n string** | True if string has nonzero length.<br>It can also be defined as "**!-z" string.** |
| **string1 = string2** | True if the strings are equal. |
| **string1 != string2** | True if the strings are not equal. |

Let's see an example how to use a test string comparison options

For checking whether a field is empty or not

```
[root@mlinux7 ~]# VAR=''
[root@mlinux7 ~]# [ -z "$VAR" ]
[root@mlinux7 ~]# echo $?
0
[root@mlinux7 ~]# [ -n "$VAR" ]
[root@mlinux7 ~]# echo $?
1
```

In the above-example we are defining a variable with an empty data and we are using ***-z*** option to check the variable has zero length and it returned a **zero** value as the string is empty. The same variable we checked with ***-n*** option and it returned a **non-zero** value as it doesn't have data.

**Note:** Make sure to use double quotes for strings while using the options.

For checking whether the strings are equal or not.

```
[root@mlinux7 ~]# VAL1='xyz'
[root@mlinux7 ~]# VAL2='abc'
[root@mlinux7 ~]# test "$VAL1" = "$VAL2"
[root@mlinux7 ~]# echo $?
1
[root@mlinux7 ~]# test "$VAL1" != "$VAL2"
[root@mlinux7 ~]# echo $?
0
```

In the above example we initialized two variables with different data in it and we are checking them whether those are equal or not. Where first returned 1 and other 0.

**Numerical Comparisons:**

The test command enables us to compare two integers. The Syntax is as follows.

| **test *integer1* option *integer2*** **or** **[ *integer1 operator integer2* ]** |
|---|

| Option | Description |
|---|---|
| **int1 -eq int2** | True if int1 is equals to int2. |
| **int1 -ne int2** | True if int1 is not equal to int2. |
| **int1 -lt int2** | True if int1 is less than int2. |
| **int1 -le int2** | True if int1 is less than or equal to int2. |
| **int1 -gt int2** | True if int1 is greater than int2. |
| **int1 -ge int2** | True if int1 is greater than or equal to int2. |

## The IF Conditions:

The ***"if"*** conditional statements perform different computations or actions depending on whether a programmer-specified Boolean condition evaluates to **true or false.** These statements are used to execute different parts of your shell program depending on whether certain conditions are true. The ability to branch makes shell scripts powerful.

We have the following ***if*** conditional statements.

- → If..then..fi statement (Simple if)
- → If..then else..fi statement (If-Else)
- → If..elif..else..fi statement (Else if ladder)
- → If..then..else..if..then..fi..fi..(Nested if)



**Simple *if:***

This statement is also called as simple if statement, the example is as follows.

```
If [ conditional expression ]
Then
        Statement 1
        Statement 2
        . . .
fi
```

If the given conditional expression is true or returns zero, it enters and executes the statements enclosed between the keywords "then" and "fi". If the given expression is false or returns a non-zero, then consequent statement list is executed.

The following the example for Simple-If conditional statement.

```
#!/bin/bash
val=10
if [ $val -eq 10 ]
then
        echo "Value is 10"
fi
```

**If Else:**

If then else statement are usually called as if else statement and the syntax is as follows.

```
if [ conditional expression ]
then
        Statement1
        Statement2
        . . .
else
        Statement3
        Statement4
        . . .
fi
```

If the conditional expression is true or returns a zero value, it executes the statement1 and 2. If the conditional expression is false or returns a nonzero, it jumps to else part, and executes the statement3 and 4. After the execution of if/else part, execution resume with the consequent statements.

```
#!/bin/bash
val=99
if [ $val -eq 100 ]
then
echo "Val is 100"
else
echo "Val is not 100"
fi
```



**Else If Ladder:**

If..elif..else..fi statements are called as Else if ladder statements. The syntax is as follows

```
if [ conditional expression1 ]
then
        Statement1
        Statement2
elif  [ conditional expression2 ]
then
        Statement3
        Statement4
else
        Statement5
fi
```

If you want to check more than one conditional expression then you have to use Else If conditional statements. It checks expression 1, if it is true executes statement 1, 2. If expression1 is false, it checks expression2, and if all the expression is false, then it enters into else block and executes the statements in the else block.

The following example narrates the usage of Else-If Statements.

```
#!/bin/bash
val=99
if [ $val -eq 100 ]
then
echo "Val is 100"
elif [ $val -gt 100 ]
then
echo "Val is greater than 100"
else
echo "Val is less than 100"
fi
```

**Nested If:**

If..then..if..then..fi..fi Statements are called as Nested If Statements. The Syntax is as follows.

```
if [ conditional expression1 ]
then
        statement1
        statement2
else
        if [ conditional expression2 ]
        then
        statement3
        fi
fi
```



An example for Nested if statements.

```
#!/bin/bash
val=99
if [ $val -le 100 ]
        then
                echo "Val is less than 100"
                if [ $val -eq 100 ]
                        then
                        echo "Val is equal to 100"
                else
                        echo "Val is Not equal to 100"
                fi
fi
```

***Lab Exercises:***

**22.** Script with numerical operators in IF statements.

**$vim script22.sh**

```
#!/bin/bash
### Numerical Operators in If Statement
a=10
b=11
if [ $a -eq $b ]; then
        echo "$a is equal to $b"
fi
if [ $a -ne $b ]; then
        echo "$a is not equal to $b"
fi
if [ $a -gt $b ]; then
        echo "$a is greater than $b"
fi
if [ $a -ge $b ]; then
        echo "$a is greater than or equal to $b"
fi
if [ $a -lt $b ]; then
        echo "$a is lesser than $b"
fi
if [ $a -le $b ]; then
        echo "$a is less than or equal to $b"
fi
```

***Output***:

```
[root@mlinux7 scripts]# chmod +x script22.sh
[root@mlinux7 scripts]# ./script22.sh
10 is not equal to 11
10 is lesser than 11
10 is less than or equal to 11
```

**23.** Script for File Tests in IF Statements.

**$vim script23.sh**

```
#!/bin/bash
#### File Tests using IF Statements.
FILE='/dev/sda'
if [ -c $FILE ]; then
        echo "$FILE is a character special file"
else
        echo "$FILE is not a character special file"
```

```
fi
if [ -b $FILE ]; then
        echo "$FILE is a block special file"
else
        echo "$FILE is not a block special file"
fi
if [ -d $FILE ]; then
        echo "$FILE is a directory"
else
        echo "$FILE is not a directory"
fi
```

***Output:***

```
[root@mlinux7 scripts]# vim script23.sh
[root@mlinux7 scripts]# chmod +x script23.sh
[root@mlinux7 scripts]# ./script23.sh
/dev/sda is not a character special file
/dev/sda is a block special file
/dev/sda is not a directory
```

**24.** Script for String comparisons in IF Statements.

**$vim script24.sh**

```
#!/bin/bash
# String Comparisons
VAL1='VPT'
VAL2='vpt'

if [ $VAL1 = $VAL2 ]; then
        echo "The given values are equal"
else
        echo "The given values are not equal"
fi
```

***Output:***

```
[root@mlinux7 scripts]# chmod +x script24.sh
[root@mlinux7 scripts]# ./script24.sh
The given values are not equal
```

**25.** Script to check whether the given input have data or not.

```
$ vim script25.sh
#!/bin/bash
### Use -z and -n option to check the variable is empty or not.
read -p 'Please Enter the value: ' A
if [ -z "$A" ]; then
echo "Variable A is empty"
fi
if [ -n "$A" ]; then
echo "Variable A is not empty"
fi
```

***Output:***

```
[root@mlinux7 scripts]# chmod +x script25.sh
[root@mlinux7 scripts]# ./script25.sh
Please Enter the value: Hi
Variable A is not empty
[root@mlinux7 scripts]# ./script25.sh
Please Enter the value:
Variable A is empty
```

**Note:** Always remember that while you are comparing strings go with Quotes or else you will some errors in some cases, so it is a best practice to start writing the scripts using quotes.

**26.** Script for Numerical Operations using IF Statements and also check whether the given input from the user is valid or no.

```
$vim script26.sh
#!/bin/bash
## Numerical Operations including If Statements and String Checks.
read -p 'Enter value1: ' a
if [ -z $a ]; then
        echo "Invalid input!!"
        exit
fi
read -p 'Enter value2: ' b
if [ -z $b ]; then
        echo "Invalid input!!"
        exit
fi
read -p 'Enter the Operator[ADD|SUB|MUL|DIV]: ' op
if [ -z $op ]; then
        echo "Invalid input!!"
        exit
fi
if [ "$op" = "ADD" ]; then
        echo "Addition = $(($a+$b))"
fi
```

```
if [ "$op" = "SUB" ]; then
        echo "Subtract = $(($a-$b))"
fi
if [ "$op" = "MUL" ]; then
        echo "Multiply = $(($a*$b))"
fi
if [ "$op" = "DIV" ]; then
        echo "Divide = $(($a/$b))"
fi
```

***Output:***

```
[root@mlinux7 scripts]# chmod +x script26.sh
[root@mlinux7 scripts]# ./script26.sh
Enter value1:
Invalid input!!
[root@mlinux7 scripts]# ./script26.sh
Enter value1: 10
Enter value2:
Invalid input!!
[root@mlinux7 scripts]# ./script26.sh
Enter value1: 20
Enter value2: 2
Enter the Operator[ADD|SUB|MUL|DIV]: DIV
Divide = 10
```

**27.** Script for Numerical Operations using ELSE-IF statement.

$vim script27.sh

```
#!/bin/bash
## Numerical operations including ELSE-If Statements and String Checks.
read -p 'Pl Enter 1st Value: ' a
if [ -z $a ];then
        echo "Invalid input!!"
        exit
fi
read -p 'Pl Enter 2nd Value: ' b
if [ -z $b ]; then
        echo "Invalid input!!"
        exit
fi
read -p 'Enter the Operator[ADD|SUB|MUL|DIV]: ' op
if [ "$op" = "ADD" ]; then
        echo "Addition= $(($a+$b))"
elif [ "$op" = "SUB" ]; then
        echo "Subtract = $(($a-$b))"
elif [ "$op" = "MUL" ]; then
        echo "Multiply= $(($a*$b))"
elif [ "$op" = "DIV" ]; then
        echo "Divide = $(($a/$b))"
```

```
else
echo "Invalid Operator!!"
exit
fi
```

***Output:***

```
[root@mlinux7 scripts]# vim script27.sh
[root@mlinux7 scripts]# chmod +x script27.sh
[root@mlinux7 scripts]# ./script27.sh
Pl Enter 1st Value: 200
Pl Enter 2nd Value: 10
Enter the Operator[ADD|SUB|MUL|DIV]: DIV
Divide = 20
```

**28.** Script for Numerical Operations using Nested ELSE-IF.

**$vim script28.sh**

```
#!/bin/bash
### Nested IF-ELSE
read -p 'Pl Enter the 1st Value: ' a
read -p 'Pl Enter the 2nd Value: ' b
if [ -n "$a" ]; then
        if [ -z "$b" ]; then
        echo "Invalid input 1st Value!!"
        exit
fi
else
        echo "Invalid input 2nd Value!!"
        exit
fi
read -p 'Pl Chose the Operator [ADD|SUB|MUL|DIV]: ' op
if [ -n "$op" ]; then
        if [ $op =  ADD ]; then
                echo "Addition = $(($a+$b))"
        else
        if [ $op =  SUB ]; then
                        echo "Subtract = $(($a-$b))"
                else
                        if [ $op =  MUL ]; then
                                echo "Multiply = $(($a*$b))"
                        else
                                if [ $op =  DIV ]; then
                                        echo "Division = $(($a/$b))"
                                else
                                        echo "Invalid Operator"
                                        exit
                                fi
                        fi
```

```
                fi
        fi
else
        echo "Invalid Operator Input"
        exit
fi
```

***Output:***

```
[root@mlinux7 scripts]# vim script28.sh
[root@mlinux7 scripts]# chmod +x script28.sh
[root@mlinux7 scripts]# ./script28.sh
Pl Enter the 1st Value: 25
Pl Enter the 2nd Value: 17
Pl Chose the Operator [ADD|SUB|MUL|DIV]: SUB
Subtract = 8
```

**29.** Script for Numerical operations using ELSE-IF and also exit with specific value if any invalid data is given.

**$vim script29.sh**

```
#!/bin/bash
## Numerical Operations including ELSE-If Statement
### We can exit with the custom exit
read -p 'Enter 1st Value: ' a
if [ -z $a ]; then
        echo "Invalid input!!"
        exit 1
fi
read -p 'Enter 2nd Value: ' b
if [ -z $b ]; then
        echo "Invalid input!!"
        exit 1
fi
read -p 'Enter the Operator[ADD|SUB|MUL|DIV]: ' op
if [ "$op" = "ADD" ]; then
        echo "Addition = $(($a+$b))"
elif [ "$op" = "SUB" ]; then
        echo "Subtract = $(($a-$b))"
elif [ "$op" = "MUL" ]; then
        echo "Multiply = $(($a*$b))"
elif [ "$op" = "DIV" ]; then
        echo "Divide = $(($a/$b))"
else
        echo "Invalid Operator!!"
        exit 1
fi
```

***Output:***

```
[root@mlinux7 scripts]# vim script29.sh
[root@mlinux7 scripts]# chmod +x script29.sh
[root@mlinux7 scripts]# ./script29.sh
Enter 1st Value:
Invalid input!!
[root@mlinux7 scripts]# echo $?
1
[root@mlinux7 scripts]# ./script29.sh
Enter 1st Value: 10
Enter 2nd Value: 2
Enter the Operator[ADD|SUB|MUL|DIV]: MUL
Multiply = 20
[root@mlinux7 scripts]# echo $?
0
```

**Compound Expressions**

- → So far you have seen individual expressions, but many times you need to combine expressions in order to satisfy a particular expression. When two or more expressions are combined, the result is called a *compound* expression.
- → You can create compound expressions using the test command's built in operators, or you can use the conditional execution operators, && and ||.
- → Also you can create a compound expression that is the negation of another expression by using the! Operator.



The following are the available options.

| Option | Description |
|---|---|
| **! expr** | True if expr is false. |
| **expr1 -a expr2** | True if both the expressions are true. |
| **expr1 - o expr2** | True if either of expression is true. |

The following is the example that narrates the usage of compound expressions of commands.

```
[root@mlinux7 ~]# test -f /etc/passwd && echo "File Exists"
File Exists
[root@mlinux7 ~]# test -d /tmp/dir1 && echo "Directory Exists"
[root@mlinux7 ~]# echo $?
1
[root@mlinux7 ~]# test -d /tmp/dir1 || mkdir /tmp/dir1
[root@mlinux7 ~]# echo $?
0
```

An example for the usage of compound expressions of if statement

```
#/bin/bash
read -p 'Please Enter your name: ' name
read -p 'Please Enter your Course: ' course
if [ -z "$name" -o -z "$course" ]; then
        echo "Invalid Data !!. Please enter again"
        exit 1
fi
echo "WELCOME TO VPTS"
```

***Output:***

```
[root@mlinux7 ~]# sh compound.sh
Please Enter your name: Musab
Please Enter your Course: Shell Scripting
WELCOME TO VPTS

[root@mlinux7 ~]# sh compound.sh
Please Enter your name:
Please Enter your Course:
Invalid Data !!. Please enter again
```

**30.** Script for compound expressions in IF Statements.

**$vim script30.sh**

```
#/bin/bash
###Compound expression in IF statements using -a (Logical AND)
### and -o (Logical OR)

###Logical AND
if [ -f /etc/passwd -a 10 -eq 10 ]; then
        echo "The file /etc/passwd exists and 10 is equal to 10"
fi
###Logical OR
if [ -f /etc/passwd -o 10 -eq 11 ]; then
        echo "The file /etc/passwd exists but 10 is not equal to 11"
fi
```

***Output:***

```
[root@mlinux7 scripts]# vim script30.sh
[root@mlinux7 scripts]# chmod +x script30.sh
[root@mlinux7 scripts]# ./script30.sh
The file /etc/passwd exists and 10 is equal to 10
The file /etc/passwd exists but 10 is not equal to 11
```

**31.** Script for Numerical Operations using IF Statements, check input using compound expressions, and exit with specific exit status.

**$vim script31.sh**

```
#!/bin/bash
### Numerical Operations Using Compound If, ELSE-IF and specific exit status
read -p 'Enter value1: ' a
read -p 'Enter value2: ' b
### Using Logical OR in IF Statement .
if [ -z "$a" -o -z "$b" ]; then
        echo "Invalid Input."
        exit 1
fi
read -p 'Enter the operator(ADD|SUB|MUL|DIV]: ' op
if [ $op = "ADD" ]; then
         OUT=$(($a+$b))
elif [ $op = "SUB" ]; then
         OUT=$(($a-$b))
elif [ $op = "MUL" ]; then
         OUT=$(($a*$b))
elif [ $op = "DIV" ]; then
         OUT=$(($a/$b))
else
         echo "Invalid Operater!!"
         exit 1
fi
echo "OUTPUT = $OUT"
```

***Output***:

```
[root@mlinux7 scripts]# chmod +x script31.sh
[root@mlinux7 scripts]# ./script31.sh
Enter value1:
Enter value2:
Invalid Input.
[root@mlinux7 scripts]# ./script31.sh
Enter value1: 10
Enter value2:
Invalid Input.
[root@mlinux7 scripts]# ./script31.sh
Enter value1: 20
Enter value2: 5
Enter the operator(ADD|SUB|MUL|DIV]: DIV
OUTPUT = 4
```

**32.** Script to nullify the output and error.

**$vim script32.sh**

```
#!/bin/bash
### Nullifying the command output and checking whether command is
## executed successfully or not.
ls >/dev/null 2>&1
no-command >/dev/null 2>&1
```

***Output***

```
[root@mlinux7 scripts]# chmod +x script32.sh
[root@mlinux7 scripts]# ./script32.sh
[root@mlinux7 scripts]#
```

**33.** Script to nullify unwanted output and check the command executed successfully or not.

**$ vim script33.sh**

```
#!/bin/bash
### Check the exit status of previtous executed command using $? variable
ls >/dev/null 2>&1
if [ $? -eq 0 ]; then
        echo "ls Command executed Successfully"
else
        echo "ls Command Not executed Successfully"
fi
lss >/dev/null 2>&1
STATUS=$?
if [ $STATUS -eq 0 ]; then
        echo "lss Command executed Successfully"
else
        echo "lss command not executed Successfully"
fi
```

***Output***:

```
[root@mlinux7 scripts]# chmod +x script33.sh
[root@mlinux7 scripts]# ./script33.sh
ls Command executed Successfully
lss command not executed Successfully
```

**34.** Script to check the given number is a number or a string with characters.

**$vim script34.sh**

```
#!/bin/bash
## Script to check the given number is a number or a string
read -p 'Pl Enter a value: ' a
test $a -eq 0 >/dev/null 2>&1
if [ $? -eq 2 ]; then
        echo "The value you have entered is not a integer"
else
        echo "The value you have entered is a integer"
fi
```

***Output:***

```
[root@mlinux7 scripts]# chmod +x script34.sh
[root@mlinux7 scripts]# ./script34.sh
Pl Enter a value: 10
The value you have entered is a integer
[root@mlinux7 scripts]# ./script34.sh
Pl Enter a value: hello
The value you have entered is not a integer
[root@mlinux7 scripts]# ./script34.sh
Pl Enter a value:
The value you have entered is not a integer
```

# LOOPS

Loops enable you to execute a series of commands multiple times. There are two basic types of loops

- → The while loop
- → The for loop

**While loop.**

The *while* loop is a control flow statement that allows code or commands to be executed repeatedly based on a given condition or expression is true or not. The following is the syntax for basic while loop.

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

If the given condition is true then the given commands will be executed continuously one by one till the condition get fails. So it is all our responsibility to control the condition otherwise the loop may get into infinite loop.

**Note:** The conditions can be either numerical or string or file check conditions.

The following is the example that narrates the usage of ***while*** loop.

```
#!/bin/bash
x=1
while [ $x -le 5 ]
do
 echo "Welcome $x times"
 x=$(( $x +1 ))
done
```



## *Lab Exercises:*

**35.** Script for basic while loop.

```
$ vim script35.sh
#!/bin/bash
### while Loop
a=1
while [ $a -le 5 ]
do
        echo "This is $a iteration"
        a=`echo $a+1|bc`
done
```

***Output:***

```
[root@mlinux7 scripts]# chmod +x script35.sh
[root@mlinux7 scripts]# ./script35.sh
This is 1 iteration
This is 2 iteration
This is 3 iteration
This is 4 iteration
This is 5 iteration
```

**36.** Script for Numerical Operations using while loop. Also check the given value for loop iteration is integer or not, also validate the data given and exit with the specific exit status.

```
#!/bin/bash
### Numerical operations using while loop.
read -p 'Pl Enter Number for iterations: ' i
if [ -n $i ]; then
        test $i -eq 0 >/dev/null 2>1
        if [ $? -eq 2 ]; then
                echo "Pl enter only integers"
                exit 2
        fi
else
        echo "Pl enter a value."
        exit 1
fi
while [ $i -gt 0 ]
do
        read -p 'Enter the 1st value: ' a
        read -p 'Enter the 2nd value : ' b
        if [ -z $a -o -z $b ]; then
                echo "Invalid Input"
                exit 1
        fi
read -p 'Enter the Operator[ADD|SUB|MUL|DIV]:' op
case $op in
        ADD) OUT=$(($a+$b));;
        SUB) OUT=$(($a-$b));;
        MUL) OUT=$(($a*$b));;
        DIV) OUT=$(($a/$b));;
        *) echo "Invalid Operator!!"
esac
echo "OUTPUT =$OUT"
i=`echo $i - 1|bc`
done
```

***Output:***

```
[root@mlinux7 scripts]# chmod +x script36.sh
[root@mlinux7 scripts]# ./script36.sh
Pl Enter Number for iterations: Hi
Pl enter only integers
[root@mlinux7 scripts]# ./script36.sh
Pl Enter Number for iterations: 1
Enter the 1st value: 10
Enter the 2nd value : 20
Enter the Operator[ADD|SUB|MUL|DIV]:ADD
OUTPUT =30
```

**UNTIL Loop:**

*Until* loop is very similar to the while loop, except that the loop executes until the expression executes successfully. As long as the given condition fails, the loop continues. The following is the basic syntax for *until* loop.

```
until [ condition ]
do
        command1
        command2
        command3
done
```

If the given condition is false then the given commands will be executed till the condition get pass or true. Until loop is actually the reverse of the while loop and also in other words these two loops are in vice versa in terms of logic in condition checking.

The following is the example narrates the usage of ***until*** loop.

```
#!/bin/bash
x=1
until [ $x -gt 5 ]
do
echo "Welcome $x times"
x=$(( $x + 1 ))
done
```

## ***Lab Exercises:***

**37.** Script for basic Until loop.

```
$vim script37.sh
#!/bin/bash
### Until Loop
a=5
until [ $a -le 0 ]
do
        echo "This is $a iteration"
        a=`echo $a-1|bc`
done
```

***Output***:

```
[root@mlinux7 scripts]# chmod +x script37.sh
[root@mlinux7 scripts]# ./script37.sh
This is 5 iteration
This is 4 iteration
This is 3 iteration
This is 2 iteration
This is 1 iteration
```

**38.** Script to do Numerical operations using until loop and also check the given values have the valid data or not.

**$ vim script38.sh**

```
#!/bin/bash
### Numerical operations using Until Loop.
read -p 'Enter Number of Iterations: ' i
if [ -n $i ]; then
        test $i -eq 0 >/dev/null 2>&1
        if [ $? -eq 2 ]; then
                echo "Pl enter Integers only"
                exit 2
        fi
else
        echo "You should enter a value.."
        exit 1
fi
until [ $i -le 0 ]
do
        read -p 'Enter the 1st Value: ' a
        read -p 'Enter the 2nd Value: ' b
        if [ -z "$a"  -o  -z "$b" ]; then
                echo "Invalid Input!!"
                exit 1
        fi
read -p 'Enter the operator[ADD|SUB|MUL|DIV]: ' op
case $op in
        ADD) OUT=$(($a+$b));;
        SUB) OUT=$(($a-$b));;
        MUL) OUT=$(($a*$b));;
        DIV) OUT=$(($a/$b));;
        *) echo "Invalid Operator!!" ;exit 1;;
esac
echo "OUTPUT = $OUT"
i=`echo $i - 1|bc`
done
```

***Output:***

```
[root@mlinux7 scripts]# chmod +x script38.sh
[root@mlinux7 scripts]# ./script38.sh
Enter Number of Iterations: 1
Enter the 1st Value:
Enter the 2nd Value: 2
Invalid Input!!
[root@mlinux7 scripts]# ./script38.sh
Enter Number of Iterations: 1
Enter the 1st Value: 2
Enter the 2nd Value:
Invalid Input!!
[root@mlinux7 scripts]# ./script38.sh
Enter Number of Iterations: 1
Enter the 1st Value: 2
Enter the 2nd Value: 8
Enter the operator[ADD|SUB|MUL|DIV]: MUL
OUTPUT = 16
```

**39.** Script for sleep command.

```
$ vim script39.sh
#!/bin/bash
## Sleep command will sleep the CPU for the number of seconds given.
sleep 10
## The above command will sleep for 10 seconds and executes
#commands.
date
```

***Output***:

```
[root@mlinux7 scripts]# chmod +x script39.sh
[root@mlinux7 scripts]# ./script39.sh
Tue Oct 31 11:58:22 IST 2017
```

**FOR Loop:**

Unlike ***While*** and ***Until*** loop, ***FOR*** Loop operates on list of items given, ***for*** loop are typically used when the number of operations is known before entering the bash loop. Bash supports two kinds of for loop. The first form of bash for loop is:

```
for varname in list
        do
                commands ##Body of the loop
        done
```

In the above syntax:

- → for, in, do and done are keywords
- → List is any list which has list of items
- → varname is any Bash variable name.

In this form, the ***for*** statement executes the command enclosed in a body, once for each item in the list. The current item from the list will be stored in a variable "varname" each time through the loop. This varname can be processed in the body of the loop. This list can be a variable that contains several words separated by spaces. If list is missing in for statement, then it takes the positional parameters that were passed into the shell.

The following is the example narrating ***FOR*** Loop first form.

```
#!/bin/bash
for var in 1 2 3 4 5
do
echo "Iteration $var"
done
```

The second form of for loop is similar to ***for*** loop in 'C' programming language, which has three expression (initialization, condition and updating).

```
for (( expr1; expr2; expr3 ))
        do
                commands
        done
```

- → In the above bash for command syntax, before the first iteration, expr1 is evaluated. This is usually used to initialize variables for the loop.
- → All the statements between do and done is executed repeatedly until the value of expr2 is TRUE.
- → After each iteration of the loop, expr3 is evaluated. This is usually use to increment a loop counter.

The following is the example narrating ***FOR*** Loop Second form:

```
#!/bin/bash

for (( i=0; i <= 5;  i++ ))
do
        echo $i
done
```



## ***Lab Exercises:***

**40.** Script for basic FOR loop.

```
$ vim script40.sh
#!/bin/bash
### FOR LOOP --First Method
for var in VALUE1 VALUE2 VALUE3
do
        echo $var
        sleep 1
done
```

***Output***:

```
[root@mlinux7 scripts]# chmod +x script40.sh
[root@mlinux7 scripts]# ./script40.sh
VALUE1
VALUE2
VALUE3
```

**41.** Script for basic FOR loop and use command to give the inputs.

**$vim script41.sh**

```
#/bin/bash
## For Loop -- Second Method
for var in `echo VALUE1 VALUE2 VALUE3`
do
        echo $var
        sleep 1
done

echo "................................."
for var in `cat out`
do
        echo $var
        sleep 1
done
```

***Output***:

```
[root@mlinux7 scripts]# cat out
VALUE4
VALUE5
VALUE6
[root@mlinux7 scripts]# chmod +x script41.sh
[root@mlinux7 scripts]# ./script41.sh
VALUE1
VALUE2
VALUE3
.................................
VALUE4
VALUE5
VALUE6
```

**42.** Script for Numerical operations using FOR Loop.

**$vim script42.sh**

```
#!/bin/bash
### Numerical Operations Using FOR LOOP and CASE Statements.
read -p 'Enter 1st Value: ' a
read -p 'Enter 2nd Value: ' b
if [ -z $a -o -z $b ]; then
        echo "Invalid Input"
        exit 1::
fi
for op in ADD SUB MUL DIV
do
        case $op in
        ADD) echo "Addition=$(($a+$b))";;
        SUB) echo "Subtraction=$(($a-$b))";;
        MUL) echo "Multiplication=$(($a*$b))";;
        DIV) echo "Division=$(($a/$b))";;
        esac
done
```

***Output***:

```
[root@mlinux7 scripts]# chmod +x script42.sh
[root@mlinux7 scripts]# ./script42.sh
Enter 1st Value: 20
Enter 2nd Value: 5
Addition=25
Subtraction=15
Multiplication=100
Division=4
```

**Select Loop with PS3 statement:**

The bash variable PS3 is exclusively used for select loop prompting message. If you would like to use the custom prompt then set the PS3 variable with the message. The following example shows using of PS3 in select loop.

```
#!/bin/bash
PS3="Please select your Option>"
select var in linux unix
do
echo "You Selected $var"
done
```

**Note:** You can see the difference in output while executing the program.

Usually *select* loop includes the either ***case*** or ***Else-if*** Statements for better control on the loops.

***Lab Exercises:***

**43.** Script for basic Select loop

**$vim script43.sh**

```
#! /bin/bash
###Select Loop
select var in VALUE1 VALUE2 VALUE3
do
echo "You have selected --> $var"
done
```

***Output***:

```
[root@mlinux7 scripts]# chmod +x script43.sh
[root@mlinux7 scripts]# ./script43.sh
1) VALUE1
2) VALUE2
3) VALUE3
#? 1
You have selected --> VALUE1
#? 3
You have selected --> VALUE3
#? 2
You have selected --> VALUE2
#? ^C
```

**Note:** Press ***CTRL + C*** to come out of the script.

**44.** Script for basic select loop and using PS3 variable.

**$ vim script44.sh**

```
#!/bin/bash
### Select Loop using PS3 variable to get the custom prompt
PS3='Select the Value > '
select var in VALUE1 VALUE2 VALUE3
do
        echo "You have selected --> $var"
done
```

***Output***:

```
[root@mlinux7 scripts]# chmod +x script44.sh
[root@mlinux7 scripts]# ./script44.sh
1) VALUE1
2) VALUE2
3) VALUE3
Select the Value > 1
You have selected --> VALUE1
Select the Value > 3
You have selected --> VALUE3
Select the Value > 2
You have selected --> VALUE2
Select the Value > ^C
```



**45.** Script for Numerical Operations using select loop.

**$vim script45.sh**

```
#/bin/bash
##Numerical Operations using Select Loop
PS3='Pl Select a value > '
select op in ADD SUB MUL DIV EXIT
do
        if [ $op = "EXIT" ]; then
                exit
        fi
        read -p 'Enter Val1: ' a
        read -p 'Enter Val2: ' b
        if [ -z "$a" -o -z "$b" ]; then
                echo "Invalid Inupt!!"
                exit 1
        fi
        case $op in
        ADD) OUT=$(($a+$b));;
        SUB) OUT=$(($a-$b));;
        MUL) OUT=$(($a*$b));;
        DIV) OUT=$(($a/$b));;
        esac
        echo "Output = $OUT"
done
```

***Output***:

```
[root@mlinux7 scripts]# chmod +x script45.sh
[root@mlinux7 scripts]# ./script45.sh
1) ADD
2) SUB
3) MUL
4) DIV
5) EXIT
Pl Select a value > 1
Enter Val1: 10
Enter Val2: 2
Output = 12
Pl Select a value > 4
Enter Val1: 10
Enter Val2: 2
Output = 5
Pl Select a value > 5
```

**Loop Control:**

The *break* and *continue* are loop control commands in shell to have better control of loops in some cases of infinite loop or even in regular loops. The *break* command terminates the loop (breaks out of it), while *continue* causes a jump to the next iteration of the loop, skipping all the remaining commands in that particular loop cycle.

Let's see how to use the break command.

```
#!/bin/bash
PS3="Please select your Option> "
select var in Linux Unix
do
        case $var in
                Linux) cat /etc/issue; break ;;
                Unix) uname -a; break ;;
        esac
done
```

And now let's see the ***continue*** option

```
#!/bin/bash
a=1
while [ $a -lt 10 ]; do
        if [ $a -gt 5 ]; then
        echo "$a is greater than 5"
        a=$(($a+1))
        continue
        fi
        echo "$a is less than or equal to 5"
        a=$(($a+1))
done
```

***Lab Exercises***:

**46.** Script for controlling the loop using break command.

**$vim script46.sh**

```
#!/bin/bash
### break command.
a=1
while [ $a -le 10 ]
do
        a=`echo $a+1|bc`
        echo "This is before break command"
        if [ $a -eq 3 ]; then
                echo "Break command Encountered"
                break
        fi
        echo "This is after break command"
done
```

***Output:***

```
[root@mlinux7 scripts]# chmod +x script46.sh
[root@mlinux7 scripts]# ./script46.sh
This is before break command
This is after break command
This is before break command
Break command Encountered
```

**47.** Script for controlling Loop using continue command.

**$vim script47.sh**

```
#!/bin/bash
### Continue command .
a=1
while [ $a -le 5 ]
do
        a=`echo $a+1|bc`
        echo "This is before continue command"
        if [ $a -gt 3 ]; then
        echo "Continue Command encountered"
        continue
        fi
        echo "This is after continue command"
done
```

***Output***:

```
[root@mlinux7 scripts]# chmod +x script47.sh
[root@mlinux7 scripts]# ./script47.sh
This is before continue command
This is after continue command
This is before continue command
This is after continue command
This is before continue command
Continue Command encountered
This is before continue command
Continue Command encountered
This is before continue command
Continue Command encountered
```

**48.** Script for infinite loop using while loop and usage of break command to come out of loop.

**$vim script48.sh**

```
#!/bin/bash
## Come out of infinite loop using break
a=1
while true
do
        echo "Iteration No : $a"
        a=`echo $a+1|bc`
        if [ $a -gt 5 ]; then
                echo "Break Command Encountered"
                break
        fi
done
```

***Output:***

```
[root@mlinux7 scripts]# chmod +x script48.sh
[root@mlinux7 scripts]# ./script48.sh
Iteration No : 1
Iteration No : 2
Iteration No : 3
Iteration No : 4
Iteration No : 5
Break Command Encountered
```

**49.** Script for Numerical Operations using While, Select loop and also use break and continue statements to control these loops.

**$ vim script49.sh**

```
#!/bin/bash
### While & Select Loop using break and continue command
PS3='Select an Option: '
while true
do
        read -p 'Enter 1st Value: ' a
        read -p 'Enter 2nd Value: ' b
        if [ -z "$a" -o -z "$b" ]; then
        echo "Invalid Inputs!!"
        continue
        fi
        select op in ADD SUB MUL DIV EXIT
                do
                        case $op in
                                ADD) echo "Addition = $(($a+$b))";break;;
                                SUB) echo "Subtract = $(($a-$b))";break;;
                                MUL) echo "Multiply = $(($a*$b))";break;;
                                DIV) echo "Divide = $(($a/$b))";break;;
                                EXIT) exit;;
                                *)echo "Wrong option selected..!! Try Again..!!";
sleep 1; continue;;
                        esac
                done
done
```

***Output:***

```
[root@mlinux7 scripts]# chmod +x script49.sh
[root@mlinux7 scripts]# ./script49.sh
Enter 1st Value: 10
Enter 2nd Value: 20
1) ADD
2) SUB
3) MUL
4) DIV
5) EXIT
Select an Option: 6
Wrong option selected..!! Try Again..!!
Select an Option: 3
Multiply = 200
Enter 1st Value: 5
Enter 2nd Value: 2
1) ADD
2) SUB
3) MUL
4) DIV
5) EXIT
Select an Option: 5
```

# FUNCTIONS

Bash shell functions are a way to group several UNIX / Linux commands for later execution using a single name for the group. Bash shell function can be executed just like a regular UNIX command. Shell functions are executed in the current shell context without creating any new process to interpret them.

Both bash aliases and functions allow you to define shortcuts for longer or more complicated commands. However, aliases don't allow control-flows, arguments, and other trickery things which these functions allows.

**Using Functions:**

The syntax for creating a functions goes as follows.

```
name( )
{
list
}
```

In the above syntax ***name*** is the name of the function and ***list*** is the list of commands. The list of commands, list, is referred to as the body of the function. And also the parenthesis i.e, **( { } )** are required followed by the name of function name. The job of a function is to bind name to list, so that whenever name is specified, list is executed. When a function is defined, list is not executed; the shell parses list to ensure that there are no syntax errors and stores name in its list of commands.

Let's see an example for the usage of functions:

```
#!/bin/bash
#### Defining a function
myfunc() {
echo "This is a function"
}
myfunc ### Calling a function
```

**Return Status in functions:**

Instead of simply returning from the function we can return with some status like the same we have in exit status. We can return only the value and a string cannot be returned here in case of how the other programming languages do.

→ You can check the return status of a function using **$?** Variable only.

Let's see an example for the same with status:

```
#!/bin/bash
myfunc() {
        echo "Executing 1st Command"
        return $?
}
myfunc
if [ $? -eq 0 ]; then
        echo "Command in the function completed succesfully"
else
        echo "Command in the function failed with an error"
fi
```

**Local Variables in functions:**

By default all the variables in shell are global. Modifying such variables will change the value in the whole script. So this may lead to have bugs while building a large scripts, we can overcome this by defining a variables locally in the function using local command. The scope these functions will be within the function and also global variables will temporarily overwritten by local variables.

The following is the syntax to define a local variable in functions.

```
function name( ){
        local var=$1
        . . .
}
```

Let's see the same with an example

```
#!/bin/bash
myfunc1() {
        a=10
}
myfunc2() {
        local b=20
}
myfunc1
myfunc2
echo "a=$a"
echo "b=$b"
```



**Export a Function:**

We can export a function using ***export*** command ***–f*** option. The following example explains the function exporting.

```
fname( ) {
echo  "Welcome to Scripting
}
export -f fname
```

**Read-only Functions:**

We can make a function as readonly using ***readonly*** command ***-f*** option. The following example narrates readonly function.

```
fname( ) {
echo  "Welcome to Scripting
}
readonly -f fname
```

## ***Lab Exercises:***

**50.** Script for defining and calling a function.

**#vim script50.sh**

```
#!/bin/bash
###Defining a function
NAME() {
echo "VPTS"
echo -n "Today date is "
date
}
### Using a function
NAME
```

***Output***:

```
[root@mlinux7 scripts]# chmod +x script50.sh
[root@mlinux7 scripts]# ./script50.sh
VPTS
Today date is Tue Oct 31 14:28:02 IST 2017
```

**51.** Script to call a function defined in shell as environment function.

**$ vim script51.sh**

```
#!/bin/bash
### Calling a function which is defined in the local shell
MYFUNC
```

***Output:***

```
[root@mlinux7 scripts]# vim script51.sh
[root@mlinux7 scripts]# chmod +x script51.sh
[root@mlinux7 scripts]# ./script51.sh
./script51.sh: line 3: MYFUNC: command not found
[root@mlinux7 scripts]# MYFUNC () {
> echo "Hello World"
> }
[root@mlinux7 scripts]# export -f MYFUNC
[root@mlinux7 scripts]# ./script51.sh
Hello World
```

**Note:** Observe that initially the script failed and later after defining a function in local shell and after exporting it, then if we execute the same script, it gives the output as expected.

**52.** Basic return command to come out from the function.

**$vim script52.sh**

```
#!/bin/bash
#### Controlling a function using return command.

MYFUNC () {
        echo "Welcome to VPTS"
        return
        echo "Have a Nice Day"
}
MYFUNC
echo "Today's date is `date`"
```

***Output:***

```
[root@mlinux7 scripts]# chmod +x script52.sh
[root@mlinux7 scripts]# ./script52.sh
Welcome to VPTS
Today's date is Tue Oct 31 14:48:58 IST 2017
```

**53.** Script to show the return status of a function.

**$ vim script53.sh**

```
#!/bin/bash
### return a function with a specific status
MYFUNC() {
        cp file1 file2 >/dev/null 2>&1
        return $?
}
MYFUNC
if [ $? -eq 0 ]; then
        echo "My Function completed successfully"
        exit 0
else
        echo "My Function failed"
        exit 1
fi
```

***Output:***

```
[root@mlinux7 scripts]# chmod +x script53.sh
[root@mlinux7 scripts]# ./script53.sh
My Function failed
```

The ***cp*** command in the function failed due to some issue and the same status have been Returned from the function to main script. Finally using the exit status we are displaying a message whether it was successful or not.

**54.** Script to do numerical operation using functions.

**$vim script54.sh**

```
#!/bin/bash
#### Numerical operations using functions.
USAGE() {
        echo "Invalid Data.. Please try Again"
        exit
        }
ADD() { echo "Addition=$(($a+$b))"; }
SUB() { echo "Subtraction=$(($a-$b))"; }
MUL() { echo "Multiplication=$(($a*$b))"; }
DIV() { echo "Division=$(($a/$b))"; }
read -p 'Enter 1st Value: ' a
read -p 'Enter 2nd Value: ' b
if [ -z "$a" -o -z "$b" ]; then
        USAGE
fi
read -p 'Enter the Operator[ADD|SUB|MUL|DIV]: ' op
case $op in
        ADD) ADD;;
        SUB) SUB;;
        MUL) MUL;;
        DIV) DIV;;
        *) USAGE;;
esac
```



***Output:***

```
[root@mlinux7 scripts]# chmod +x script54.sh
[root@mlinux7 scripts]# ./script54.sh
Enter 1st Value: 10
Enter 2nd Value:
Invalid Data.. Please try Again
[root@mlinux7 scripts]# ./script54.sh
Enter 1st Value: 10
Enter 2nd Value: 20
Enter the Operator[ADD|SUB|MUL|DIV]: HHH
Invalid Data.. Please try Again
[root@mlinux7 scripts]# ./script54.sh
Enter 1st Value: 10
Enter 2nd Value: 20
Enter the Operator[ADD|SUB|MUL|DIV]: ADD
Addition=30
```

## OPTION PARSING

**Special Variables:**

The shell defines several special variables that are relevant to option parsing. In addition to these, a few variables give the status of commands that the script executes. The following table describes all of the special variables defined by the shell.

| Variable | Description |
|---|---|
| **$0** | The name of the command being executed. For shell scripts, this is the path with which it was invoked. |
| **$n** | These variables correspond to the arguments with which a script was invoked. Here ***n*** is a positive decimal number corresponding to the position of an argument (the first argument is $1, the second argument is $2, and so on). |
| **$#** | The number of arguments supplied to a script. |
| **$*** | All the arguments are double quoted. If a script receives two arguments, $* is equivalent to $1 $2. |
| **$@** | All the arguments are individually double quoted. If a script receives two arguments, $@ is equivalent to $1 $2. |
| **$?** | The exit status of the last command executed. |
| **$$** | The process number of the current shell. For shell scripts, this is the process ID under which they are executing. |
| **$!** | The process number of the last background command. |



Let's see how to use a special variable

```
#!/bin/bash

echo "Script Name: $0"
echo "First Parameter : $1"
echo "Second Parameter: $2"
echo "All Values: $*"
echo "All Values: $@"
echo "Arguments Passed: $#"
```

**Usage Statements:**

Another common use for $0 is in the usage statement for a script, which is a short message informing the user how to invoke the script properly. All scripts used by more than one user should include such a message.

In general, the usage statement is something like the following:

```
echo  "Usage: $0  [options] [files]"
```

**Using Basename**

Currently, the message displays the entire path with which the shell script was invoked, but what is really required is the name of the shell script. You can correct this by using the basename command. The basename command takes an absolute or relative path and returns the file or directory name. Its basic syntax is

```
basename file
```

Let's see it with an example

```
[root@mlinux7 ~]# basename /usr/bin/bash
bash
```

You can use the basename command in echo input giving it as command in command quotes.

```
$ USAGE="Usage:  `basename $0` Input"
$ echo $USAGE
```

**Shift Command:**

Using ***shift*** command, command line arguments can be accessed. This command causes the positional parameters shift to the left. Shift [n] where n defaults to 1. It is useful when several parameters need to be parsed to a script because the values are limited to only 9.

For example in early $1=10, $2=20, $3=30. If we use the ***shift*** command the pointer will shift a value and $1=20 and $2=30. If we give a number to ***shift*** command then it will shift that many values at a time. A detailed example will be coming later in the manual.

## ***Lab Exercises:***

**55.** Script to see the values parsed to the script.

**#vim script55.sh**

```
#!/bin/bash
#### Special Variables
echo 'echo $0= '$0 ### Name of the script
echo 'echo $1= '$1 ### First valie parsed to the script
echo 'echo $2= '$2 ### Second value
echo 'echo $3= '$3 ### Third Value
echo 'echo $*= '$* ### All values parsed to script
echo 'echo $@= '$@ ### All values parse to script
echo 'echo $#= '$# ### Number of values parsed to script
```

***Output:***

```
[root@mlinux7 scripts]# chmod +x script55.sh
[root@mlinux7 scripts]# ./script55.sh  10 20 xyz
echo $0= ./script55.sh
echo $1= 10
echo $2= 20
echo $3= xyz
echo $*= 10 20 xyz
echo $@= 10 20 xyz
echo $#= 3
```

**56.** Script to get the use of basename and dirname commands.

**$vim script56.sh**

```
#!/bin/bash
### basename and~rnam commands
echo 'Script Name ($0)= ' $0
echo "Basename of the Script (basename \$0)= `basename $0`"
echo "Basedir of the Script (dirname \$0)= `dirname $0`"
```

***Output:***

```
[root@mlinux7 scripts]# chmod +x script56.sh
[root@mlinux7 scripts]# ./script56.sh
Script Name ($0)=  ./script56.sh
Basename of the Script (basename $0)= script56.sh
Basedir of the Script (dirname $0)= .

[root@mlinux7 /]# root/scripts/script56.sh
Script Name ($0)=  root/scripts/script56.sh
Basename of the Script (basename $0)= script56.sh
Basedir of the Script (dirname $0)= root/scripts
```

**Observation:** If you use ***basename*** you will get only the name of the script instead of the getting the complete path, and also sometimes we need the base directory and we can get that from ***dirname*** command.

**57.** Script to do Numerical Operations using Special variables by parsing the values.

**$vim script57.sh**

```
#!/bin/bash
### Numerical Operations using Special Variables.
USAGE() {
        echo -e "Usage : `basename $0` val1 val2 operator\nOperators Allowed = [ADD|SUB|MUL|DIV]\nEx : `basename $0` 10 11
 ADD"
        exit 1
}
### Checking the number of values parsed and equal to 3 or not.
if [ "$#" -ne "3" ]; then
USAGE
fi
### Checking the given values are integers or not.
test $1 -eq 0 >/dev/null 2>&1
STAT1=$?
test $2 -eq 0 >/dev/null 2>&1
STAT2=$?
if [ $STAT1 -eq 2 -o $STAT2 -eq 2 ];then
        echo "The given values are not integers"
        USAGE
fi
case $3 in
        ADD) OUT=$(($1+$2));;
        SUB) OUT=$(($1-$2));;
        MUL) OUT=$(($1*$2));;
        DIV) OUT=$(echo scale=3\;$1/$2|bc);;
        *) echo "Invalid Operator!!"; USAGE;;
esac
echo "Result = $OUT"
exit 0
```

***Output***:

```
[root@mlinux7 scripts]# chmod +x script57.sh
[root@mlinux7 scripts]# ./script57.sh 10 20 MUL
Result = 200
[root@mlinux7 scripts]# chmod +x script57.sh
[root@mlinux7 scripts]# ./script57.sh
Usage : script57.sh val1 val2 operator
Operators Allowed = [ADD|SUB|MUL|DIV]
Ex : script57.sh 10 11 ADD
[root@mlinux7 scripts]# ./script57.sh 10 20 ADD
Result = 30
[root@mlinux7 scripts]# ./script57.sh 10 20 MUL
Result = 200
[root@mlinux7 scripts]# ./script57.sh 30 15 DIV
Result = 2.000
```

**58.** Values parsing to functions.

**$ vim script58.sh**

```
#!/bin/bash
### Option parsing to functions
funcl() {
echo "Function1 values"
echo 'echo $1= ' $1
echo 'echo $2= ' $2
}
func2() {
echo "Function2 values"
echo 'echo $1= ' $1
echo 'echo $2= ' $2
}
funcl ax bx
func2
```

***Output:***

```
[root@mlinux7 scripts]# vim script58.sh
[root@mlinux7 scripts]# chmod +x script58.sh
[root@mlinux7 scripts]# ./script58.sh
Function1 values
echo $1=  ax
echo $2=  bx
Function2 values
echo $1=
echo $2=
```



**Note:** You can see the two different functions and we are parsing the values to main script and also we are parsing the values to 1st function. Hence we are getting the values parsed in function. In the second function we are not parsing any values and trying to access the values of script i.e. $1 and $2, but we are getting null values as functions will have its own environment and its own values.

**59.** Script to show the usage of ***tr*** command to convert the lower case to upper case.

**$vim script59.sh**

```
#!/bin/bash
#### Script for tr command to convert small character to capital alphabets.
read -p 'Enter Some Small Alphabets: ' c
c=`echo $c|tr [a-z] [A-Z]`
echo "Converted into Capital Alphatets: $c"
```

***Output:***

```
[root@mlinux7 scripts]# chmod +x script59.sh
[root@mlinux7 scripts]# ./script59.sh
Enter Some Small Alphabets: vpts
Converted into Capital Alphatets: VPTS
```

**60.** Script for the usage of shift command.

**$ vim script60.sh**

```
#!/bin/bash
### shift command is used to load the values if the given values are more than
##than 9
## execute the command as shown ./script62.sh aa ab ac ad ae af ag ah ai aj ak al
echo "Before Shift Command"
echo '$1 = ' $1
echo '$2 = ' $2
echo '$9 = ' $9
echo '$10 = ' $10
#####
echo 'After using shift command'
shift
echo '$1 = ' $1
echo '$2 = ' $2
echo 'After using shift 9 command'
shift 9
echo '$1 = ' $1
```

***Output***:

```
[root@mlinux7 scripts]# chmod +x script60.sh
[root@mlinux7 scripts]# ./script62.sh aa ab ac ad ae af ag ah ai aj ak al
Before Shift Command
$1 =  aa
$2 =  ab
$9 =  ai
$10 =  aa0
After using shift command
$1 =  ab
$2 =  ac
After using shift 9 command
$1 =  ak
```

## GETOPTS

Getopts obtains options and their arguments from a list of parameters that follows the standard POSIX.2 option syntax (that is, single letters proceeded by a - and possibly followed by an argument value). Typically, shell scripts use ***getopts*** to parse arguments passed to them. When you specify args on the ***getopts*** command line, ***getopts*** parses those arguments instead of the script command line.

The process by which ***getopts*** parses the options given on the command line is

1. The getopts option examines all the command line arguments, looking for arguments starting with the - character.
2. When an argument starting with the - character is found, it compares the characters following the – to the characters given in the option-siring.

3. If a match is found, the specified variable is set to the option: otherwise, variable is set to the? Character.
4. Steps 1 through 3 are repeated until all the options have been considered.

Getopts always includes with while loop in order to read all the options in a loop manner and also most of the times getopts has to include in order to read the arguments and the values parsed to arguments and to store in a constant variable.

**OPTARG:** Stores the value of the argument found by getopts.
**OPTIND:** Contains the index of the next argument to be processed.
After all the options considered were done then OPTIND is set to the number of last value.

Let's understand it with an example

```
#!/bin/bash
while getopts a:b: var
do
        case $var in
                a) a=$OPTARG;;
                b) b=$OPTARG;;
                \?) echo "Invalid Option" ; exit ;;
        esac
done
echo -e "a=$a\tb=$b"
```



If any value is missing to the argument or any other unknown options were parsed then *getopts* by default gives errors, In order to handle these errors we can either set OPTERR value to 0 or we can start the options in getopts with colon.

**OPTERR**: (Values 0 or 1) Indicates if Bash should display error messages generated by the *getopts* built-in. The value is initialized to 1 on every shell startup - so be sure to always set it to 0 if you don't want to see annoying messages!
The other way of managing the unwanted output is redirecting the error output to /dev/null.

**Using OPTERR:**

```
OPTERR=0
while getopts a:b: var
```

**Using Colon:**

```
while getopts :a:b: var
```

**Using redirection:**

```
while getopts a:b: var  >/dev/null 2>&1
```

**Parsing values to functions:**

The special variables we are talking about are different for each and every command we execute, likewise different for each and individual script. Function will have its own environment created and at the same time the special variables also will be in initialized for a function too in shell.

The following example narrates the usage of functions with special variables.

```
#!/bin/bash

### Defining a function

myfunc() {
        echo '$1 of function =' $1
        echo '$2 of function =' $2
        echo '$* of function =' $*
        echo '$# of function =' $#
}
myfunc 10 20
```

In the above example you can see the values parsed to the function and we accessing those values like the same as in normal script using the special variables.

**Note:** The scope of these special variables in a function is in the function itself, it cannot affect the other functions as well in the main program.

## *Lab Exercises:*

**60.** Script for option parsing using GETOPTS.

```
#vim script60.sh
#!/bin/bash
### Option Parsing using GETOPTS
if [ "$#" -ne "6" ]; then
        echo "Invalid parameters"
        exit 1
fi
while getopts a:b:c: var >/dev/null 2>&1
do
        case $var in
                a) a=$OPTARG;;
                b) b=$OPTARG;;
                c) c=$OPTARG;;
                *)echo Invalid Suffix exit 1;;
        esac
done
echo "Values given to suffix a = $a"
echo "Values given to suffix b = $b"
echo "Values given to suffix c = $c"
```

***Output***:

```
[root@mlinux7 scripts]# chmod +x script60.sh
[root@mlinux7 scripts]# ./script60.sh
Invalid parameters
[root@mlinux7 scripts]# ./script60.sh -a 5 -b 25 -c 30
Values given to suffix a = 5
Values given to suffix b = 25
Values given to suffix c = 30
```

**Note:** In the above example we are giving the values with a suffix in the command line and we are loading the values with ***getopts*** in the script.

**61.** Numerical Operations using GETOPTS.

**$ vim script61.sh**

```
#!/bin/bash
### Numerical Operators using GETOPTS option parsing
### Validating number of values parsed.
USAGE() {
        echo -e "Usage : `basename $0` -a val1 -b val2 -o operator\nEx: `basename $0` -a 10 -b 20 -o ADD"
        exit 1
        }
if [ "$#" -ne "6" ]; then
        echo "Invalid Number of Arguments"
        USAGE
fi
while getopts a:b:o: var>/dev/null 2>&1
do
        case $var in
                a) a=$OPTARG;;
                b) b=$OPTARG;;
                o) op=$OPTARG;;
                *) echo "Invalid Suffix Given"; USAGE;;
        esac
done

op=`echo $op|tr [A-Z] [a-z]`
case $op in
         add) OUT=$(($a+$b));;
         sub) OUT=$(($a-$b));;
         mul) OUT=$(($a*$b));;
         div) OUT=`echo scale=3\;$a/$b|bc`;;
         *) echo "Invalid Operator given"; USAGE;;
esac
echo "Result = $OUT"
exit 0
```

***Output:***

```
[root@mlinux7 scripts]# chmod +x script61.sh
[root@mlinux7 scripts]# ./script61.sh
Invalid Number of Arguments
Usage : script61.sh -a val1 -b val2 -o operator
Ex: script61.sh -a 10 -b 20 -o ADD
[root@mlinux7 scripts]# ./script61.sh -a 10 -b 5 -o MUL
Result = 50
```

# SAMPLE CLASSROOM SCRIPTS WITH OUTPUT

```
[root@mlinux1 ~]# cat script01.sh
#!/bin/bash
echo "MY FIRST SCRIPT"
[root@mlinux1 ~]#
[root@mlinux1 ~]# ./script01.sh
MY FIRST SCRIPT
```

```
[root@mlinux1 ~]# cat script02.sh
#!/bin/bash
#Script to print message
#Author: Musab
#Date: 6-4-2020
#Version:01
echo "My 2nd script" #printing a message, this is also a comment
[root@mlinux1 ~]#
[root@mlinux1 ~]# ./script02.sh
My 2nd script
```

```
[root@mlinux1 ~]# cat script03.sh
#!/bin/bash
#Script to def and use variables
name=Musab
loc=INDIA

echo "Your Name is $name"
echo "Your Location is $loc"
[root@mlinux1 ~]#
[root@mlinux1 ~]# ./script03.sh
Your Name is Musab
Your Location is INDIA
[root@mlinux1 ~]#
```

```
[root@mlinux1 ~]# cat script04.sh
#!/bin/bash
#Script to create users and assign password
user=syed
pass=`date |md5sum |cut -c 1-7` #generating random passwords

echo "Adding a user $user"
useradd $user
echo "User $user added successfully"

echo $pass |passwd $user --stdin # assigning password in single attempt

echo -e "Username=$user\nPassword=$pass"

chage -d 0 $user # forcing the user to change password at next login
[root@mlinux1 ~]#
[root@mlinux1 ~]# ./script04.sh
Adding a user syed
User syed added successfully
Changing password for user syed.
passwd: all authentication tokens updated successfully.
Username=syed
Password=88dc9eb
```

```
[root@mlinux1 ~]# cat script05.sh
#!/bin/bash
#Script to do calculations
a=100
b=40

echo "Add=$(($a+$b))"
echo "Sub=$(($a-$b))"
echo "Mul=$(($a*$b))"
echo "Div=$(($a/$b))"
[root@mlinux1 ~]#
[root@mlinux1 ~]# ./script05.sh
Add=140
Sub=60
Mul=4000
Div=2
```

```
[root@mlinux1 ~]# cat mem.sh
#!/bin/bash
#Script to convert mem usage into percentages
TOTAL=`free -m |awk '/Mem/{print $2}'`
USED=`free -m |awk '/Mem/{print $3}'`
FREE=`free -m |awk '/Mem/{print $4}'`

echo "Total Memory = $TOTAL MB"
echo "Used Memory = $(($USED*100/$TOTAL)) %"
echo "Free Memory = $(($FREE*100/$TOTAL)) %"
[root@mlinux1 ~]#
[root@mlinux1 ~]# ./mem.sh
Total Memory = 1838 MB
Used Memory = 27 %
Free Memory = 52 %
```

```
[root@mlinux1 ~]# cat script06.sh
#!/bin/bash
#Script to take input from users with read command
echo "Pl type your name"
read name

echo "Pl type your location"
read loc

echo "your name is $name"
echo "your location is $loc"
[root@mlinux1 ~]#
[root@mlinux1 ~]# ./script06.sh
Pl type your name
Musab
Pl type your location
India
your name is Musab
your location is India
```

```
[root@mlinux1 ~]# cat script07.sh
#!/bin/bash
#Script to take input from users with read command
echo -n "Pl type your name: "
read name

echo -n "Pl type your location: "
read loc

echo "your name is $name"
echo "your location is $loc"
[root@mlinux1 ~]#
[root@mlinux1 ~]# ./script07.sh
Pl type your name: Musab
Pl type your location: INDIA
your name is Musab
your location is INDIA
```

```
[root@mlinux1 ~]# cat script08.sh
#!/bin/bash
#Script to take input from users with read command
read -p "Pl type your name: " name
read -p "Pl type your location: " loc

echo "your name is $name"
echo "your location is $loc"
[root@mlinux1 ~]#
[root@mlinux1 ~]# ./script08.sh
Pl type your name: Musab
Pl type your location: TS-INDIA
your name is Musab
your location is TS-INDIA
```

```
[root@mlinux1 ~]# cat script09.sh
#!/bin/bash
#Script to take passwords in input using read command
read -p "Pl type user name: " user
read -s -p "Pl type password: " pass #secretly taking input

echo -e "\nYour username is $user\nYour passord is $pass"
[root@mlinux1 ~]#
[root@mlinux1 ~]# ./script09.sh
Pl type user name: Musab
Pl type password:
Your username is Musab
Your passord is redhat123
```

```
[root@mlinux1 ~]# cat script10.sh
#!/bin/bash
#Script to do calculations
read -p "Pl give 1st val: " a
read -p "Pl give 2nd val: " b

echo "Add=$(($a+$b))"
echo "Sub=$(($a-$b))"
echo "Mul=$(($a*$b))"
echo "Div=$(($a/$b))"
[root@mlinux1 ~]#
[root@mlinux1 ~]# ./script10.sh
Pl give 1st val: 100
Pl give 2nd val: 20
Add=120
Sub=80
Mul=2000
Div=5
```



```
[root@mlinux1 ~]# cat script11.sh
#!/bin/bash
#script to use spl variables
echo '$0='$0
echo '$1='$1
echo '$2='$2
echo '$3='$3
echo '$*='$*
echo '$@='$@
echo '$#='$#
echo '$$='$$
[root@mlinux1 ~]#
[root@mlinux1 ~]# ./script11.sh 10 20 30
$0=./script11.sh
$1=10
$2=20
$3=30
$*=10 20 30
$@=10 20 30
$#=3
$$=3550
```

```
[root@mlinux1 ~]# cat script12.sh
#!/bin/bash
#Script to do calculations using spl var
a=$1
b=$2

echo "Add=$(($a+$b))"
echo "Sub=$(($a-$b))"
echo "Mul=$(($a*$b))"
echo "Div=$(($a/$b))"
[root@mlinux1 ~]#
[root@mlinux1 ~]# ./script12.sh 100 20
Add=120
Sub=80
Mul=2000
Div=5
```



```
[root@mlinux1 ~]# cat script13.sh
#!/bin/bash
#Script to do calculations using function
calc(){
echo "Add=$(($a+$b))"
echo "Sub=$(($a-$b))"
echo "Mul=$(($a*$b))"
echo "Div=$(($a/$b))"
}
read -p "Pl give 1st val: " a
read -p "Pl give 2nd val: " b
calc
[root@mlinux1 ~]#
[root@mlinux1 ~]# ./script13.sh
Pl give 1st val: 100
Pl give 2nd val: 20
Add=120
Sub=80
Mul=2000
Div=5
```

```
[root@mlinux1 ~]# cat script14.sh
#!/bin/bash
#Script to do calculations with function by using return option
calc(){
echo "Add=$(($a+$b))"
echo "Sub=$(($a-$b))"
return  # return is used to break the function,but cont with script
echo "Mul=$(($a*$b))"
echo "Div=$(($a/$b))"
}
read -p "Pl give 1st val: " a
read -p "Pl give 2nd val: " b
calc
echo "The script completed"
[root@mlinux1 ~]#
[root@mlinux1 ~]# ./script14.sh
Pl give 1st val: 100
Pl give 2nd val: 20
Add=120
Sub=80
The script completed
```

- **Creating a function to create script file and add shebang stmt.**
- **Adding shebang and comment line.**
- **Opening file in vim editor and at exit adding +x permission to file**



```
[root@mlinux1 ~]# type myscript
myscript is a function
myscript ()
{
    i=`ls -1 script*|tail -1|sed -e 's/script//' -e 's/.sh//'`;
    i=$(($i+1));
    echo '#!/bin/bash' > script$i.sh;
    echo '#Script to' >> script$i.sh;
    vim script$i.sh;
    chmod +x script$i.sh;
    ls --color=auto --color=auto -l script$i.sh
}
[root@mlinux1 ~]#
[root@mlinux1 ~]# tail -11 .bash_profile
myscript ()
{
    i=`ls -1 script*|tail -1|sed -e 's/script//' -e 's/.sh//'`;
    i=$(($i+1));
    echo '#!/bin/bash' > script$i.sh;
    echo '#Script to' >> script$i.sh;
    vim script$i.sh;
    chmod +x script$i.sh;
    ls --color=auto -l script$i.sh
}
export -f myscript
```

```
[root@mlinux1 ~]# #myscript
[root@mlinux1 ~]# cat script15.sh
#!/bin/bash
#Script to print message
echo "File created by function"
```

```
[root@mlinux1 ~]# cat script16.sh
#!/bin/bash
#Script to use case stmt
read -p "Pl select an OS[UNIX|LINUX]: " op
case $op in
        UNIX)uname -a ;;
        LINUX)cat /etc/redhat-release ;;
        *)echo -e "\e[31mINVALID OPTION\e[0m"
esac
[root@mlinux1 ~]# ./script16.sh
Pl select an OS[UNIX|LINUX]: UNIX
Linux mlinux1.vpts.com 3.10.0-1062.el7.x86_64 #1 SMP Thu Jul 18 20:25:13
[root@mlinux1 ~]# ./script16.sh
Pl select an OS[UNIX|LINUX]: LINUX
Red Hat Enterprise Linux Server release 7.7 (Maipo)
[root@mlinux1 ~]# ./script16.sh
Pl select an OS[UNIX|LINUX]: ABC
INVALID OPTION
```

```
[root@mlinux1 ~]# cat script17.sh
#!/bin/bash
#Script to do calculations with case statement
calc(){
read -p "Pl select an operator[ADD|SUB|MUL|DIV|EXIT]: " op
op=`echo $op|tr [a-z] [A-Z]`
case $op in
        ADD)echo "Add=$(($a+$b))" ;;
        SUB)echo "Sub=$(($a-$b))" ;;
        MUL)echo "Mul=$(($a*$b))" ;;
        DIV)echo "Div=$(($a/$b))" ;;
        EXIT)echo -e "\e[33mExiting the script\e[0m"; exit ;;
        *)echo -e "\e[31mInvalid operator selected, try again...\e[0m" ;;
esac
calc
}

read -p "Pl give 1st val: " a
read -p "Pl give 2nd val: " b
calc
[root@mlinux1 ~]# ./script17.sh
Pl give 1st val: 100
Pl give 2nd val: 20
Pl select an operator[ADD|SUB|MUL|DIV|EXIT]: add
Add=120
Pl select an operator[ADD|SUB|MUL|DIV|EXIT]: sub
Sub=80
Pl select an operator[ADD|SUB|MUL|DIV|EXIT]: mul
Mul=2000
Pl select an operator[ADD|SUB|MUL|DIV|EXIT]: div
Div=5
Pl select an operator[ADD|SUB|MUL|DIV|EXIT]: xyz
Invalid operator selected, try again...
Pl select an operator[ADD|SUB|MUL|DIV|EXIT]: exit
Exiting the script
```

```
[root@mlinux1 ~]# cat script18.sh
#!/bin/bash
#Script to use simple if
read -p "pl give full path of file to check: " file
if [ -e "$file" ];then
        echo "File Exist"
fi
[root@mlinux1 ~]# ./script18.sh
pl give full path of file to check: /etc/passwd
File Exist
```

```
[root@mlinux1 ~]# cat script19.sh
#!/bin/bash
#Script to use if then else
read -p "pl give full path of file to check: " file
if [ -e "$file" ];then
        echo "File Exist"
else
        echo "File doesn't Exist"
fi
[root@mlinux1 ~]# ./script19.sh
pl give full path of file to check: /etc/passwd
File Exist
[root@mlinux1 ~]# ./script19.sh
pl give full path of file to check: /xyz
File doesn't Exist
```

```
[root@mlinux1 ~]# cat script20.sh
#!/bin/bash
#Script to do calculations with case statement
calc(){
read -p "Pl select an operator[ADD|SUB|MUL|DIV|EXIT]: " op
op=`echo $op|tr [a-z] [A-Z]`
case $op in
        ADD)echo "Add=$(($a+$b))" ;;
        SUB)echo "Sub=$(($a-$b))" ;;
        MUL)echo "Mul=$(($a*$b))" ;;
        DIV)echo "Div=$(($a/$b))" ;;
        EXIT)echo -e "\e[33mExiting the script\e[0m"; exit ;;
        *)echo -e "\e[31mInvalid operator selected, try again...\e[0m" ;;
esac
calc
}
#String check for empty values
read -p "Pl give 1st val: " a
if [ -z "$a" ];then
        echo -e "\e[31mInvalid Input\e[0m"
        exit 1
fi
read -p "Pl give 2nd val: " b
if [ -z "$b" ];then
        echo -e "\e[31m Invalid Input\e[0m"
        exit 1
fi

calc
```

```
[root@mlinux1 ~]# cat script21.sh
#!/bin/bash
#Script to do calculations with case stat and if cond with string comparisons
calc(){
read -p "Pl select an operator[ADD|SUB|MUL|DIV|EXIT]: " op
op=`echo $op|tr [a-z] [A-Z]`
if [ "$op" = "ADD" ];then
        echo "Add=$(($a+$b))"
fi

if [ "$op" = "SUB" ];then
        echo "Sub=$(($a-$b))"
fi

if [ "$op" = "MUL" ];then
        echo "Mul=$(($a*$b))"
fi

if [ "$op" = "DIV" ];then
        echo "Div=$(($a/$b))"
fi

if [ "$op" = "EXIT" ];then
        echo -e "\e[33mExiting the script\e[0m"; exit
fi
calc
}

read -p "Pl give 1st val: " a
if [ -z "$a" ];then
        echo -e "\e[31mInvalid Input\e[0m"
        exit 1
fi
read -p "Pl give 2nd val: " b
if [ -z "$b" ];then
        echo -e "\e[31m Invalid Input\e[0m"
        exit 1
fi

calc

[root@mlinux1 ~]# ./script21.sh
Pl give 1st val:
Invalid Input
[root@mlinux1 ~]# ./script21.sh
Pl give 1st val: 100
Pl give 2nd val:
 Invalid Input
[root@mlinux1 ~]# ./script21.sh
Pl give 1st val: 100
Pl give 2nd val: 20
Pl select an operator[ADD|SUB|MUL|DIV|EXIT]: add
Add=120
Pl select an operator[ADD|SUB|MUL|DIV|EXIT]: sub
Sub=80
Pl select an operator[ADD|SUB|MUL|DIV|EXIT]: mul
Mul=2000
Pl select an operator[ADD|SUB|MUL|DIV|EXIT]: div
Div=5
Pl select an operator[ADD|SUB|MUL|DIV|EXIT]: exit
Exiting the script
```

```
[root@mlinux1 ~]# cat script22.sh
#!/bin/bash
#Script to do ping test
read -p "Pl give the IP to test: " ip
ping -c4 $ip &>/dev/null
if [ $? -eq 0 ];then
        echo "$ip is pinging"
else
        echo "$ip is not pinging"
fi
[root@mlinux1 ~]# ./script22.sh
Pl give the IP to test: 192.168.10.20
192.168.10.20 is pinging
[root@mlinux1 ~]# ./script22.sh
Pl give the IP to test: 192.168.10.30
192.168.10.30 is not pinging
```

```
[root@mlinux1 ~]# cat script23.sh
#!/bin/bash
#Script to do calculations with case stat and elif cond with string comparisons
calc(){
read -p "Pl select an operator[ADD|SUB|MUL|DIV|EXIT]: " op
op=`echo $op|tr [a-z] [A-Z]`
if [ "$op" = "ADD" ];then
        echo "Add=$(($a+$b))"

elif [ "$op" = "SUB" ];then
        echo "Sub=$(($a-$b))"

elif [ "$op" = "MUL" ];then
        echo "Mul=$(($a*$b))"

elif [ "$op" = "DIV" ];then
        echo "Div=$(($a/$b))"

elif [ "$op" = "EXIT" ];then
        echo -e "\e[33mExiting the script\e[0m"; exit

else
        echo -e "\e[31mInvalid operator selected, try again...\e[0m"
fi
calc
}

read -p "Pl give 1st val: " a
if [ -z "$a" ];then
        echo -e "\e[31mInvalid Input\e[0m"
        exit 1
fi
read -p "Pl give 2nd val: " b
if [ -z "$b" ];then
        echo -e "\e[31m Invalid Input\e[0m"
        exit 1
fi

calc
```

```
[root@mlinux1 ~]# cat script24.sh
#!/bin/bash
#Script to check the status of service using nested if
read -p "Pl type the name of service to check: " ser
comm=`systemctl is-active $ser`
if [ "$comm" = "active" ];then
        echo $ser is already active
else
        echo "$ser is not running"
        read -p "Would you like to start $ser [Y/N]: " input
                if [ "$input" = Y ]; then
                        echo "Starting $ser"
                        systemctl enable $ser --now
                else
                        echo "Exiting without starting service"
                fi
fi
[root@mlinux1 ~]# ./script24.sh
Pl type the name of service to check: crond
crond is not running
Would you like to start crond [Y/N]: Y
Starting crond
Created symlink from /etc/systemd/system/multi-user.target.wants/crond.service
[root@mlinux1 ~]# systemctl is-active crond
active
[root@mlinux1 ~]# systemctl is-enabled crond
enabled
```

```
[root@mlinux1 ~]# cat script25.sh
#!/bin/bash
#Script to do calculations with case stat and elif cond with compound expr
calc(){
read -p "Pl select an operator[ADD|SUB|MUL|DIV|EXIT]: " op
op=`echo $op|tr [a-z] [A-Z]`
if [ "$op" = "ADD" ];then
        echo "Add=$(($a+$b))"

elif [ "$op" = "SUB" ];then
        echo "Sub=$(($a-$b))"

elif [ "$op" = "MUL" ];then
        echo "Mul=$(($a*$b))"

elif [ "$op" = "DIV" ];then
        echo "Div=$(($a/$b))"

elif [ "$op" = "EXIT" ];then
        echo -e "\e[33mExiting the script\e[0m"; exit

else
        echo -e "\e[31mInvalid operator selected, try again...\e[0m"
fi
calc
}

read -p "Pl give 1st val: " a
read -p "Pl give 2nd val: " b
if [ -z "$a" -o -z "$b" ];then
        echo -e "\e[31m Invalid Input\e[0m"
        exit 1
fi

calc
```

```
[root@mlinux1 ~]# cat script26.sh
#!/bin/bash
#Script to use while loop
read -p "Pl give number of iterations: " i
while [ $i -gt 0 ]
do
        echo "Iteration no: $i"
        sleep 1
i=$(($i-1))
done
[root@mlinux1 ~]# ./script26.sh
Pl give number of iterations: 5
Iteration no: 5
Iteration no: 4
Iteration no: 3
Iteration no: 2
Iteration no: 1
```

```
[root@mlinux1 ~]# cat script27.sh
#!/bin/bash
#Script to use until loop
read -p "Pl give number of iterations: " i
until [ $i -le 0 ]
do
        echo "Iteration no: $i"
        sleep 1
i=$(($i-1))
done
[root@mlinux1 ~]# ./script27.sh
Pl give number of iterations: 5
Iteration no: 5
Iteration no: 4
Iteration no: 3
Iteration no: 2
Iteration no: 1
```

```
[root@mlinux1 ~]# cat script28.sh
#!/bin/bash
#Script to do calculations with case statement
calc(){
for op in ADD SUB MUL DIV EXIT
do
        case $op in
                ADD)echo "Add=$(($a+$b))" ;;
                SUB)echo "Sub=$(($a-$b))" ;;
                MUL)echo "Mul=$(($a*$b))" ;;
                DIV)echo "Div=$(($a/$b))" ;;
                EXIT)echo -e "\e[33mExiting the script\e[0m"; exit ;;
                #*)echo -e "\e[31mInvalid operator selected, try again...\e[0m" ;;
        esac
done
}
read -p "Pl give 1st val: " a
read -p "Pl give 2nd val: " b
if [ -z "$a" -o -z "$b" ];then
        echo -e "\e[31m Invalid Input\e[0m"
        exit 1
fi
calc
[root@mlinux1 ~]# ./script28.sh
Pl give 1st val: 100
Pl give 2nd val: 2
Add=102
Sub=98
Mul=200
Div=50
Exiting the script
```

```
[root@mlinux1 ~]# cat script29.sh
#!/bin/bash
#Script to do calculations with select loop
calc(){
PS3="Pl Select an option: "
select op in ADD SUB MUL DIV EXIT
do
        case $op in
                ADD)echo "Add=$(($a+$b))" ;;
                SUB)echo "Sub=$(($a-$b))" ;;
                MUL)echo "Mul=$(($a*$b))" ;;
                DIV)echo "Div=$(($a/$b))" ;;
                EXIT)echo -e "\e[33mExiting the script\e[0m"; exit ;;
                *)echo -e "\e[31mInvalid operator selected, try again...\e[0m" ;;
        esac
done
}
read -p "Pl give 1st val: " a
read -p "Pl give 2nd val: " b
if [ -z "$a" -o -z "$b" ];then
        echo -e "\e[31m Invalid Input\e[0m"
        exit 1
fi
calc
[root@mlinux1 ~]# ./script29.sh
Pl give 1st val: 100
Pl give 2nd val: 20
1) ADD
2) SUB
3) MUL
4) DIV
5) EXIT
Pl Select an option: 1
Add=120
Pl Select an option: 2
Sub=80
Pl Select an option: 5
Exiting the script
```

```
[root@mlinux1 ~]# cat script30.sh
#!/bin/bash
#Script to use getopts
while getopts a:b:o: opt
do
        case $opt in
                a)a=$OPTARG;;
                b)b=$OPTARG;;
                o)o=$OPTARG;;
        esac
done

echo -e "\na=$a\nb=$b\no=$o\n"

case $o in
                ADD)echo "Add=$(($a+$b))" ;;
                SUB)echo "Sub=$(($a-$b))" ;;
                MUL)echo "Mul=$(($a*$b))" ;;
                DIV)echo "Div=$(($a/$b))" ;;
                *)echo -e "\e[31mInvalid operator selected, try again...\e[0m" ;;
esac

[root@mlinux1 ~]# ./script30.sh -a 10 -b 2 -o ADD

a=10
b=2
o=ADD

Add=12
```

```
[root@mlinux1 ~]# cat script31.sh
#!/bin/bash
#Script to use getopts with usage stmt
usage(){
        echo "Usage: `echo $0` -a val -b val -o operator[ADD|SUB|MUL|DIV]"
        echo "Usage: `echo $0` -a 100 -b 20 -o ADD"
        exit
}
if [ $# -le 0 ];then
        usage
fi

while getopts a:b:o: opt
do
        case $opt in
                a)a=$OPTARG;;
                b)b=$OPTARG;;
                o)o=$OPTARG;;
        esac
done

echo -e "\na=$a\tb=$b\to=$o\n"

case $o in
                ADD)echo "Add=$(($a+$b))" ;;
                SUB)echo "Sub=$(($a-$b))" ;;
                MUL)echo "Mul=$(($a*$b))" ;;
                DIV)echo "Div=$(($a/$b))" ;;
                *)echo -e "\e[31mInvalid operator selected, try again...\e[0m" ;;
esac

[root@mlinux1 ~]# ./script31.sh
Usage: ./script31.sh -a val -b val -o operator[ADD|SUB|MUL|DIV]
Usage: ./script31.sh -a 100 -b 20 -o ADD
[root@mlinux1 ~]# ./script31.sh -a 100 -b 20 -o MUL

a=100   b=20    o=MUL

Mul=2000
```

```
[root@mlinux1 ~]# cat script32.sh
#!/bin/bash
#Script to use trap option
trap '' 1 2 20 ##traping a signal, not to disturb the script
echo "Backup of system started"
sleep 1
echo "Backup is in progess, pl don't disturb the script"
sleep 10
echo -e "\nBackup completed sucessfully"
[root@mlinux1 ~]#
[root@mlinux1 ~]# ./script32.sh
Backup of system started
Backup is in progess, pl don't disturb the script
^C^C^C^C^C^Z^Z^Z^Z^Z^Z^Z^Z
Backup completed sucessfully
```

# SOME ADDITIONAL IMPORTANT STUFF AND SCRIPTS

**Here Document or heredoc**

A here document (or ***heredoc***) is a way of getting text input into a script without having to feed it in from a separate file. If you've got a small quantity of data you don't expect to change often, it's a quick and tidy way of keeping script and data together. For more complicated data or scripts, separating the two is still usually a better idea.

The basic version of this in bash looks like this:

```
#!/bin/bash
cat << EOF
several lines of
my data
listed here
EOF
```

***Output***

```
several lines of
my data
listed here
```

A heredoc can also be used to comment or inactive multiple lines instead of using ***"#"***.
An example for commenting multiple lines with heredoc is as follows.

```
#!/bin/bash
### Numerical operations using while loop.
read -p 'Pl Enter Number for iterations: ' i
<<EOF if [ -n $i ]; then    ############## from here
        test $i -eq 0 >/dev/null 2>1
        if [ $? -eq 2 ]; then
                echo "Pl enter only integers"
                exit 2
        fi
else
        echo "Pl enter a value."
        exit 1
fi                      ################ till here got commented
EOF
while [ $i -gt 0 ]
do
        read -p 'Enter the 1st value: ' a
        read -p 'Enter the 2nd value : ' b
        if [ -z $a -o -z $b ]; then
                echo "Invalid Input"
                exit 1
        fi
```

# SOME USEFUL SCRIPTS

1. Script to download data from ftp server using here document

```
#!/bin/bash
#scrip to download multiple files from ftp
server=192.168.10.81  ##server address
user=ftp              ##user name
pass=anything         ##password
dir=pub/down
file=*
#Giving instructions to connect to ftp server and downloading data
ftp -n $server <<FTP  ##Start of heredoc
quote USER $user
quote PASS $pass
bin
prompt
hash
cd $dir
mget $file
FTP     ##End of heredoc
```



2. Script to take backup of a filesystem and storing in a directory

```
$vim backup.sh
#!/bin/bash
#Script to take backup of /etc filesystem
clear
echo "Backup of etc started"
    sleep 2
tar -zcvf /opt/etc.tar.gz /etc
if [ $? -eq 0 ];then
      clear
      echo "Backup is Successful"
else
      clear
      echo "Backup failed"
fi
```

**3.** Script to take backup of /etc and transfer it to other server

```
$vim bkp2.sh
#!/bin/bash
##SCRIPT TO TAKE BACKUP
src=/etc                    ##file to be backed up
dest=/opt                   ## destination  to store backup
server=192.168.10.61        ##Sever to transfer backup
date=`date +%Y%m%d`
clear
echo "BACKUP OF /etc/ IS STARTING"
sleep 2
tar -czvf $dest/$src-$date.tgz $src
if [ $? -eq 0 ];then
      clear
      echo "BACKUP IS SUCCESFULL"

else
      echo "BACKUP FAILED"
      exit
fi
      sleep 2
      echo "COPYING THE FILE ON 61 SERVER"
      scp $dest$src* $server:/opt
if [ $? -eq 0 ];then
      echo "copying done successfully"
else
      echo "Copying failed"
fi
```

***Note:*** if there is a trusted relationship or password less login is configured the files will be automatically transferred, else it will wait for manual entry of password.

**4.** Taking backup of the user given input file, storing on user defined destination and transferring to other server.

```
#!/bin/bash
##SCRIPT TO TAKE BACKUP
server=192.168.10.61
date=`date +%Y%m%d`
clear
read -p 'Pl enter full path of the file to backup: ' file
read -p 'Pl enter the destination to store the backup: ' dest
if [ -z $file -o -z $dest ]; then
      echo "Invalid Input"
      exit
fi
echo "BACKUP IS STARTING"
sleep 2
```

```
tar -czvf $dest/$file-$date.tgz $file
if [ $? -eq 0 ];then
      clear
      echo "BACKUP IS SUCCESFULL"

else
      echo "BACKUP FAILED"
      exit
fi
      sleep 2
      echo "COPYING THE FILE ON 61 SERVER"
      scp $dest/$file* $server:/opt
if [ $? -eq 0 ];then
      echo "copying done successfully"
else
      echo "Copying failed"
fi
```

***Note:*** if there is a trusted relationship or password less login is configured the files will be automatically transferred, else it will wait for manual entry of password.

**5.** Script to monitor CPU usage and send mails as per critical and warning situation

```
$vim cpumon.sh

#!/bin/bash
##PurPose: Real time CPU Utilization Monitoring Shell Script
##Date: 13th Nov 2017
HOSTNAME=`hostname`
PATHS="/"
WARNING=90
CRIT=98
CAT=/bin/cat
MAILER=/bin/mail
CRITmailto="YOUREMAIL@DOMAIN.COM"  ##replace your email for critical alerts
mailto="YOUREMAIL@DOMAIN.COM"      ##replace your email for warning alerts
mkdir -p /var/log/cpuhistory
LOGFILE=/var/log/cpuhistory/hist-`date +%h%d%y`.log
touch $LOGFILE
for path in $PATHS
do
CPU_LOAD=`top -b -n 2 -d1 | grep "Cpu(s)" | tail -n1 | awk '{print $2}' |awk -F. '{print $1}'`
if [ -n "$WARNING" -a -n "$CRIT" ]; then
        if [ "$CPU_LOAD" -ge "$WARNING" -a "$CPU_LOAD" -lt "$CRIT" ]; then
                echo " `date "+%F %H:%M:%S"` WARNINGING - $CPU_LOAD on Host $HOSTNAME" >> $LOGFILE
                echo "CPU Load is Warning $CPU_LOAD on $HOSTNAME" | $MAILER -s "CPU Load is Warning $CPU_LOAD on $HOSTNAME" $mailto
                exit 1
```

```
elif [ "$CPU_LOAD" -ge "$CRIT" ]; then
        echo "`date "+%F %H:%M:%S"` CRITICAL - $CPU_LOAD on $HOSTNAME" >> $LOGFILE
        echo "CPU Load is Critical $CPU_LOAD on $HOSTNAME" | $MAILER -s "CPU Load is Critical $CPU_LOAD on $HOSTNAME" $CRITmailto
        exit 2
else
        echo "`date "+%F %H:%M:%S"` OK - $CPU_LOAD on $HOSTNAME" >> $LOGFILE
        exit 0
        fi
fi
done
```

*My dear students/aspirants, scripting is all about logic and creativity. There are many situations in which you can use scripting for automation, monitoring and other important activities. It's all about your necessity and requirements which you can put in a script and make your work easy. After all "**Necessity is the mother of all inventions**". So, keep inventing and exploring the power of scripting, you'll never be bored of it and will always be surprised and admire it. **–Musabuddin Syed**.*