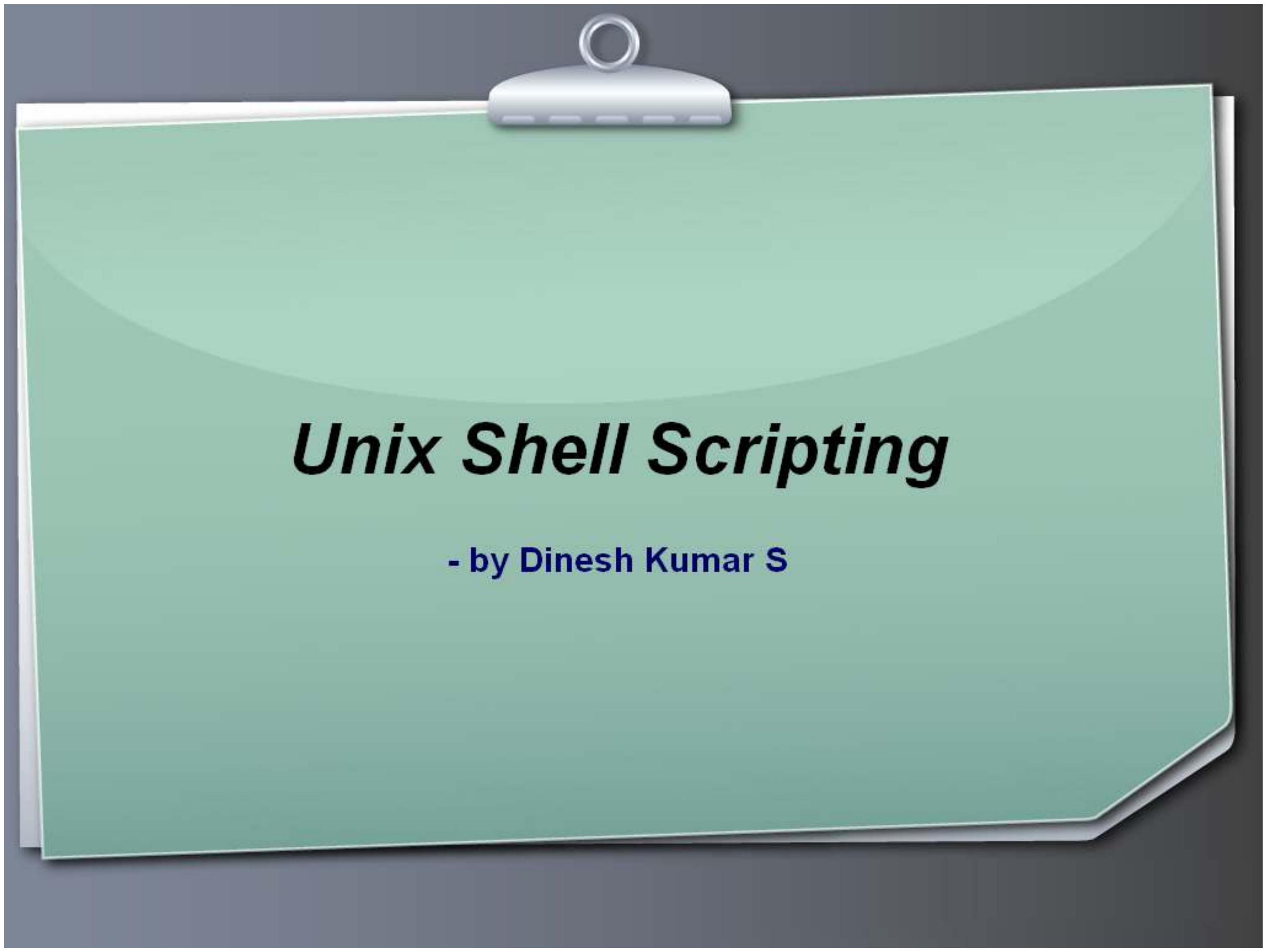


Unix Shell Scripting

- by Dinesh Kumar S

Contents

Chapters	Topic	Page no.
Chapter 1	Introduction	3
Chapter 2	SSH Client	4
Chapter 3	Unix Shells	8
Chapter 4	Text Editors	11
Chapter 5	A Beginning to Shell Scripting	19
Chapter 6	Operators	33
Chapter 7	Variables Manipulation (Advance)	39
Chapter 8	Conditional Statements	43
Chapter 9	Looping Statements	47
Chapter 10	Control Statements	74
Chapter 11	Functions	79

Chapter 1

Introduction

Linux:

It is an operating system based on **UNIX**.

Kernel:

It is the backbone of Linux OS, which is used to manage resources of Linux OS like memory, I/O, software, hardware management processes.

User → Shell Script → Kernel → PC h/w

- User writes script.
- Script contains instructions.
- Kernel interprets the instruction in machine language.
- As per the instruction kernel controls the PC hardware.

Shell script:

It's a collection of OS commands or instructions.

Advantages of Shell Script:

Script is always a platform independent.
Performance will be faster than programming languages.
Very easy to debug.

Chapter 2

SSH Client

Secure Shell (or) SSH is a network protocol that is used to exchange or share information between two different networks.

This is used on Linux & UNIX systems to access SHELL accounts.

All the information exchanged/transmitted between networks is encrypted.

It uses public key cryptography to authenticate remote computer user.

Free Serial, Telnet, and SSH client

- **Putty**
- **Tera Term**

Putty:

It is a terminal emulator application which acts as client for SSH, Telnet, rLogin.

Download: http://www.putty.org/

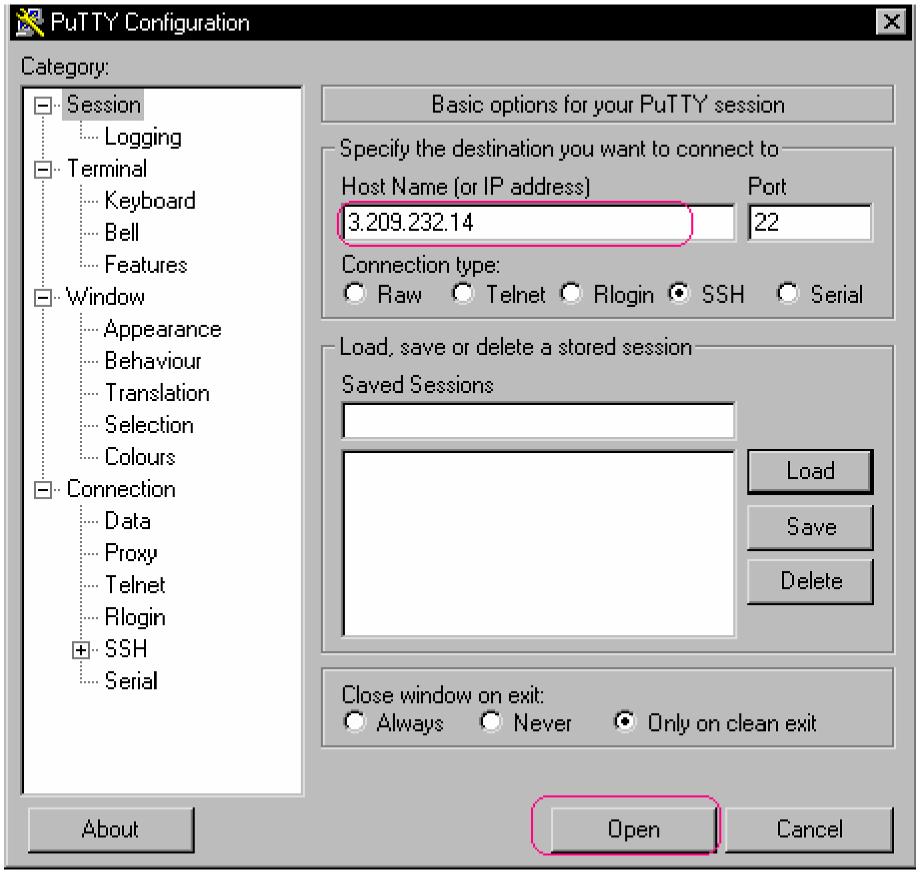
Tera Term:

It's an open source terminal emulator. It emulates different terminals from VT100 to VT382. It also supports telnet, SSH1, SSH2 and serial connections.

Download: http://hp.vector.co.jp/authors/VA002416/teraterm.html

Putty:

Double click 'Putty.exe' then,



Specify the destination you want to connect to

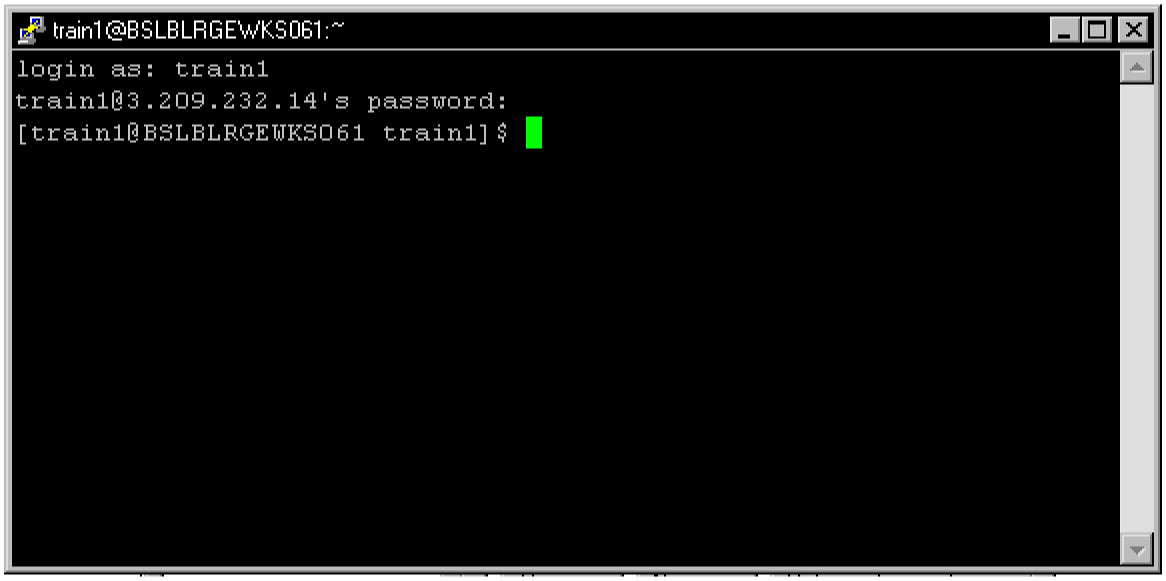
Host Name (or IP address): 3.209.232.14

Port: 22

Connection type: Raw Telnet Rlogin [✓ SSH] Serial

Close window on exit: Always Never [✓ Only on clean exit]

Input the server IP Address and open the session. Provide Username/password.



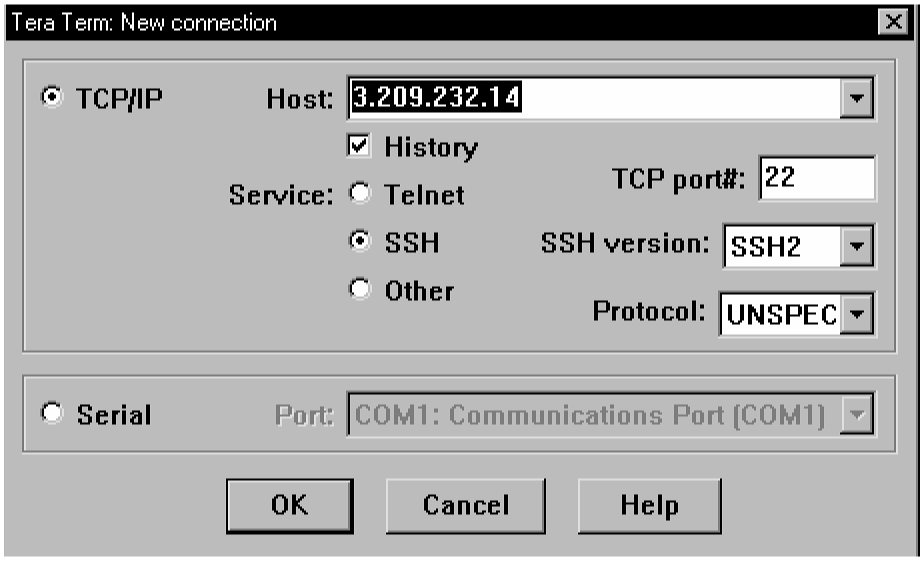
train1@BSLBLRGEWKS061:~

```
login as: train1
train1@3.209.232.14's password:
[train1@BSLBLRGEWKS061 train1]$
```

Tera Term:

Double click 'TeraTrain.exe' then.

Input the IP Address. Then select the options as per the screen shot below and press 'ok'.


Tera Term: New connection

[✓ TCP/IP] Host: 3.209.232.14

[✓ History]

Service: Telnet [✓ SSH] Other

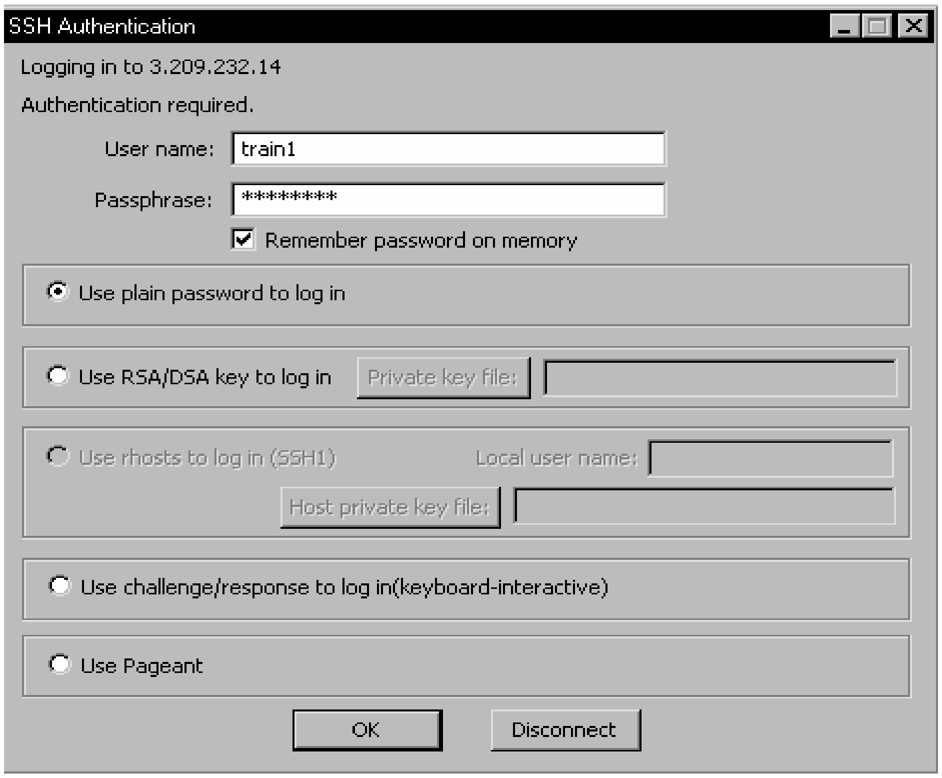
TCP port#: 22

SSH version: SSH2

Protocol: UNSPEC

Serial Port: COM1: Communications Port (COM1)

Now provide the username & password.



SSH Authentication

Logging in to 3.209.232.14

Authentication required.

User name: train1

Passphrase: *********

[✓] Remember password on memory

(●) Use plain password to log in

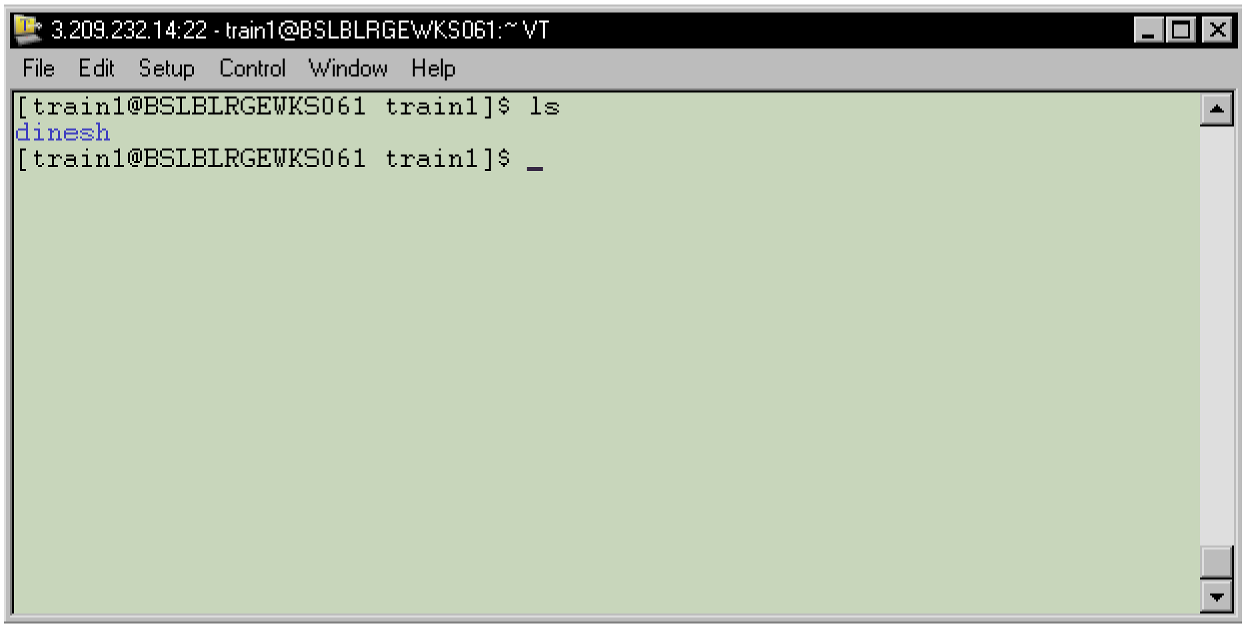
( ) Use RSA/DSA key to log in

( ) Use rhosts to log in (SSH1)

( ) Use challenge/response to log in(keyboard-interactive)

( ) Use Pageant

Then click "ok".



3.209.232.14:22 - train1@BSLBLRGEWKS061:~ VT

```
[train1@BSLBLRGEWKS061 train1]$ ls
dinesh
[train1@BSLBLRGEWKS061 train1]$ _
```

I will be using "**Tera Term**" SSH Client.

Chapter 3

UNIX Shells

UNIX shell is a command line like DOS in Windows. It's a user interface for UNIX operating system. Mainly shells are used for inputting user OS commands.

It is called "Shell" because it hides all the information behind the shell interface.

Types of Shells:

Bourne Shell (sh):

It's the default UNIX shell. Most of the scripts to configure OS is written using this shell. It was developed by Stephen Bourne.

C Shell (csh):

It is called C shell because the syntax used here is similar to c language. It adds many features compare to bourne shell. This shell is not widely used now. It was developed by Bill Joy.

Korn Shell (ksh):

This shell is backward compatible with bourne shell & inherits many features of C shell. This was developed by David Korn.

Bash Shell (bash):

It stands for Bourne again Shell i.e. It is superset of bourne shell. It was built by Stephen Bourne.

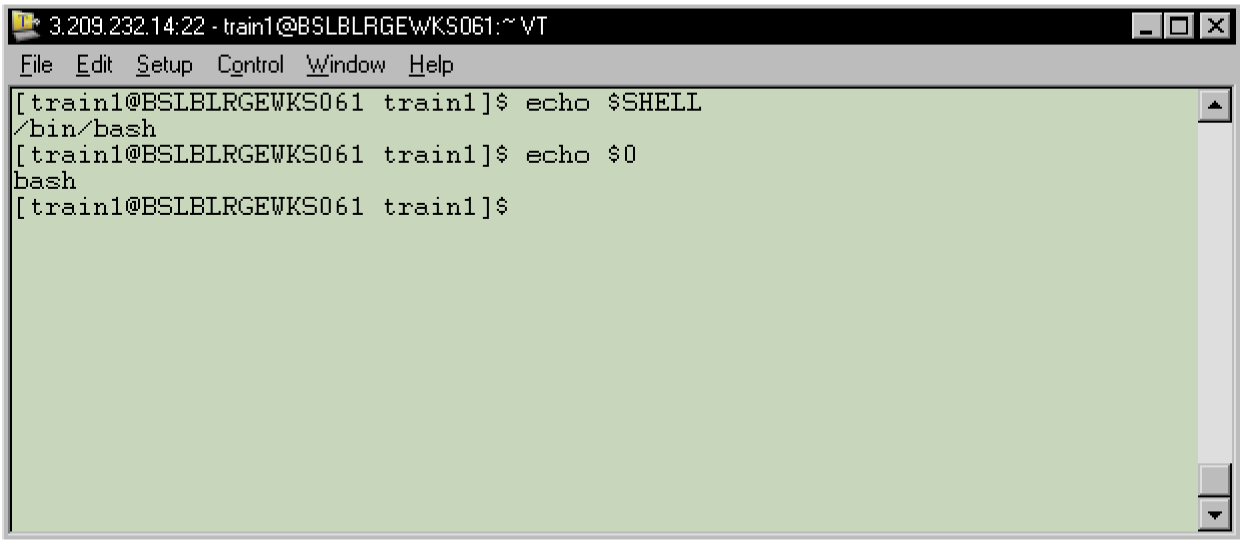
Note: I have discussed only main shells used in UNIX.

Finding Shell:

How to find which shell we are working at?
There is a simple command to check which shell we are working at.

echo $SHELL (or)

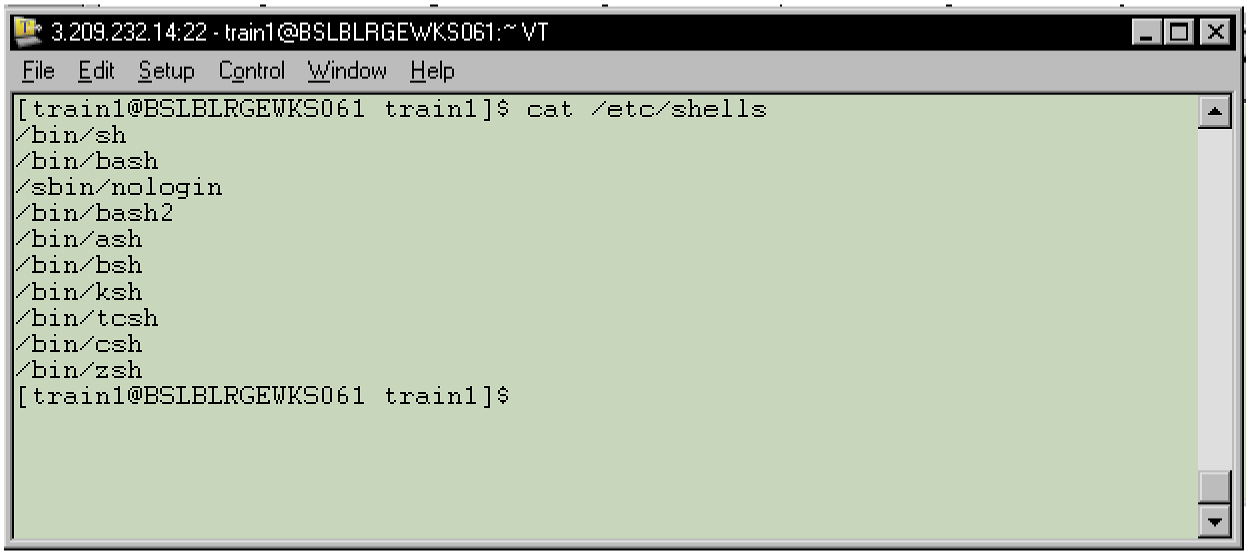
echo $0


3.209.232.14:22 - train1@BSLBLRGEWKS061:~ VT
File Edit Setup Control Window Help

```
[train1@BSLBLRGEWKS061 train1]$ echo $SHELL
/bin/bash
[train1@BSLBLRGEWKS061 train1]$ echo $0
bash
[train1@BSLBLRGEWKS061 train1]$
```

To check Shells available in UNIX, type the following command

Cat /etc/shells


3.209.232.14:22 - train1@BSLBLRGEWKS061:~ VT
File Edit Setup Control Window Help

```
[train1@BSLBLRGEWKS061 train1]$ cat /etc/shells
/bin/sh
/bin/bash
/sbin/nologin
/bin/bash2
/bin/ash
/bin/bsh
/bin/ksh
/bin/tcsh
/bin/csh
/bin/zsh
[train1@BSLBLRGEWKS061 train1]$
```

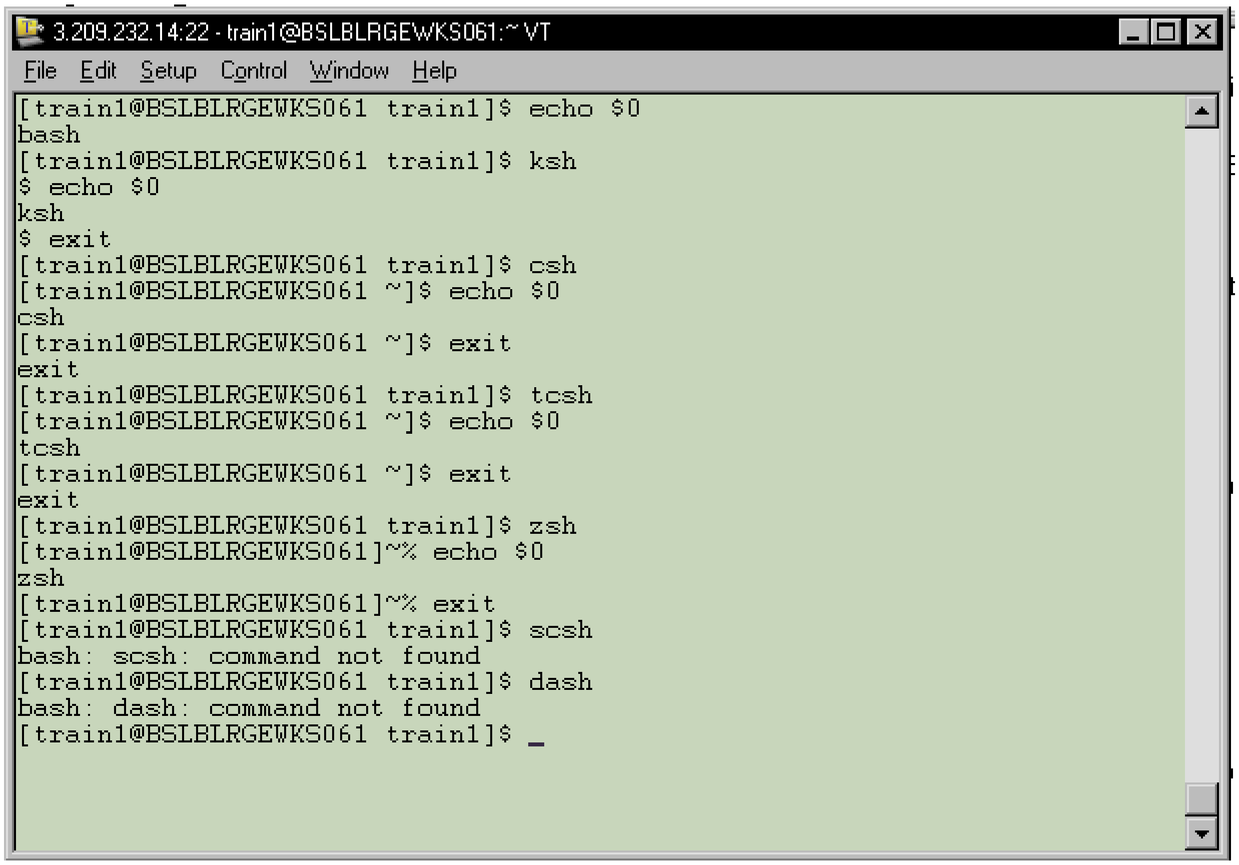
Switching Shell:

We can switch between shells in two ways:

- Temporarily
- Permanently

Temporarily:

By default we will be using bash shell. After logging into the UNIX we can change to other shell in same session. This is done by entering the 'name' of the shell.



3.209.232.14:22 - train1@BSLBLRGEWKS061:~ VT
File Edit Setup Control Window Help

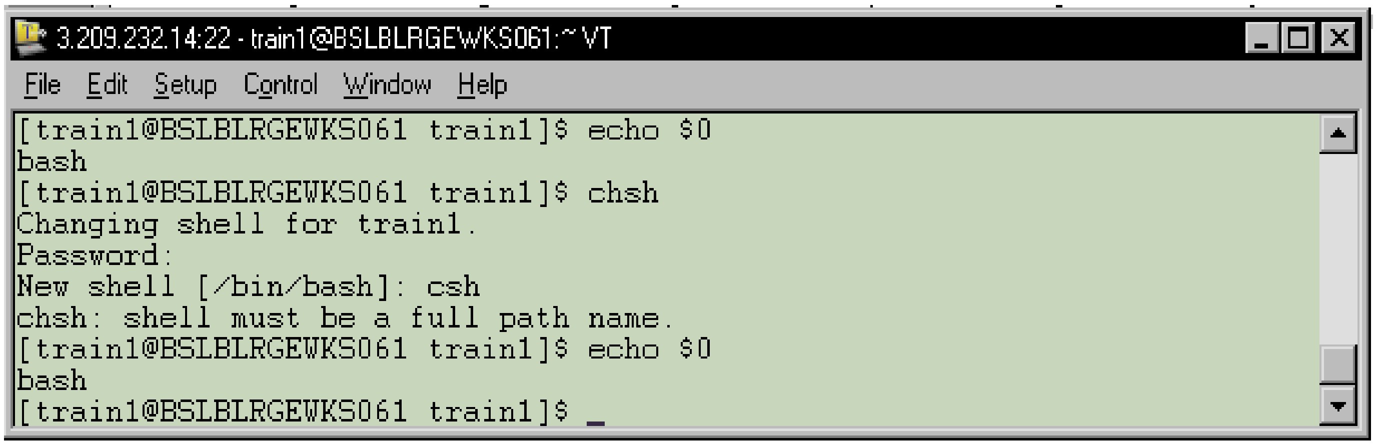
```
[train1@BSLBLRGEWKS061 train1]$ echo $0
bash
[train1@BSLBLRGEWKS061 train1]$ ksh
$ echo $0
ksh
$ exit
[train1@BSLBLRGEWKS061 train1]$ csh
[train1@BSLBLRGEWKS061 ~]$ echo $0
csh
[train1@BSLBLRGEWKS061 ~]$ exit
exit
[train1@BSLBLRGEWKS061 train1]$ tcsh
[train1@BSLBLRGEWKS061 ~]$ echo $0
tcsh
[train1@BSLBLRGEWKS061 ~]$ exit
exit
[train1@BSLBLRGEWKS061 train1]$ zsh
[train1@BSLBLRGEWKS061]~% echo $0
zsh
[train1@BSLBLRGEWKS061]~% exit
[train1@BSLBLRGEWKS061 train1]$ scsh
bash: scsh: command not found
[train1@BSLBLRGEWKS061 train1]$ dash
bash: dash: command not found
[train1@BSLBLRGEWKS061 train1]$ _
```

Bash - Bash Shell	zsh - Z Shell
Ksh - Korn Shell	scsh - Scheme Shell (not found)
Csh - C Shell	dash - Debain Almquist Shell (not found)
Tcsh - TENEX C Shell	

Permanently:

To change the shell permanently whenever we log in into the UNIX use the command chsh (Change Shell).

For these changes to be done we need to modify shell and environment variables.


3.209.232.14:22 - train1@BSLBLRGEWKS061:~ VT
File Edit Setup Control Window Help

```
[train1@BSLBLRGEWKS061 train1]$ echo $0
bash
[train1@BSLBLRGEWKS061 train1]$ chsh
Changing shell for train1.
Password:
New shell [/bin/bash]: csh
chsh: shell must be a full path name.
[train1@BSLBLRGEWKS061 train1]$ echo $0
bash
[train1@BSLBLRGEWKS061 train1]$ _
```

I have changed the shell from bash to csh (c Shell). When I check the current shell for the current session it is bash.

Now log in again & check which shell you are in for current session, it will be csh.

As I am using SSH client, we cannot change the shell.

Chapter 4

Text Editors

Text editors have similar functionality like word processors. Several text editors are available in Linux. We are going to see only major ones.

Editors:

Vim
Pico
Emacs
Joe

Vim:

It's an improved version of 'Vi'. It is also called as programmer's editor. Also it contains many power tools.

Pico:

It's a simple text editor with 'pine' e-mailer. It is very easy to use & powerful.

Emacs:

It's an extensible & customizable editor. This editor is user friendly & supports many languages.

Joe:

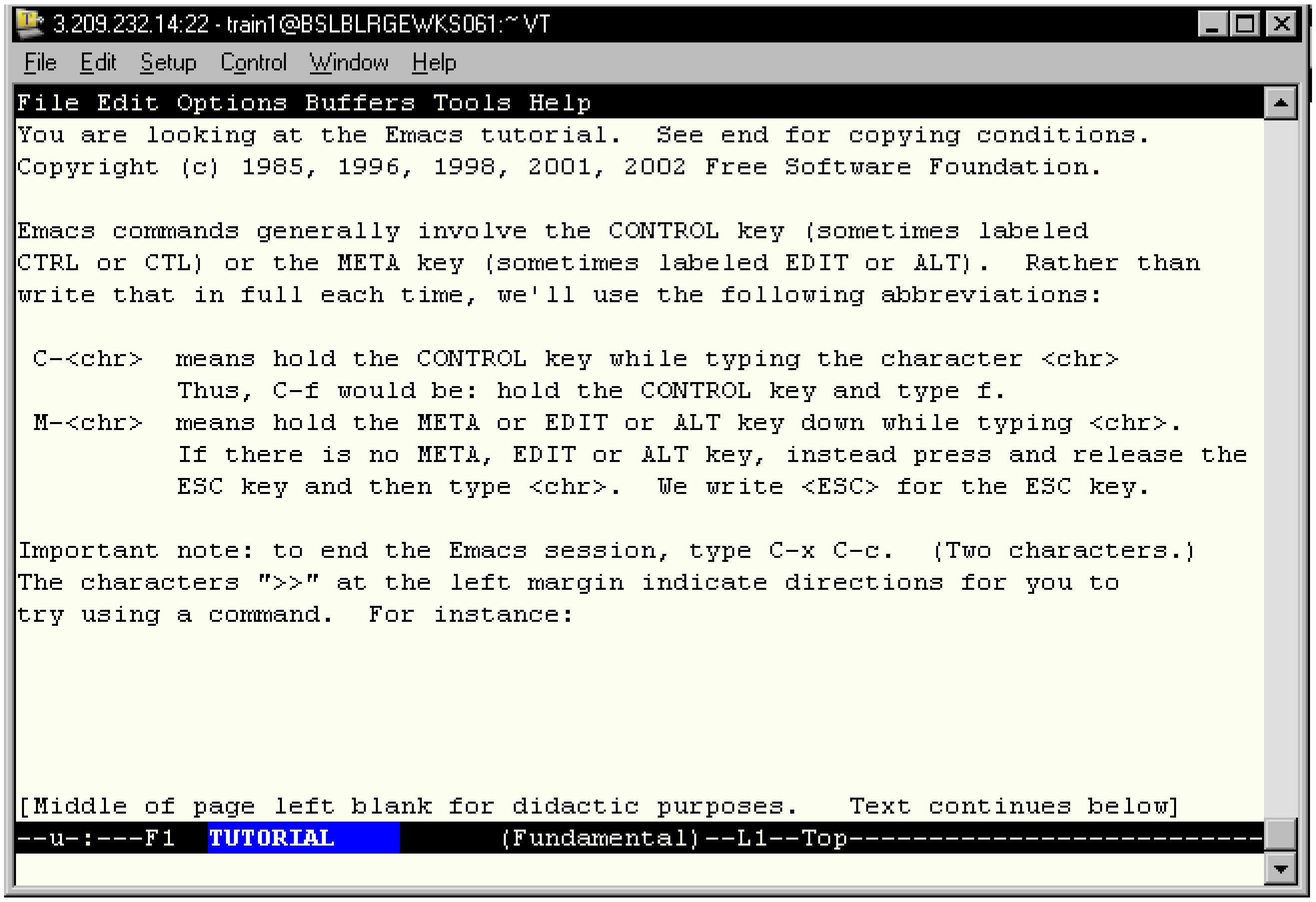
It's a full featured terminal based editor. It is very similar to WordStar & Emacs word processor.

Note: I am going to use '**Emacs**' for all exercises.

Basics Command in Emacs:

Help Commands:

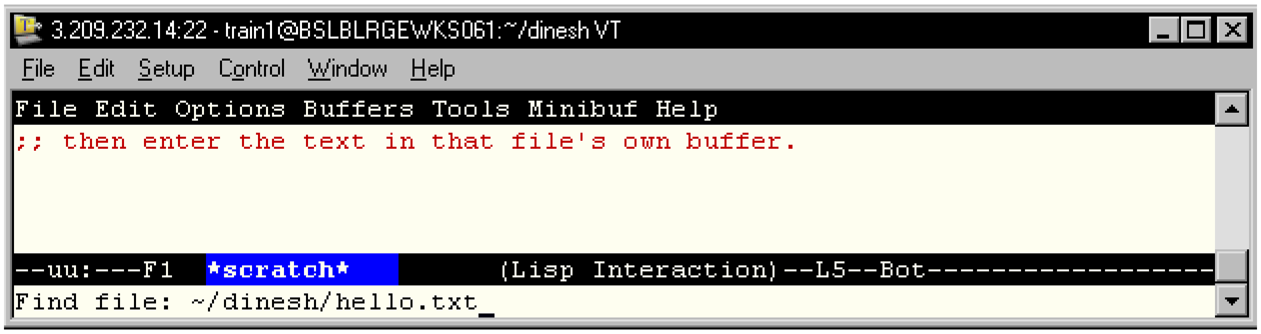
Ctrl + h	→ help command.
Ctrl + h t	→ help with tutorial


3.209.232.14:22 - train1@BSLBLRGEWKS061:~ VT
File Edit Setup Control Window Help
File Edit Options Buffers Tools Help

```
You are looking at the Emacs tutorial.  See end for copying conditions.
Copyright (c) 1985, 1996, 1998, 2001, 2002 Free Software Foundation.

Emacs commands generally involve the CONTROL key (sometimes labeled
CTRL or CTL) or the META key (sometimes labeled EDIT or ALT).  Rather than
write that in full each time, we'll use the following abbreviations:

 C-<chr>  means hold the CONTROL key while typing the character <chr>
          Thus, C-f would be: hold the CONTROL key and type f.
 M-<chr>  means hold the META or EDIT or ALT key down while typing <chr>.
          If there is no META, EDIT or ALT key, instead press and release the
          ESC key and then type <chr>.  We write <ESC> for the ESC key.

Important note: to end the Emacs session, type C-x C-c.  (Two characters.)
The characters ">>" at the left margin indicate directions for you to
try using a command.  For instance:






[Middle of page left blank for didactic purposes.   Text continues below]
--u-:---F1  TUTORIAL        (Fundamental)--L1--Top-------------------------
```

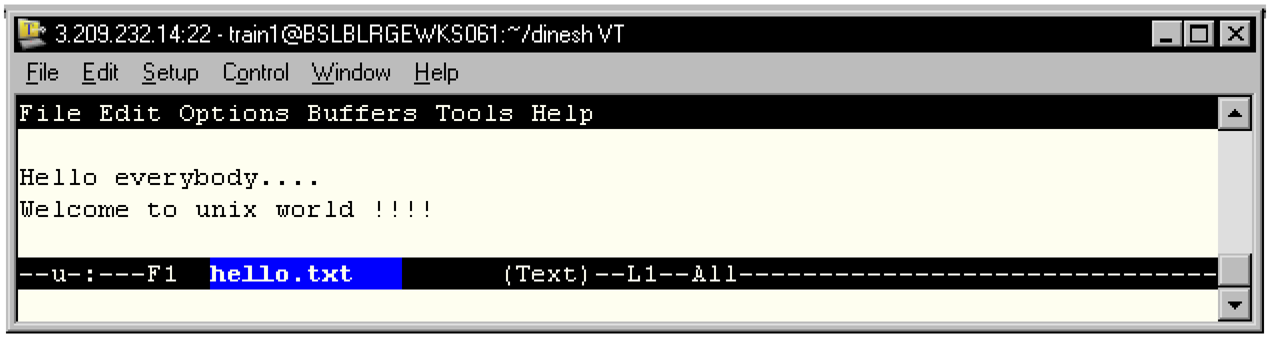
File Commands:

Ctrl x + Ctrl f → Finds file, it prompt for a file name & loads the file into editor.



3.209.232.14:22 - train1@BSLBLRGEWKS061:~/dinesh VT
File Edit Setup Control Window Help
File Edit Options Buffers Tools Minibuf Help

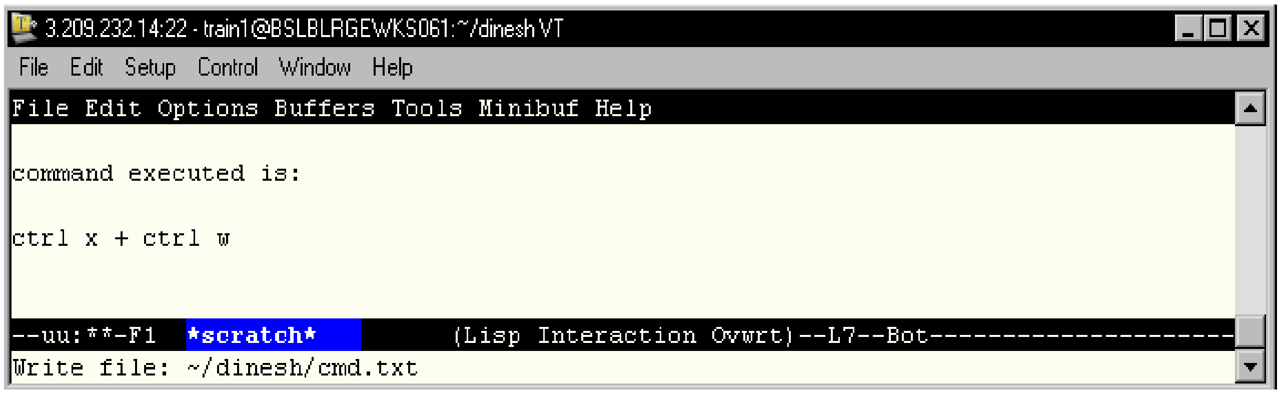
```
;; then enter the text in that file's own buffer.

--uu:---F1  *scratch*        (Lisp Interaction)--L5--Bot------------------
Find file: ~/dinesh/hello.txt_
```



3.209.232.14:22 - train1@BSLBLRGEWKS061:~/dinesh VT
File Edit Setup Control Window Help
File Edit Options Buffers Tools Help

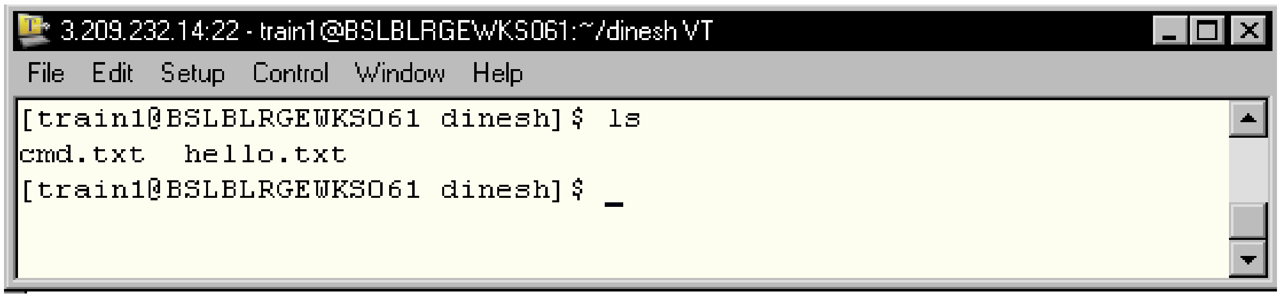
```

Hello everybody....
Welcome to unix world !!!!

--u-:---F1  hello.txt        (Text)--L1--All-------------------------------
```

Ctrl x + Ctrl s → Saves the buffer with associate file name.



3.209.232.14:22 - train1@BSLBLRGEWKS061:~/dinesh VT
File Edit Setup Control Window Help
File Edit Options Buffers Tools Minibuf Help

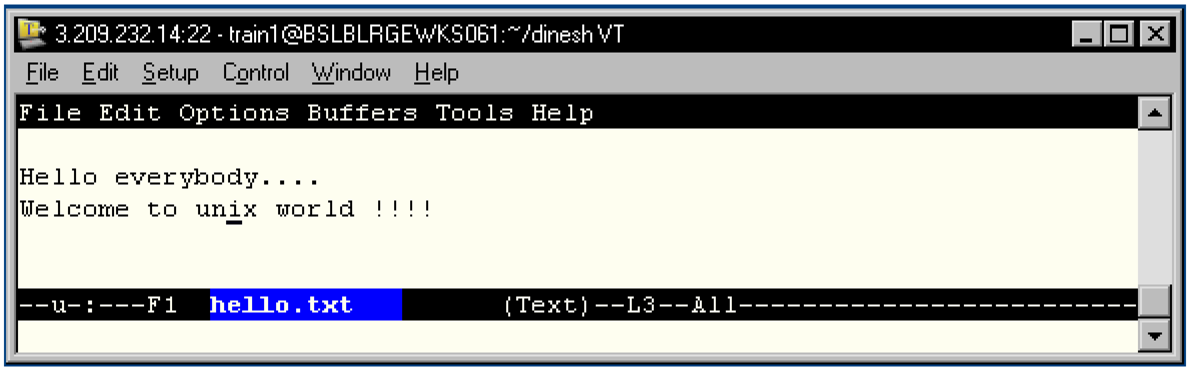
```

command executed is:

ctrl x + ctrl w


--uu:**-F1  *scratch*        (Lisp Interaction Ovwrt)--L7--Bot--------------------
Write file: ~/dinesh/cmd.txt
```



3.209.232.14:22 - train1@BSLBLRGEWKS061:~/dinesh VT
File Edit Setup Control Window Help

```
[train1@BSLBLRGEWKS061 dinesh]$ ls
cmd.txt   hello.txt
[train1@BSLBLRGEWKS061 dinesh]$ _
```

Cursor Movements:


3.209.232.14:22 - train1@BSLBLRGEWKS061:~/dinesh VT
File Edit Setup Control Window Help
File Edit Options Buffers Tools Help

```
Hello everybody....
Welcome to unix world !!!!

--u-:---F1  hello.txt        (Text)--L3--All------------------------
```

Ctrl + a → Move Cursor to beginning of line.

```
Hello everybody....
Welcome to unix world !!!!
```

Ctrl + e → Move cursor to end of line.

```
Hello everybody....
Welcome to unix world !!!!_
```

Ctrl + n → Move cursor to next line.

```
Hello everybody....
Welcome to unix world !!!!

_
```

Ctrl + p → Move Cursor Previous line.

```
Hello everybody....
Welcome to unix world !!!!
```

Esc + f → Move cursor one word forward.

```
Hello everybody....
Welcome_to unix world !!!!
```

Esc + b → Move cursor one word backward.

```
Hello everybody....
Welcome to unix world !!!!
```

Ctrl + f → Move cursor one character forward.

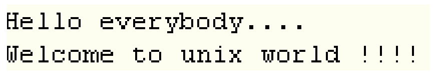
Ctrl + b → Move cursor one character back.

Ctrl + v → Scroll file forward by one screen.

Esc + v → Scroll file backward by one screen.

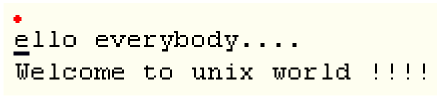
<u>Copy, Paste, Delete Commands</u>:

Original Text:



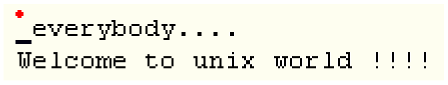
```
Hello everybody....
Welcome to unix world !!!!
```

Ctrl + d → Delete a char.



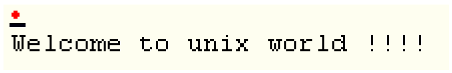
```
ello everybody....
Welcome to unix world !!!!
```

Esc + d → Delete word.



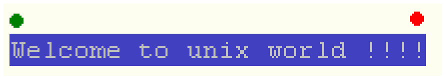
```
 everybody....
Welcome to unix world !!!!
```

Ctrl + k → Kill line.



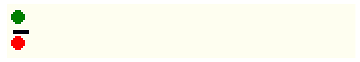
```

Welcome to unix world !!!!
```

Ctrl + @ → Set region.

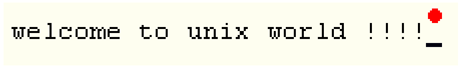


```
Welcome to unix world !!!!
```

Ctrl + w → Kill region.

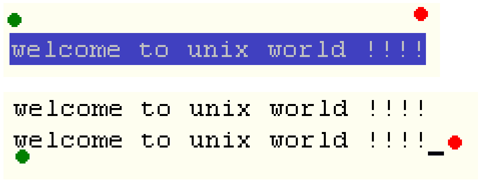


Ctrl + y → insert at cursor position.



```
welcome to unix world !!!!_
```

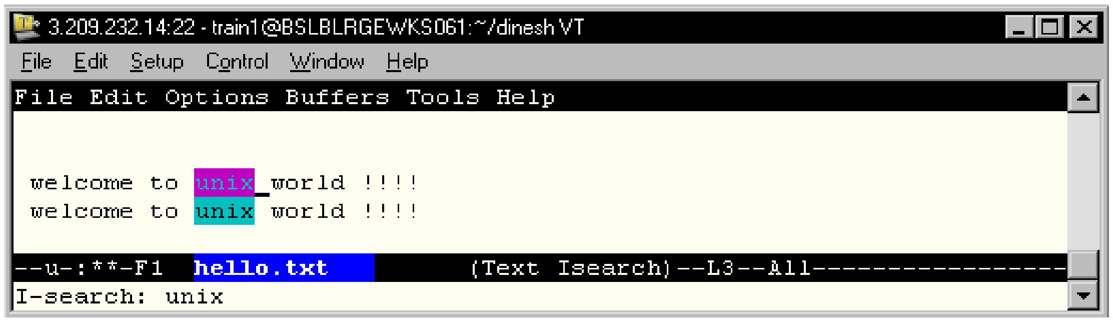
Esc + w → Copy region.



```
welcome to unix world !!!!

welcome to unix world !!!!
welcome to unix world !!!!_
```

Search Commands:

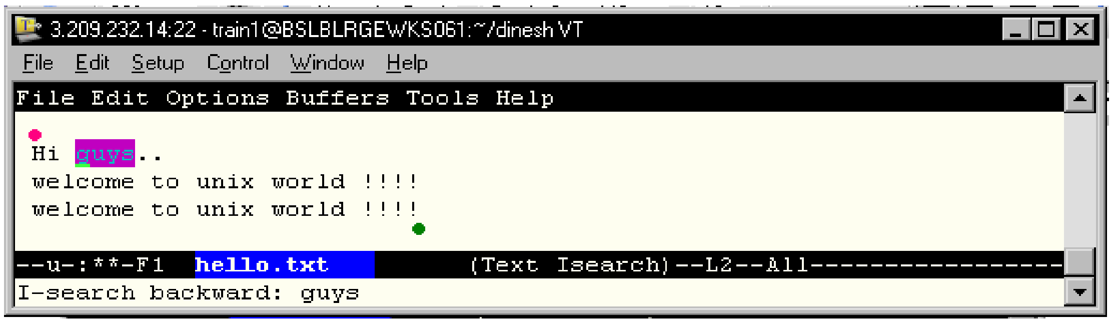
Ctrl + s → Search forward.



3.209.232.14:22 - train1@BSLBLRGEWKS061:~/dinesh VT
File Edit Setup Control Window Help

```
File Edit Options Buffers Tools Help

 welcome to unix_world !!!!
 welcome to unix world !!!!

--u-:**-F1  hello.txt        (Text Isearch)--L3--All----------------
I-search: unix
```

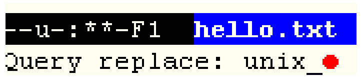
Ctrl + r → Search backward.



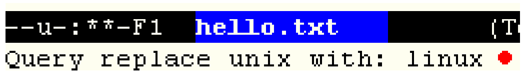
3.209.232.14:22 - train1@BSLBLRGEWKS061:~/dinesh VT
File Edit Setup Control Window Help

```
File Edit Options Buffers Tools Help

 Hi guys..
 welcome to unix world !!!!
 welcome to unix world !!!!

--u-:**-F1  hello.txt        (Text Isearch)--L2--All----------------
I-search backward: guys
```

Esc + % → Search & Replace.

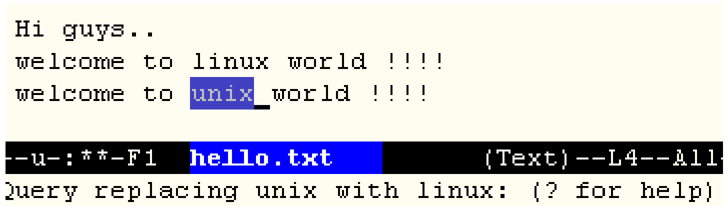


```
--u-:**-F1  hello.txt
Query replace: unix_
```



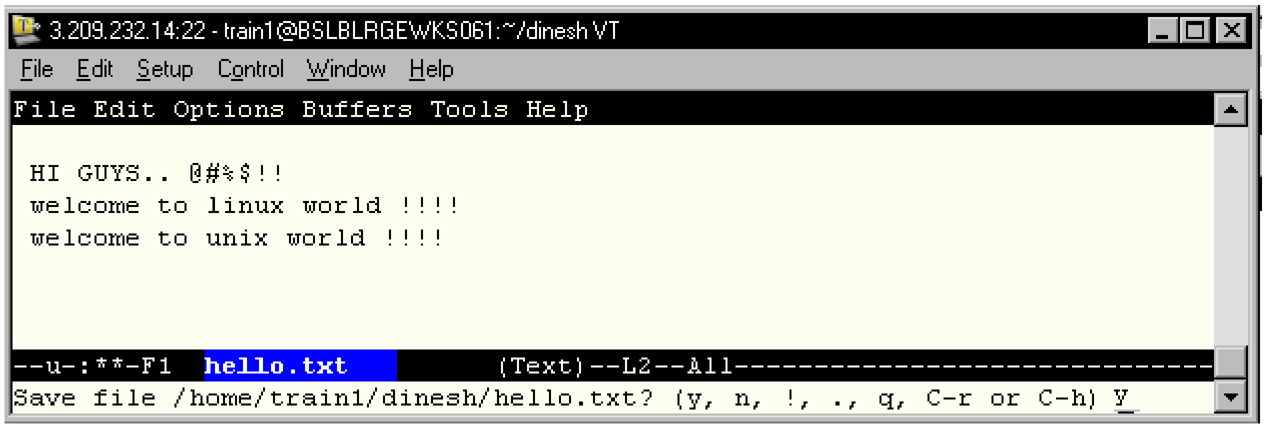
```
--u-:**-F1  hello.txt           (T
Query replace unix with: linux
```

Press 'y' to replace 'n' to skip.



```
Hi guys..
welcome to linux world !!!!
welcome to unix_world !!!!

--u-:**-F1  hello.txt        (Text)--L4--All
Query replacing unix with linux: (? for help)
```

Save & Exitcommands:



```
3.209.232.14:22 - train1@BSLBLRGEWKS061:~/dinesh VT
File Edit Setup Control Window Help
File Edit Options Buffers Tools Help

 HI GUYS.. @#%$!!
 welcome to linux world !!!!
 welcome to unix world !!!!

--u-:**-F1  hello.txt        (Text)--L2--All-------------------------------
Save file /home/train1/dinesh/hello.txt? (y, n, !, ., q, C-r or C-h) y_
```

Chapter 5

A Beginning to Shell Scripting

Rule to write Shell script:

Write Script

Give Execute
permission to user

Run Script

Debug (optional)

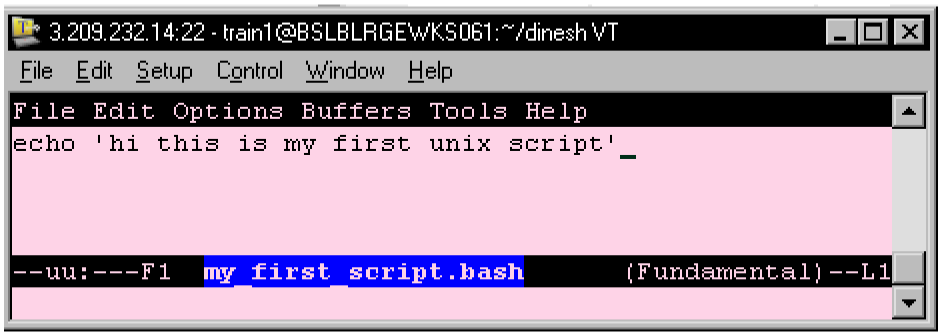
Write Script:

Create a shell script using a text editor (Emacs). Save the script file as,

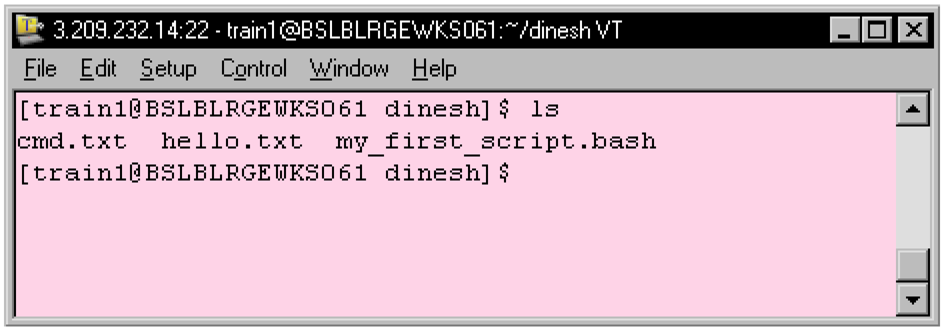
- .sh
- .bash

Example: emacs my_first_script.bash

emacs my_first_script.sh


3.209.232.14:22 - train1@BSLBLRGEWKS061:~/dinesh VT
File Edit Setup Control Window Help
File Edit Options Buffers Tools Help

```
echo 'hi this is my first unix script'_

--uu:---F1  my_first_script.bash       (Fundamental)--L1
```


3.209.232.14:22 - train1@BSLBLRGEWKS061:~/dinesh VT
File Edit Setup Control Window Help

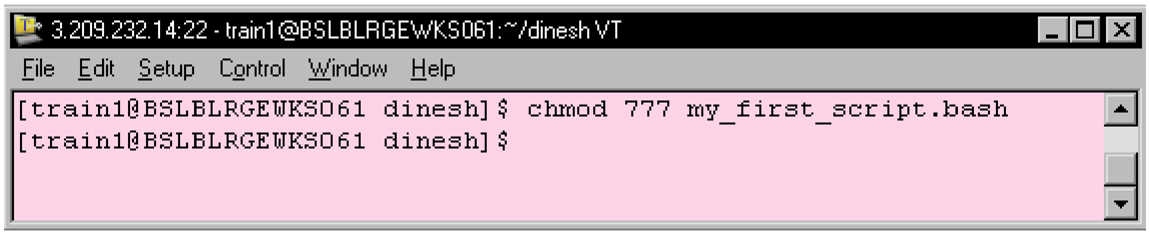
```
[train1@BSLBLRGEWKS061 dinesh]$ ls
cmd.txt   hello.txt   my_first_script.bash
[train1@BSLBLRGEWKS061 dinesh]$
```

Setting up Execute Permission:

Before executing the script you need to set permission to read, write and execute.

To set file permission use command,

Chmod 777 <script_name>

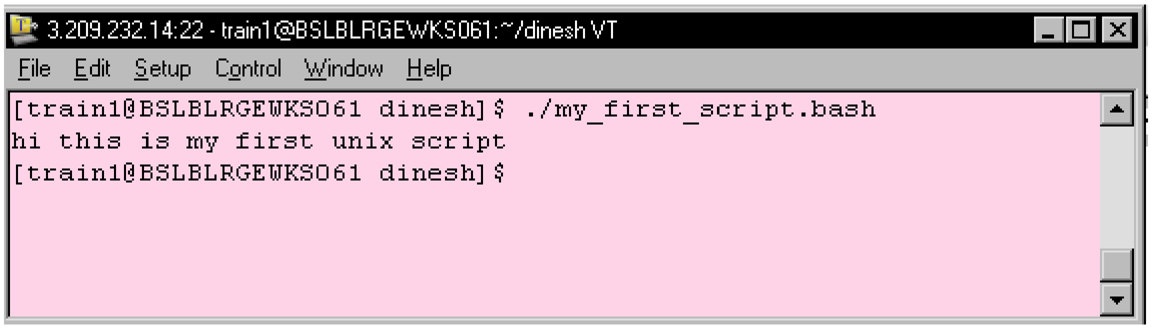

3.209.232.14:22 - train1@BSLBLRGEWKS061:~/dinesh VT
File Edit Setup Control Window Help

```
[train1@BSLBLRGEWKS061 dinesh]$ chmod 777 my_first_script.bash
[train1@BSLBLRGEWKS061 dinesh]$
```

Run Script:

Run the shell script as below,

Bash <script_name>
Sh <script_name>
./<script_name>


3.209.232.14:22 - train1@BSLBLRGEWKS061:~/dinesh VT
File Edit Setup Control Window Help

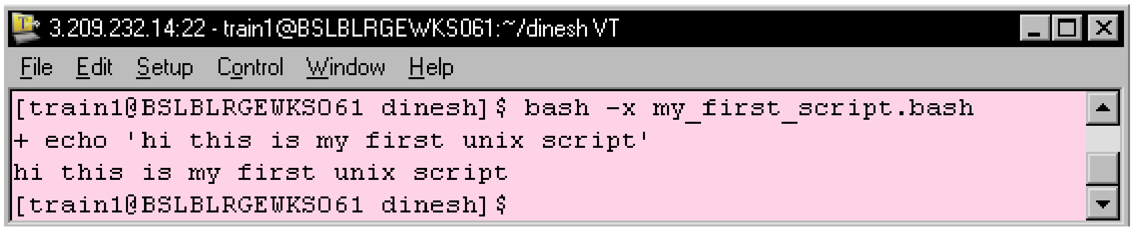
```
[train1@BSLBLRGEWKS061 dinesh]$ ./my_first_script.bash
hi this is my first unix script
[train1@BSLBLRGEWKS061 dinesh]$
```

Debug:

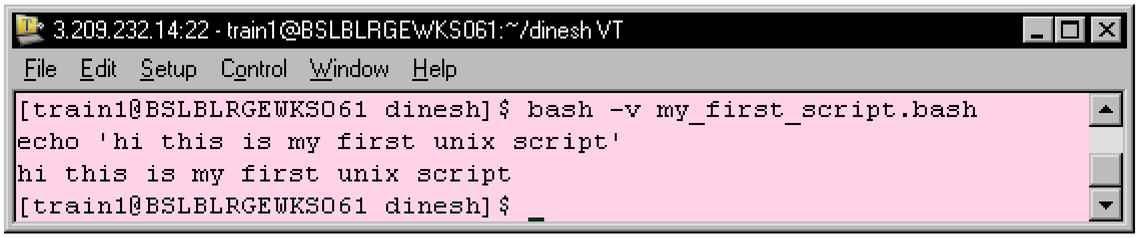
If there is an error in shell script, to find out the error we need to type the following command with options.

Bash <option> <script_name>
Sh <option> <script_name>

Options:
v → Print script line as they read.
x → While executing command it expands system variables and arguments.


3.209.232.14:22 - train1@BSLBLRGEWKS061:~/dinesh VT
File Edit Setup Control Window Help

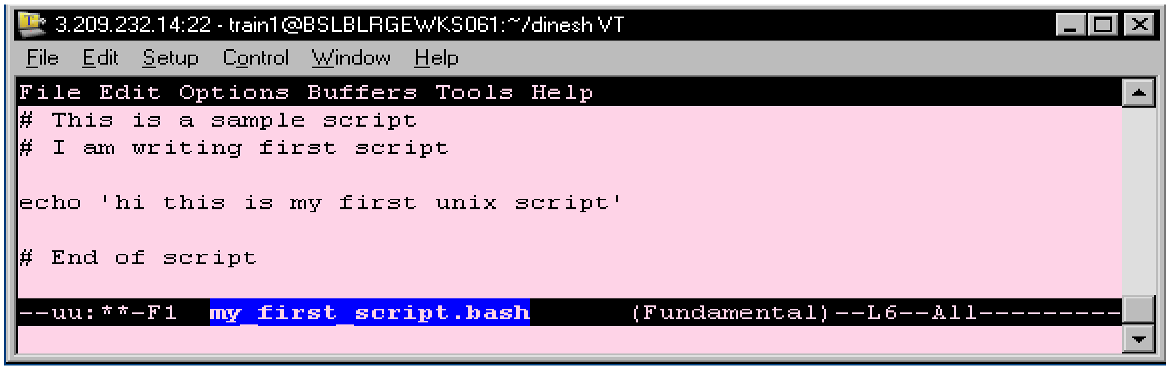
```
[train1@BSLBLRGEWKS061 dinesh]$ bash -x my_first_script.bash
+ echo 'hi this is my first unix script'
hi this is my first unix script
[train1@BSLBLRGEWKS061 dinesh]$
```


3.209.232.14:22 - train1@BSLBLRGEWKS061:~/dinesh VT
File Edit Setup Control Window Help

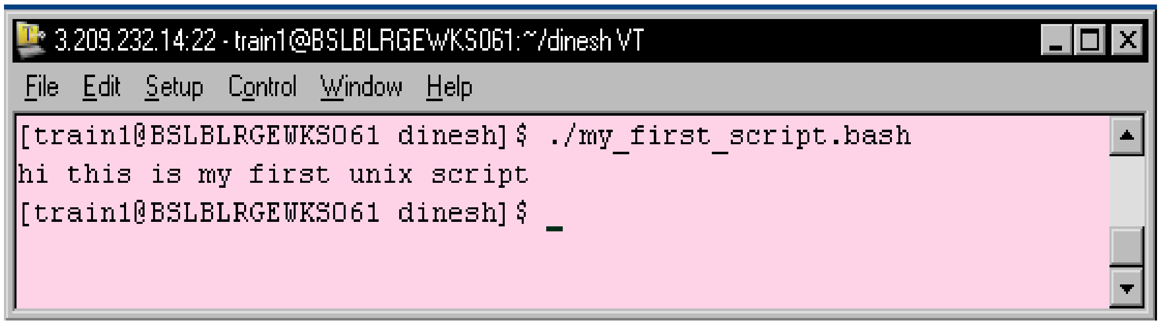
```
[train1@BSLBLRGEWKS061 dinesh]$ bash -v my_first_script.bash
echo 'hi this is my first unix script'
hi this is my first unix script
[train1@BSLBLRGEWKS061 dinesh]$ _
```

Comments in Shell Script:

To make the script understandable to other users we need to add comments inside shell script. To comment the lines add '#' before the line. When a line is commented that is ignored when the script is running.


3.209.232.14:22 - train1@BSLBLRGEWKS061:~/dinesh VT
File Edit Setup Control Window Help
File Edit Options Buffers Tools Help

```
# This is a sample script
# I am writing first script

echo 'hi this is my first unix script'

# End of script

--uu:**-F1  my_first_script.bash      (Fundamental)--L6--All---------
```

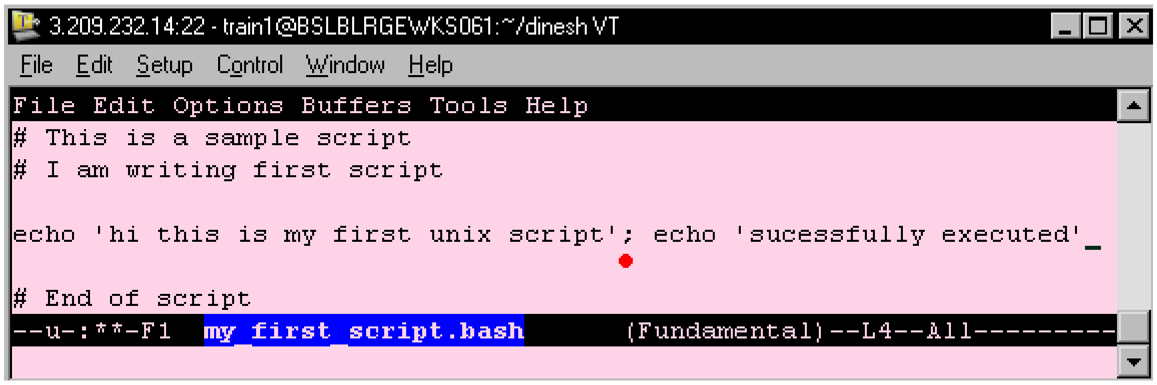

3.209.232.14:22 - train1@BSLBLRGEWKS061:~/dinesh VT
File Edit Setup Control Window Help

```
[train1@BSLBLRGEWKS061 dinesh]$ ./my_first_script.bash
hi this is my first unix script
[train1@BSLBLRGEWKS061 dinesh]$ _
```

Now see the commented lines are not displayed in output.

Command Separator (Semicolon):

We can write two or more commands in single line. To do this we need to use semicolon (;).


3.209.232.14:22 - train1@BSLBLRGEWKS061:~/dinesh VT
File Edit Setup Control Window Help
File Edit Options Buffers Tools Help

```
# This is a sample script
# I am writing first script

echo 'hi this is my first unix script'; echo 'sucessfully executed'_

# End of script
--u-:**-F1  my_first_script.bash      (Fundamental)--L4--All---------
```

```
[train1@BSLBLRGEWKS061 dinesh]$ ./my_first_script.bash
hi this is my first unix script
sucessfully executed
[train1@BSLBLRGEWKS061 dinesh]$
```

Variables in Shell Script:

When we are executing or running any operation is OS with the help of RAM (Random Access Memory). RAM is divided into many locations with unique number.

When we run a shell script some data will be processed and will get stored in RAM in some memory location. Now it will difficult for the user to identify that memory location to reuse the data value. To avoid this situation we assign a unique name to each memory location such that we can access that data value during runtime of the script. This is done by creating a **data variable**.

Types of Variables:

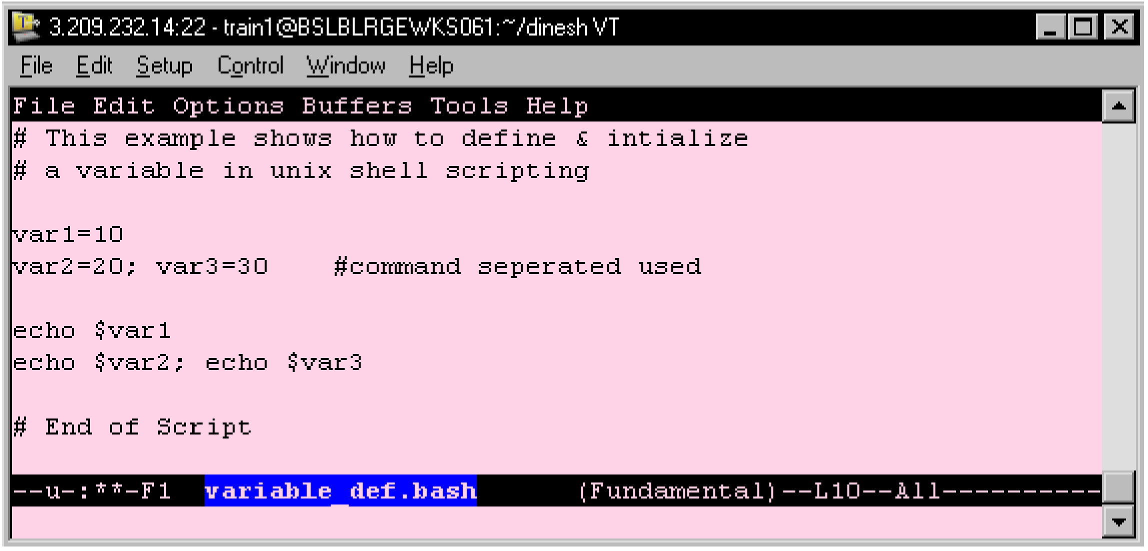
- **System Variable**
- **User Defined Variable**

System Variable: Created by Operating System (OS). Example, SHELL, HOME

User Defined Variable: Created by User. Define a user variable as
Variable_name=**value**

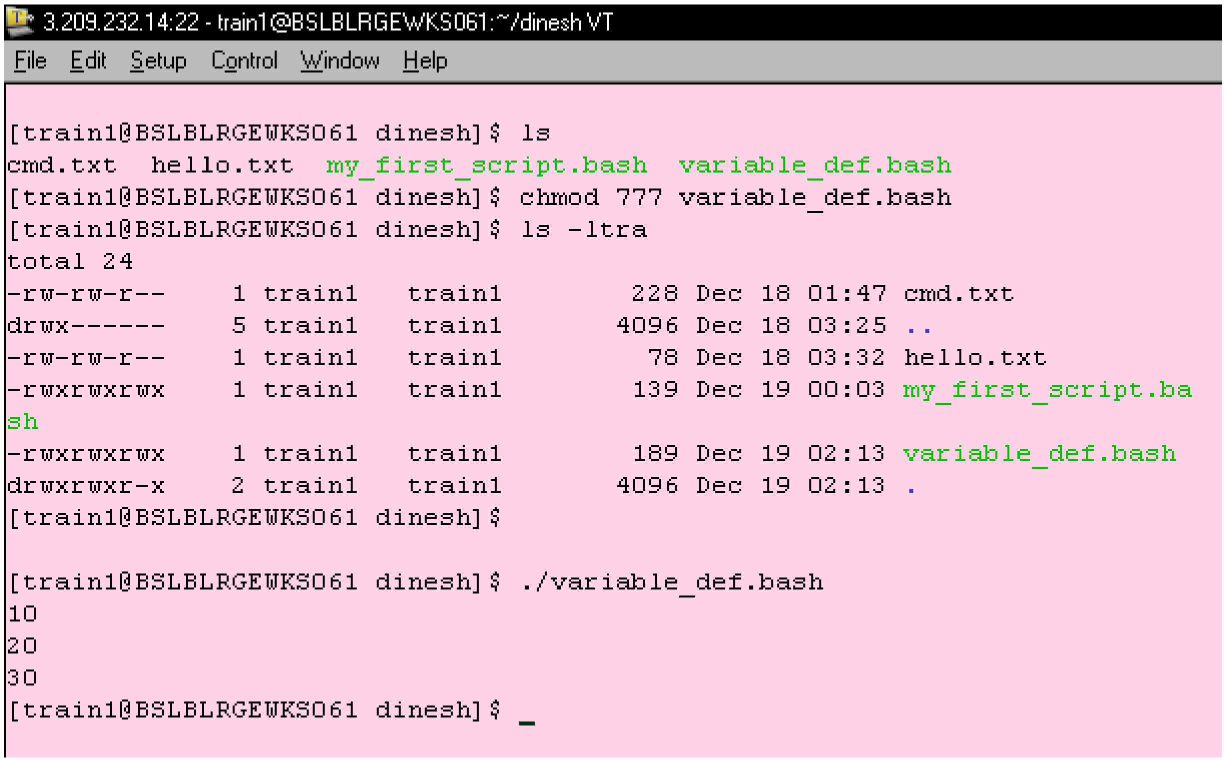
Multiple variables in single line is defined by command separator.
Variable_name=**value**; **Variable_name**=**value**

To print a variable add '**$**' in front of variable.


3.209.232.14:22 - train1@BSLBLRGEWKS061:~/dinesh VT
File Edit Setup Control Window Help
File Edit Options Buffers Tools Help

```
# This example shows how to define & intialize
# a variable in unix shell scripting

var1=10
var2=20; var3=30    #command seperated used

echo $var1
echo $var2; echo $var3

# End of Script
```

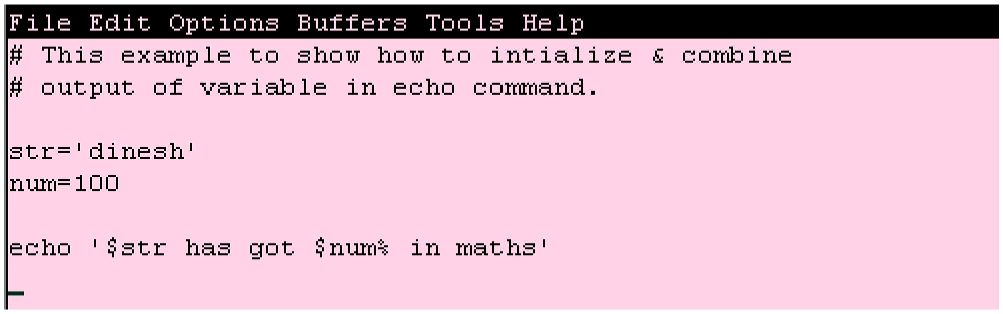
--u-:**-F1 variable_def.bash   (Fundamental)--L10--All----------


3.209.232.14:22 - train1@BSLBLRGEWKS061:~/dinesh VT
File Edit Setup Control Window Help

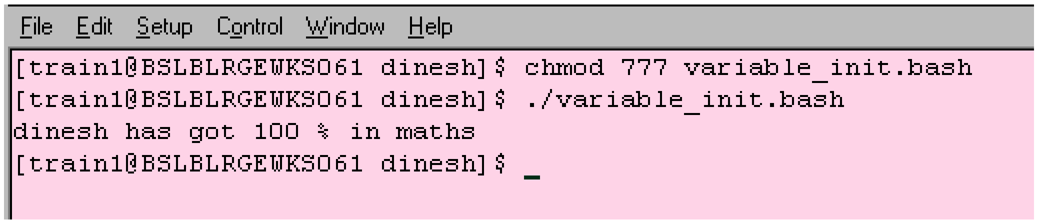
```
[train1@BSLBLRGEWKS061 dinesh]$ ls
cmd.txt  hello.txt  my_first_script.bash  variable_def.bash
[train1@BSLBLRGEWKS061 dinesh]$ chmod 777 variable_def.bash
[train1@BSLBLRGEWKS061 dinesh]$ ls -ltra
total 24
-rw-rw-r--    1 train1   train1        228 Dec 18 01:47 cmd.txt
drwx------    5 train1   train1       4096 Dec 18 03:25 ..
-rw-rw-r--    1 train1   train1         78 Dec 18 03:32 hello.txt
-rwxrwxrwx    1 train1   train1        139 Dec 19 00:03 my_first_script.ba
sh
-rwxrwxrwx    1 train1   train1        189 Dec 19 02:13 variable_def.bash
drwxrwxr-x    2 train1   train1       4096 Dec 19 02:13 .
[train1@BSLBLRGEWKS061 dinesh]$

[train1@BSLBLRGEWKS061 dinesh]$ ./variable_def.bash
10
20
30
[train1@BSLBLRGEWKS061 dinesh]$ _
```

Note: While initializing a variable there should not be any space between variable, operator and value.

We can display multiple variables in single echo command.



```
# This example to show how to intialize & combine
# output of variable in echo command.

str='dinesh'
num=100

echo '$str has got $num% in maths'
```



```
[train1@BSLBLRGEWKS061 dinesh]$ chmod 777 variable_init.bash
[train1@BSLBLRGEWKS061 dinesh]$ ./variable_init.bash
dinesh has got 100 % in maths
[train1@BSLBLRGEWKS061 dinesh]$ _
```

"echo" Command:

This command is used to display variable values or texts.
Syntax:

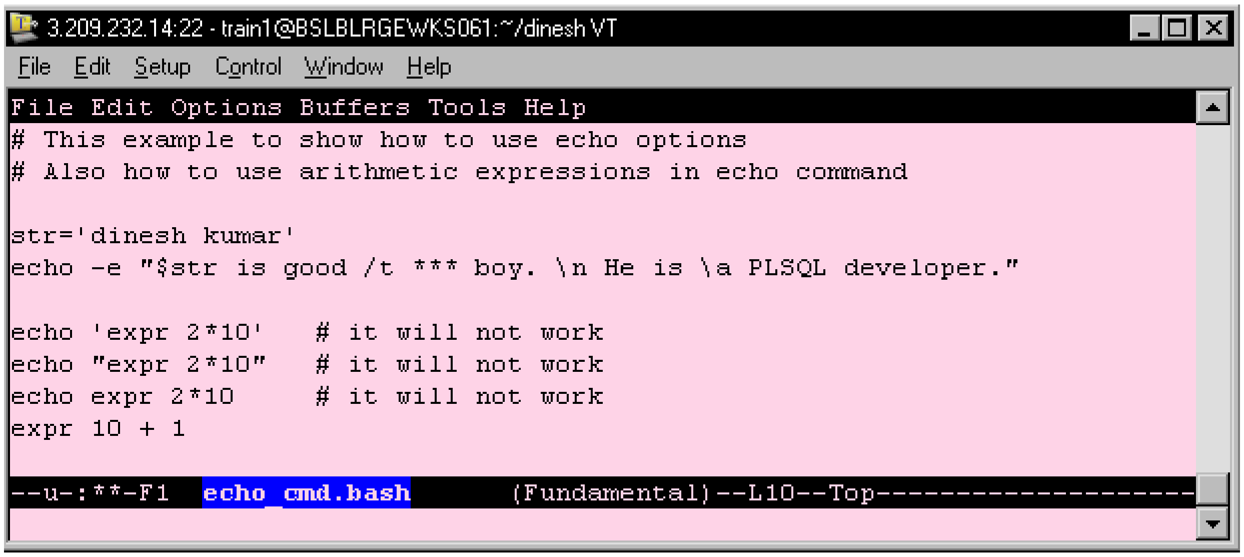
```
Echo                    #Displays empty line

Echo <option> $variable_name

Echo <option> "texts"

Echo <option> "$var1……. Texts ……… $var2"
```

Example: echo -e "texts \t $var1 \n $var2

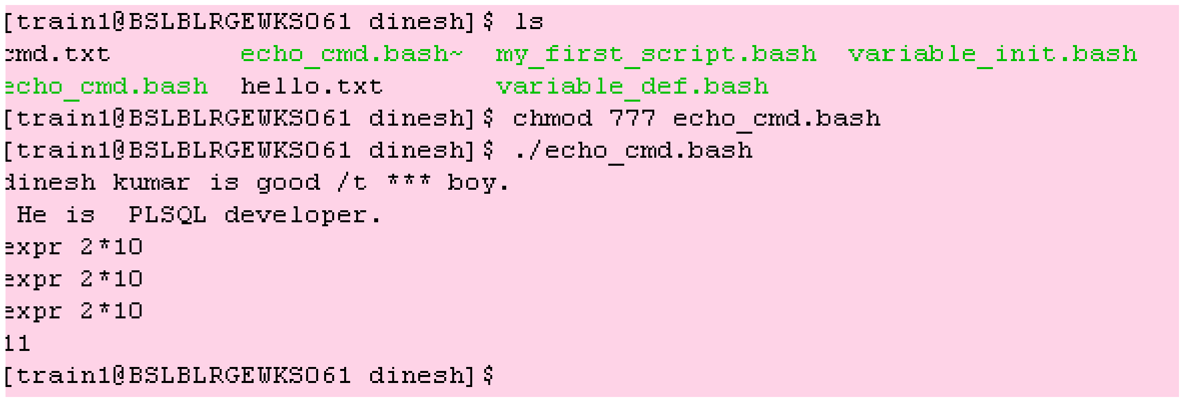
Options:
\a → alert
\n → new line
\t → tab
\r → carriage return
\\ → back slash
-n → do not output trail new line
-e → enables the above option to use in echo command. (Mandatory)

Shell Arithmetic's:

To perform a arithmetic operation we need to use key word "expr".


3.209.232.14:22 - train1@BSLBLRGEWKS061:~/dinesh VT
File Edit Setup Control Window Help
File Edit Options Buffers Tools Help

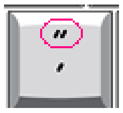
```
# This example to show how to use echo options
# Also how to use arithmetic expressions in echo command

str='dinesh kumar'
echo -e "$str is good /t *** boy. \n He is \a PLSQL developer."

echo 'expr 2*10'    # it will not work
echo "expr 2*10"    # it will not work
echo expr 2*10      # it will not work
expr 10 + 1

--u-:**-F1  echo_cmd.bash       (Fundamental)--L10--Top--------------------
```



```
[train1@BSLBLRGEWKS061 dinesh]$ ls
cmd.txt         echo_cmd.bash~  my_first_script.bash  variable_init.bash
echo_cmd.bash   hello.txt       variable_def.bash
[train1@BSLBLRGEWKS061 dinesh]$ chmod 777 echo_cmd.bash
[train1@BSLBLRGEWKS061 dinesh]$ ./echo_cmd.bash
dinesh kumar is good /t *** boy.
 He is  PLSQL developer.
expr 2*10
expr 2*10
expr 2*10
11
[train1@BSLBLRGEWKS061 dinesh]$
```

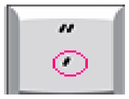
Quotes:

Types of quotes:



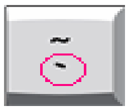
" " → Double quote

Anything inside double quotes removes meaning of string except \ and $.



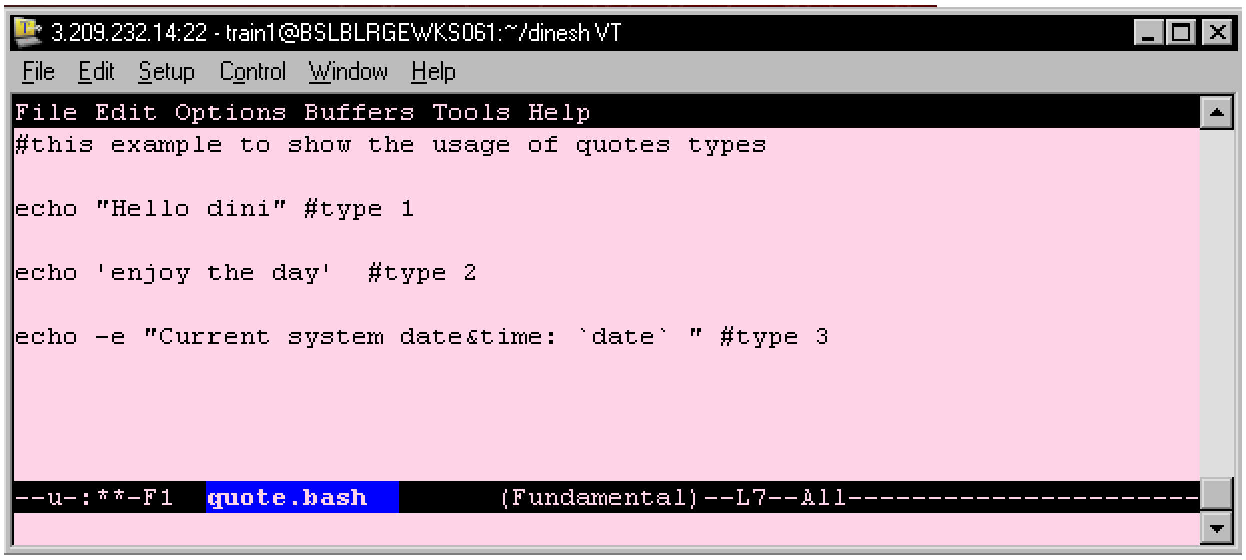
' ' → Single quote

Anything inside single quotes remains unchanged.



` ` → Back quote

Anything inside back quote executes command.



3.209.232.14:22 - train1@BSLBLRGEWKS061:~/dinesh VT
File Edit Setup Control Window Help
File Edit Options Buffers Tools Help

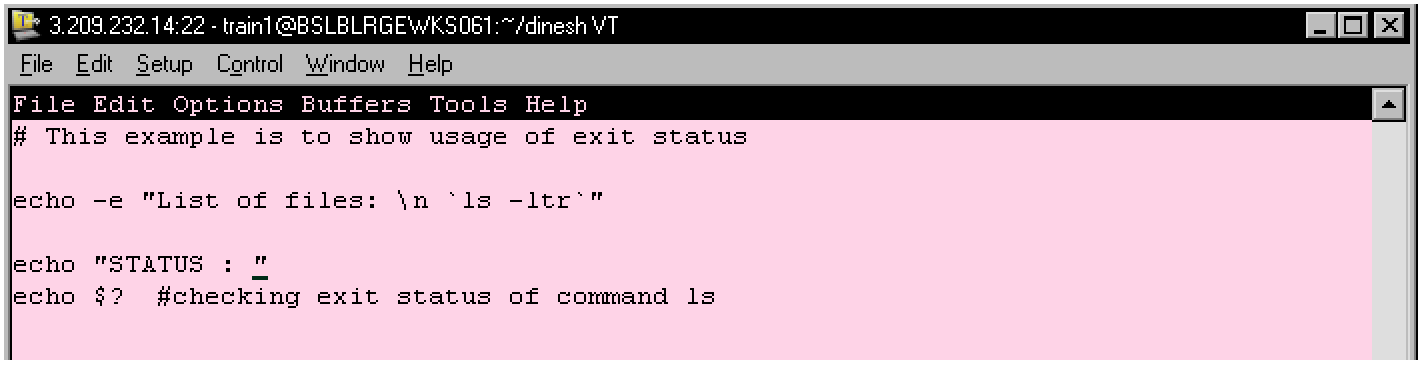
```
#this example to show the usage of quotes types

echo "Hello dini" #type 1

echo 'enjoy the day'  #type 2

echo -e "Current system date&time: `date` " #type 3
```

--u-:**-F1 quote.bash   (Fundamental)--L7--All----------------------

```
[train1@BSLBLRGEWKS061 dinesh]$ ls
cmd.txt          hello.txt             quote.bash~
echo_cmd.bash    my_first_script.bash  variable_def.bash
echo_cmd.bash~   quote.bash            variable_init.bash
[train1@BSLBLRGEWKS061 dinesh]$ chmod 777 quote.bash
[train1@BSLBLRGEWKS061 dinesh]$ ./quote.bash
Hello dini
enjoy the day
Current system date&time: Fri Dec 19 04:00:53 IST 2008
[train1@BSLBLRGEWKS061 dinesh]$
```

Exit Status:
As we know we can embed a shell command inside shell script. If we want to know about the status of the executed command i.e. whether it is '**success**' or '**failure**' we are going to use exit status.

Syntax: **exit $?**
Zero (0) → success
Others → Error


3.209.232.14:22 - train1@BSLBLRGEWKS061:~/dinesh VT
File Edit Setup Control Window Help
File Edit Options Buffers Tools Help

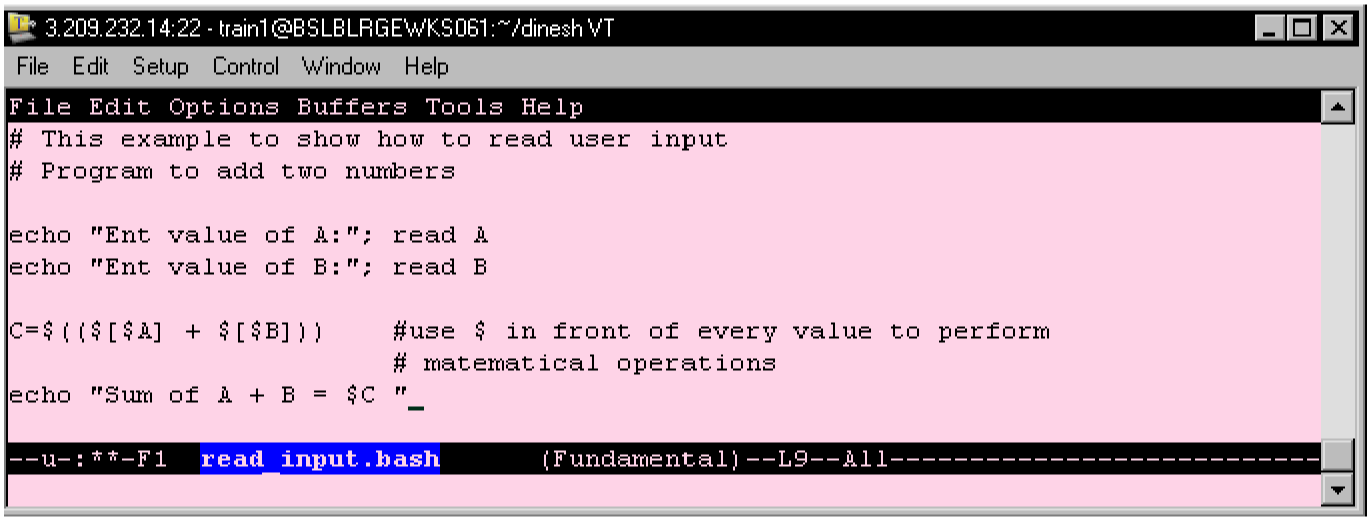
```
# This example is to show usage of exit status

echo -e "List of files: \n `ls -ltr`"

echo "STATUS : "
echo $?  #checking exit status of command ls
```

```
[train1@BSLBLRGEWKS061 dinesh]$ chmod 777 exit_status.bash
[train1@BSLBLRGEWKS061 dinesh]$ ./exit_status.bash
List of files:
 total 44
-rw-rw-r--    1 train1   train1        228 Dec 18 01:47 cmd.txt
-rw-rw-r--    1 train1   train1         78 Dec 18 03:32 hello.txt
-rwxrwxrwx    1 train1   train1        139 Dec 19 00:03 my_first_script.bash
-rwxrwxrwx    1 train1   train1        189 Dec 19 02:13 variable_def.bash
-rwxrwxrwx    1 train1   train1        147 Dec 19 02:48 variable_init.bash
-rwxrwxrwx    1 train1   train1        330 Dec 19 03:33 echo_cmd.bash~
-rwxrwxrwx    1 train1   train1        317 Dec 19 03:34 echo_cmd.bash
-rwxrwxrwx    1 train1   train1        182 Dec 19 03:59 quote.bash~
-rwxrwxrwx    1 train1   train1        169 Dec 19 04:00 quote.bash
-rwxrwxrwx    1 train1   train1        132 Dec 19 04:15 exit_status.bash~
-rwxrwxrwx    1 train1   train1        149 Dec 19 04:17 exit_status.bash
STATUS :
0
[train1@BSLBLRGEWKS061 dinesh]$
```

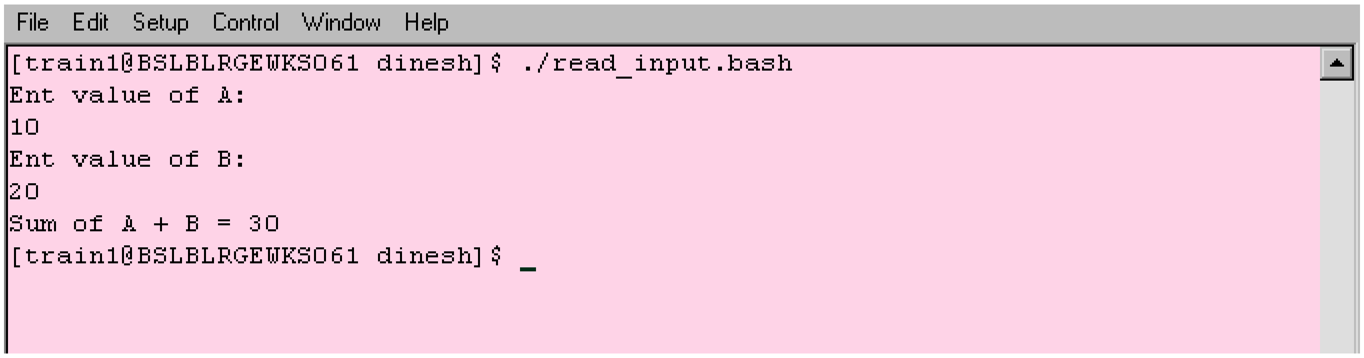
Read variables from user:

To read an input from keyboard the following syntax is used.

read variable_name


3.209.232.14:22 - train1@BSLBLRGEWKS061:~/dinesh VT
File Edit Setup Control Window Help
File Edit Options Buffers Tools Help

```
# This example to show how to read user input
# Program to add two numbers

echo "Ent value of A:"; read A
echo "Ent value of B:"; read B

C=$(($[$A] + $[$B]))     #use $ in front of every value to perform
                         # matematical operations
echo "Sum of A + B = $C "_

--u-:**-F1  read_input.bash       (Fundamental)--L9--All---------------------------
```



```
[train1@BSLBLRGEWKS061 dinesh]$ ./read_input.bash
Ent value of A:
10
Ent value of B:
20
Sum of A + B = 30
[train1@BSLBLRGEWKS061 dinesh]$ _
```

Redirecting input/output:

Types: There are 3 redirection types as follows:

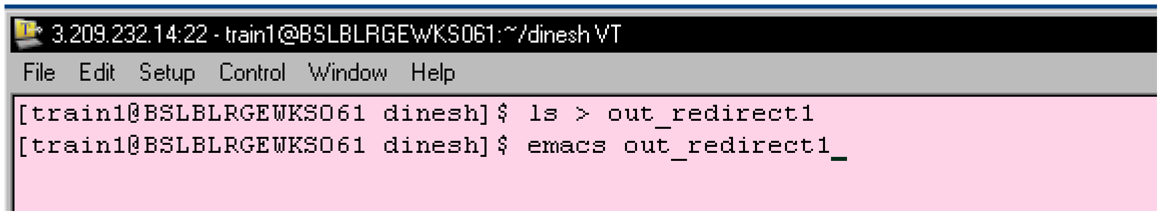
1) >
2) >>
3) <

>: This will redirect the output to a file.
Example: ls > out_redirect1

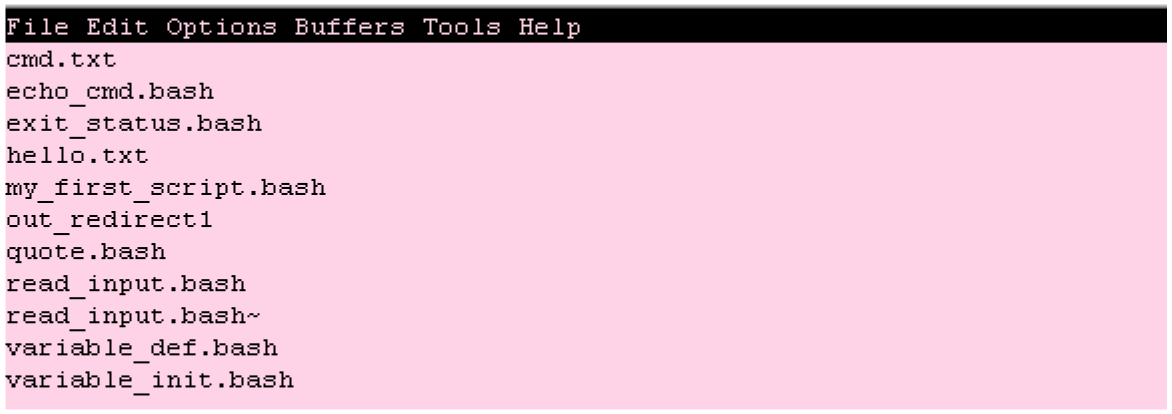
>>: This will append the output to existing file at last. If data exists it will be left if not it will be added. Example: ls >> out_redirect1

<: This will take input from file to a command.
Example: cat < out_redirect1

Example 2: cat < out_redirect1 >> out_redirect1 > out_redirect2
This will take input from out_redirect1 & append the result again into out_redirect1, then sends the complete output to out_redirect2.

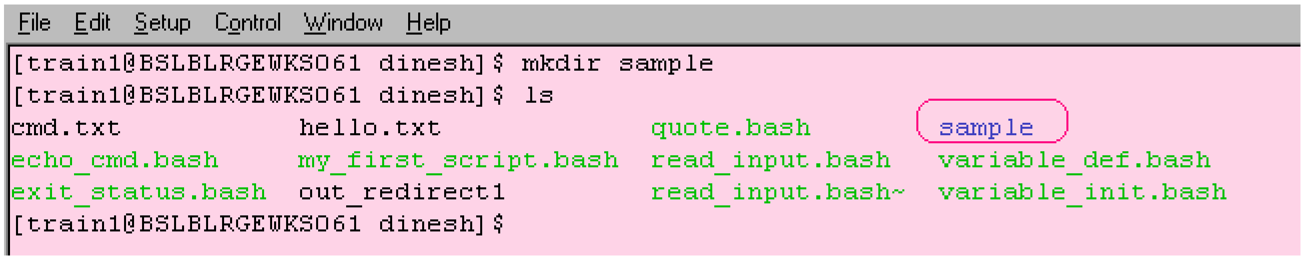

3.209.232.14:22 - train1@BSLBLRGEWKS061:~/dinesh VT
File Edit Setup Control Window Help

```
[train1@BSLBLRGEWKS061 dinesh]$ ls > out_redirect1
[train1@BSLBLRGEWKS061 dinesh]$ emacs out_redirect1_
```

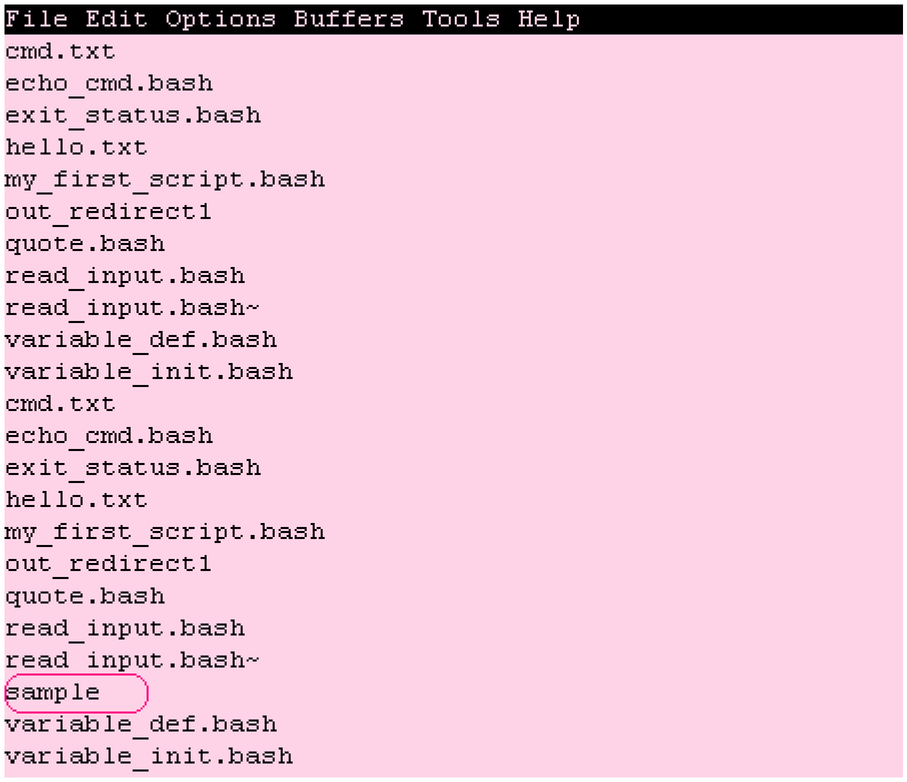


```
cmd.txt
echo_cmd.bash
exit_status.bash
hello.txt
my_first_script.bash
out_redirect1
quote.bash
read_input.bash
read_input.bash~
variable_def.bash
variable_init.bash
```

ls >> out_redirect1



```
[train1@BSLBLRGEWKS061 dinesh]$ mkdir sample
[train1@BSLBLRGEWKS061 dinesh]$ ls
cmd.txt            hello.txt             quote.bash         sample
echo_cmd.bash      my_first_script.bash  read_input.bash    variable_def.bash
exit_status.bash   out_redirect1         read_input.bash~   variable_init.bash
[train1@BSLBLRGEWKS061 dinesh]$
```

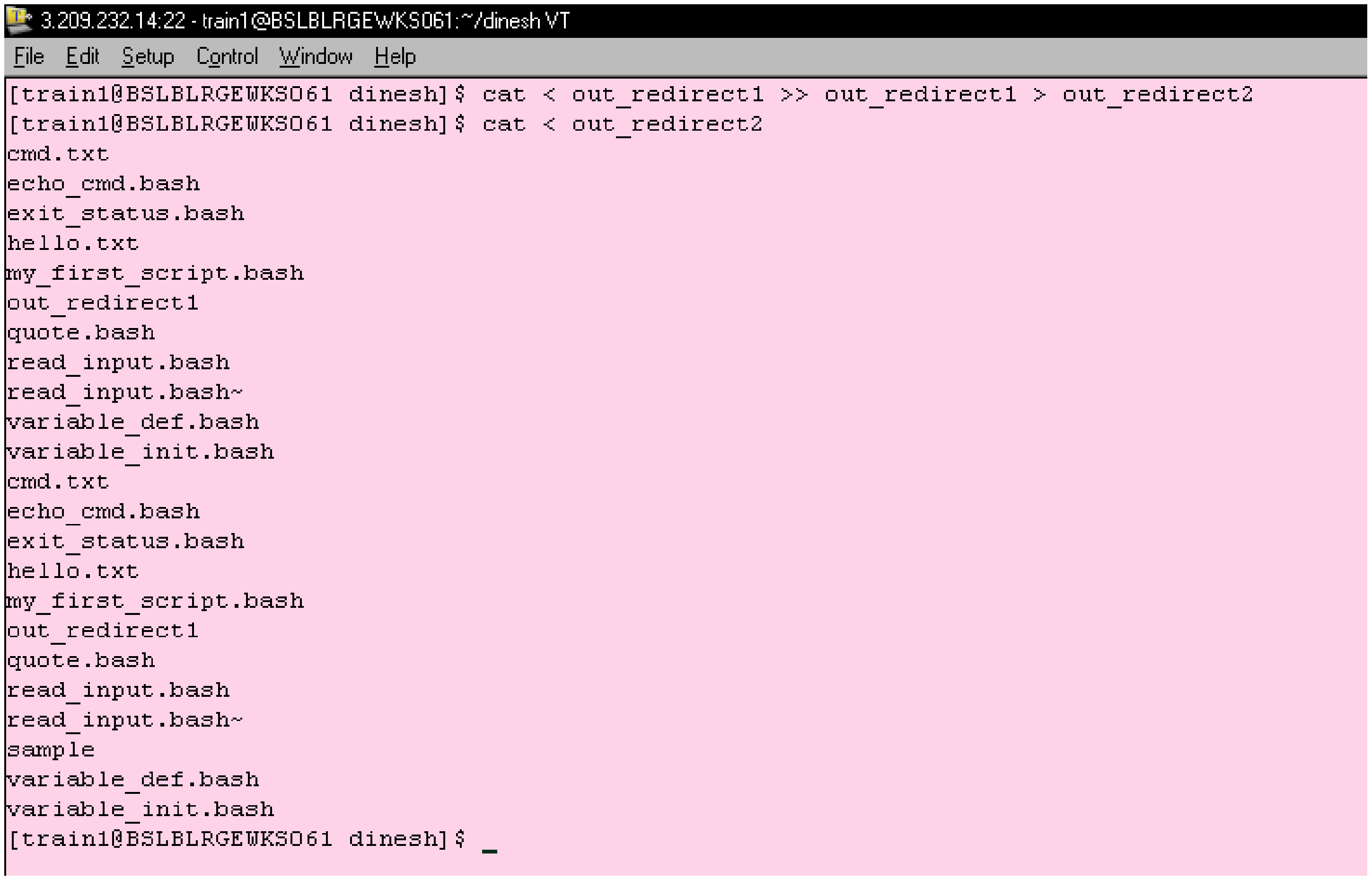


```
cmd.txt
echo_cmd.bash
exit_status.bash
hello.txt
my_first_script.bash
out_redirect1
quote.bash
read_input.bash
read_input.bash~
variable_def.bash
variable_init.bash
cmd.txt
echo_cmd.bash
exit_status.bash
hello.txt
my_first_script.bash
out_redirect1
quote.bash
read_input.bash
read_input.bash~
sample
variable_def.bash
variable_init.bash
```

cat < out_redirect1

We will be getting the same output as above.

cat < out_redirect1 >> out_redirect1 > out_redirect2


3.209.232.14:22 - train1@BSLBLRGEWKS061:~/dinesh VT
File Edit Setup Control Window Help

```
[train1@BSLBLRGEWKS061 dinesh]$ cat < out_redirect1 >> out_redirect1 > out_redirect2
[train1@BSLBLRGEWKS061 dinesh]$ cat < out_redirect2
cmd.txt
echo_cmd.bash
exit_status.bash
hello.txt
my_first_script.bash
out_redirect1
quote.bash
read_input.bash
read_input.bash~
variable_def.bash
variable_init.bash
cmd.txt
echo_cmd.bash
exit_status.bash
hello.txt
my_first_script.bash
out_redirect1
quote.bash
read_input.bash
read_input.bash~
sample
variable_def.bash
variable_init.bash
[train1@BSLBLRGEWKS061 dinesh]$ _
```

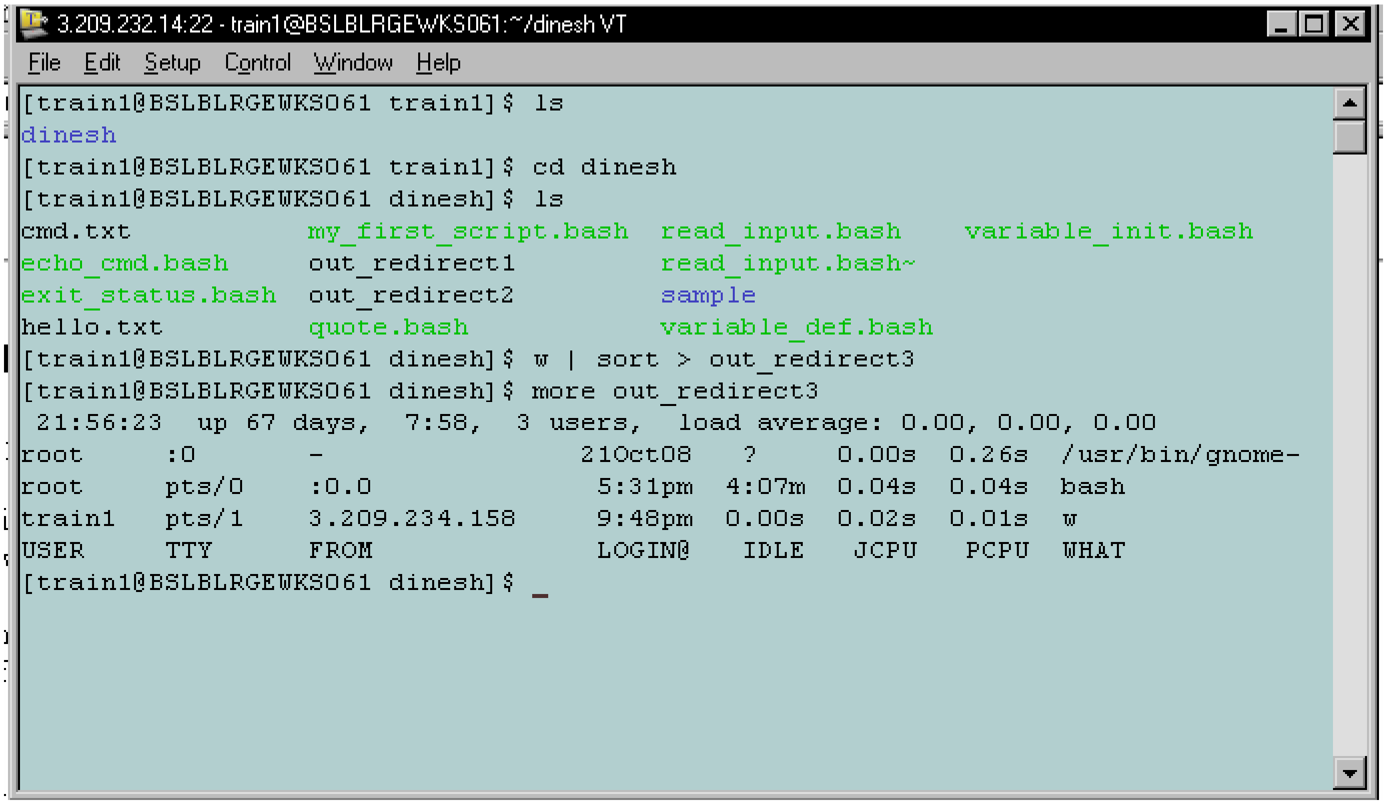
Pipe: (|)

Pipe is used to link output of one program as input to another program.

Example: **ls | cat**
As above the output of ls is given as input to cat command.

Example 2: **w | sort > out_redirect3**
The output of w i.e. users will be sorted & output will be redirected to the file.

Output:


3.209.232.14:22 - train1@BSLBLRGEWKS061:~/dinesh VT
File Edit Setup Control Window Help

```
[train1@BSLBLRGEWKS061 train1]$ ls
dinesh
[train1@BSLBLRGEWKS061 train1]$ cd dinesh
[train1@BSLBLRGEWKS061 dinesh]$ ls
cmd.txt            my_first_script.bash  read_input.bash    variable_init.bash
echo_cmd.bash      out_redirect1         read_input.bash~
exit_status.bash   out_redirect2         sample
hello.txt          quote.bash            variable_def.bash
[train1@BSLBLRGEWKS061 dinesh]$ w | sort > out_redirect3
[train1@BSLBLRGEWKS061 dinesh]$ more out_redirect3
 21:56:23  up 67 days,  7:58,  3 users,  load average: 0.00, 0.00, 0.00
root     :0       -                21Oct08   ?     0.00s  0.26s  /usr/bin/gnome-
root     pts/0    :0.0              5:31pm  4:07m  0.04s  0.04s  bash
train1   pts/1    3.209.234.158     9:48pm  0.00s  0.02s  0.01s  w
USER     TTY      FROM              LOGIN@   IDLE   JCPU   PCPU  WHAT
[train1@BSLBLRGEWKS061 dinesh]$ _
```

Chapter 6

Operators

Test Operators:

Test Operators	True then
-e	File exists
-f	File is normal file not directory or system or device files.
-b	File is blocked device
-c	File is character device
-p	File is pipe
-s	File is symbolic link
-r	File has read permission
-w	File has write permission
-x	File has execute permission
-g	Group id flag set to file
-u	User id flag set to file
-O	User is owner of file
-nt	Newer than (f1 –nt f2)
-ot	Older than (f1 –ot f2)

Comparison Operators:

Numbers:

Operator	Description	Example
-eq	Equal	**if ["$var1" –eq "$var2"]**
-ne	Not equal	**if ["$var1" –ne "$var2"]**
-gt	Greater than	**if ["$var1" –gt "$var2"]**
-ge	Greater than or equal to	**if ["$var1" –ge "$var2"]**
-lt	Less than	**if ["$var1" –lt "$var2"]**
-le	Lesser than or equal to	**if ["$var1" –lt"$var2"]**
<	Less than	if (("$var1" < "$var2"))
<=	Less than or equal to	if (("$var1" <= "$var2"))
>	Greater than	if (("$var1" > "$var2"))
>=	Greater than or equal to	if (("$var1" >= "$var2"))

Note: While using operator use double parenthesis.

Strings:

Operator	Description	Example
=	equal to	if ["$var1" = "$var2"]
==	For comparison	if ["$var1" == "$var2"]
!=	Not equal to	if ["$var1" != "$var2"]

<	Less than	if [["$var1" < "$var2"]] (or)
		if ["$var1" \< "$var2"]

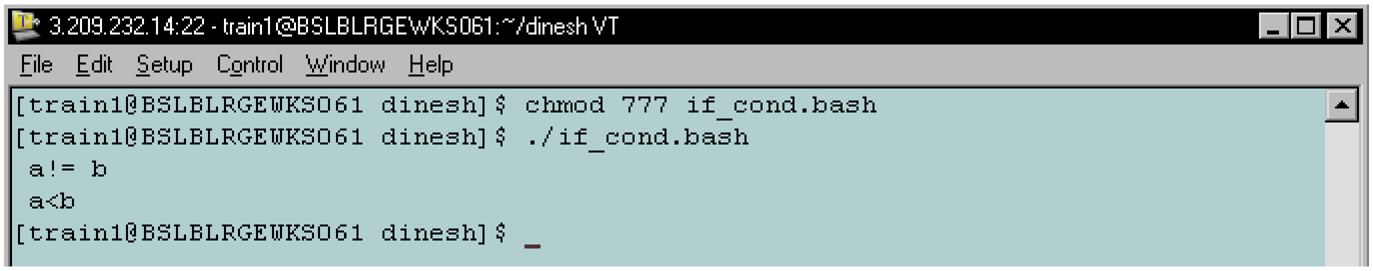
>	Greater than	if [["$var1" > "$var2"]] (or)
		if ["$var1" \> "$var2"]

-z	String is null	if [-z "$str"]
-n	String not null	if [-n "$str"]

Note: you can use "( )" parenthesis or "[ ]" Square brackets to enclose variables in conditions.

Example 1: Using operators with integers.

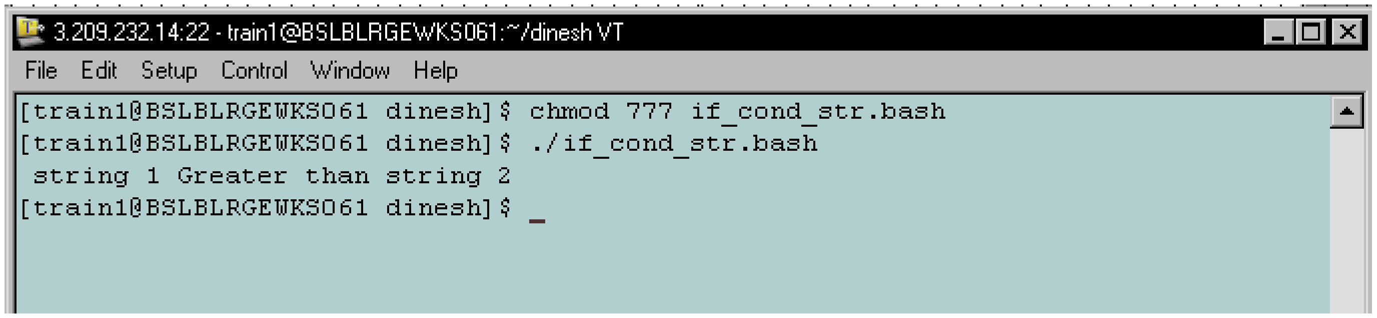
```
# This exercise to show the usage of conditional statements.

a=10
b=20

if [ "$a" -eq "$b" ]     #use [] or () brackets for using this condition
then
        echo " a = b"

elif [ "$a" -ne "$b" ]
then
        echo " a!= b"
fi

if (( "$a" < "$b" ))   # use [[]] or (()) brackets while using this operator
then
        echo " a<b"

elif (( "$a" > "$b" ))
then
        echo "a > b"
fi
```


3.209.232.14:22 - train1@BSLBLRGEWKS061:~/dinesh VT
File Edit Setup Control Window Help

```
[train1@BSLBLRGEWKS061 dinesh]$ chmod 777 if_cond.bash
[train1@BSLBLRGEWKS061 dinesh]$ ./if_cond.bash
 a!= b
 a<b
[train1@BSLBLRGEWKS061 dinesh]$ _
```

Example 2: Using operators with strings.

```
# This program to show how to use operator with strings.

str1=dinesh
str2=dineshkumars
str=

if [ "$str1" == "$str2" ]
then
        echo " Both Strings are equal"

elif [[ "$str1" < "$str2" ]]  #use [[]] or (()) while using operator
then
        echo " string 1 Greater than string 2"

elif [ "$str1" \< "$str2" ]    #use [] or () if '/' is added with operator
then
        echo -e " \n String 2 greater than string 1"

else
        echo " operation aborted"
fi
```



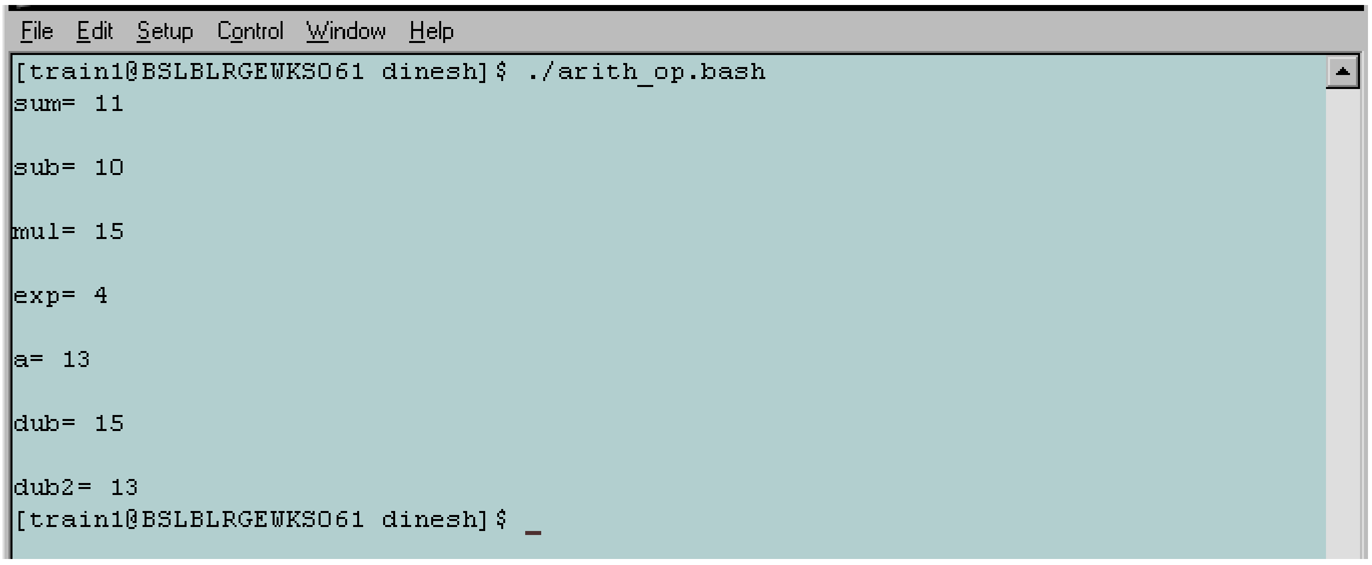
```
[train1@BSLBLRGEWKS061 dinesh]$ chmod 777 if_cond_str.bash
[train1@BSLBLRGEWKS061 dinesh]$ ./if_cond_str.bash
 string 1 Greater than string 2
[train1@BSLBLRGEWKS061 dinesh]$ _
```

Arithmetic Operators:

```
+ --> Plus (used for addition or increment operation)

- --> Minus (used for subtraction or decrement operation)

* --> Multiplication ( used to multiply numbers)

/ --> division ( used to divide numbers )

** --> Exponentional ( used for power ooperations)
```

```
%  --> Modulo ( Returns remainder)
+= --> Plus equal (used to add a variable with another variable or constant)
-= --> minus equal (used to subtract a variable with another variable or constant)
*= --> Multiply equal (used to multiply a variable with another variable or constant)
/= --> Division equal (used to multiply a variable with another variable or constant)
%= --> modulo equal (used to divide remainder of  a variable with another variable or constant)
bc --> Use bc to add or subtract or divide any floating point variables.
```

```
# This example to show how to use arithmetic operators

a=10

let "sum=5+6"

let "sub=30-20"

let "mul=5*3"

let "exp=2**2"

let "a+=5"

let "dub=a--"
let "dub2=--a"

echo -e "sum= $sum\n"
echo -e "sub= $sub\n"
echo -e "mul= $mul\n"
echo -e "exp= $exp\n"
echo -e "a= $a \n"
echo -e "dub= $dub\n"
echo -e "dub2= $dub2"
```

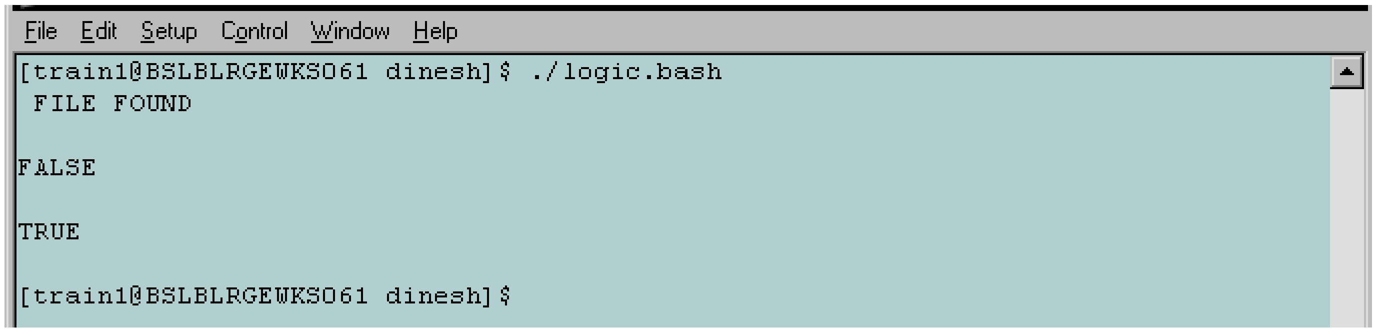


```
[train1@BSLBLRGEWKS061 dinesh]$ ./arith_op.bash
sum= 11

sub= 10

mul= 15

exp= 4

a= 13

dub= 15

dub2= 13
[train1@BSLBLRGEWKS061 dinesh]$ _
```

Logical boolean Operator:

```
 !  --> NOT (IF  condition is true the result is false)
&& --> AND (Both condition should be true for getting result)
 || --> OR (If any one condition is true the result is true)
```

```
# this example to show how to use logical boolean operators

a=10
b=20
file=quote.bash

if [ ! -f "$file" ]
then
    echo -e " File Not Found\n"

else
    echo -e " FILE FOUND \n"

fi

if [[ "$a" = 10 && "$b" = 0 ]]
then
    echo -e "TRUE \n"  # Since only one condition is satisfied the result is false

else
    echo -e "FALSE \n"

fi

if [[ "$a" = 10 || "$b" = 0 ]]
then
     echo -e "TRUE \n" #Since on condition is satisfied result is true"

else
    echo -e "FALSE \n"

fi
```

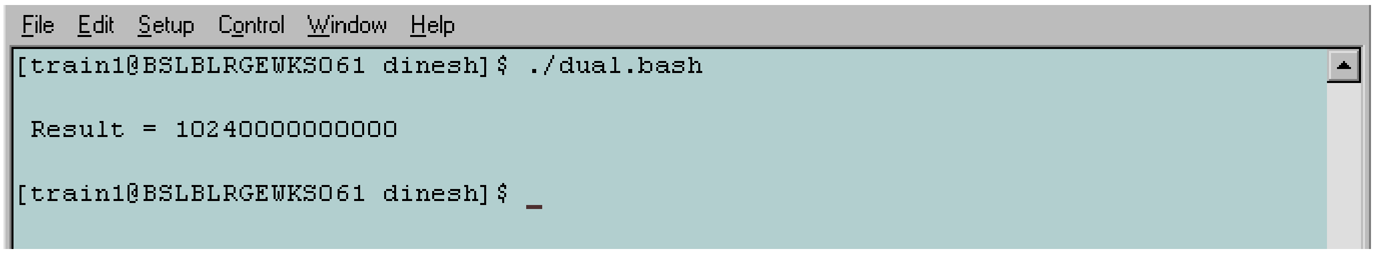


```
[train1@BSLBLRGEWKS061 dinesh]$ ./logic.bash
 FILE FOUND

FALSE

TRUE

[train1@BSLBLRGEWKS061 dinesh]$
```

Comma Operator:

It combines two or more arithmetic operations. But only last arithmetic operation is value is returned.

```
#This example to show how to use comma operator

a=10
let "Result=((a++, --a, a+=10, a**10))"

echo -e  "\n Result = $Result \n"
```



```
[train1@BSLBLRGEWKS061 dinesh]$ ./dual.bash

 Result = 10240000000000

[train1@BSLBLRGEWKS061 dinesh]$ _
```

Chapter 7

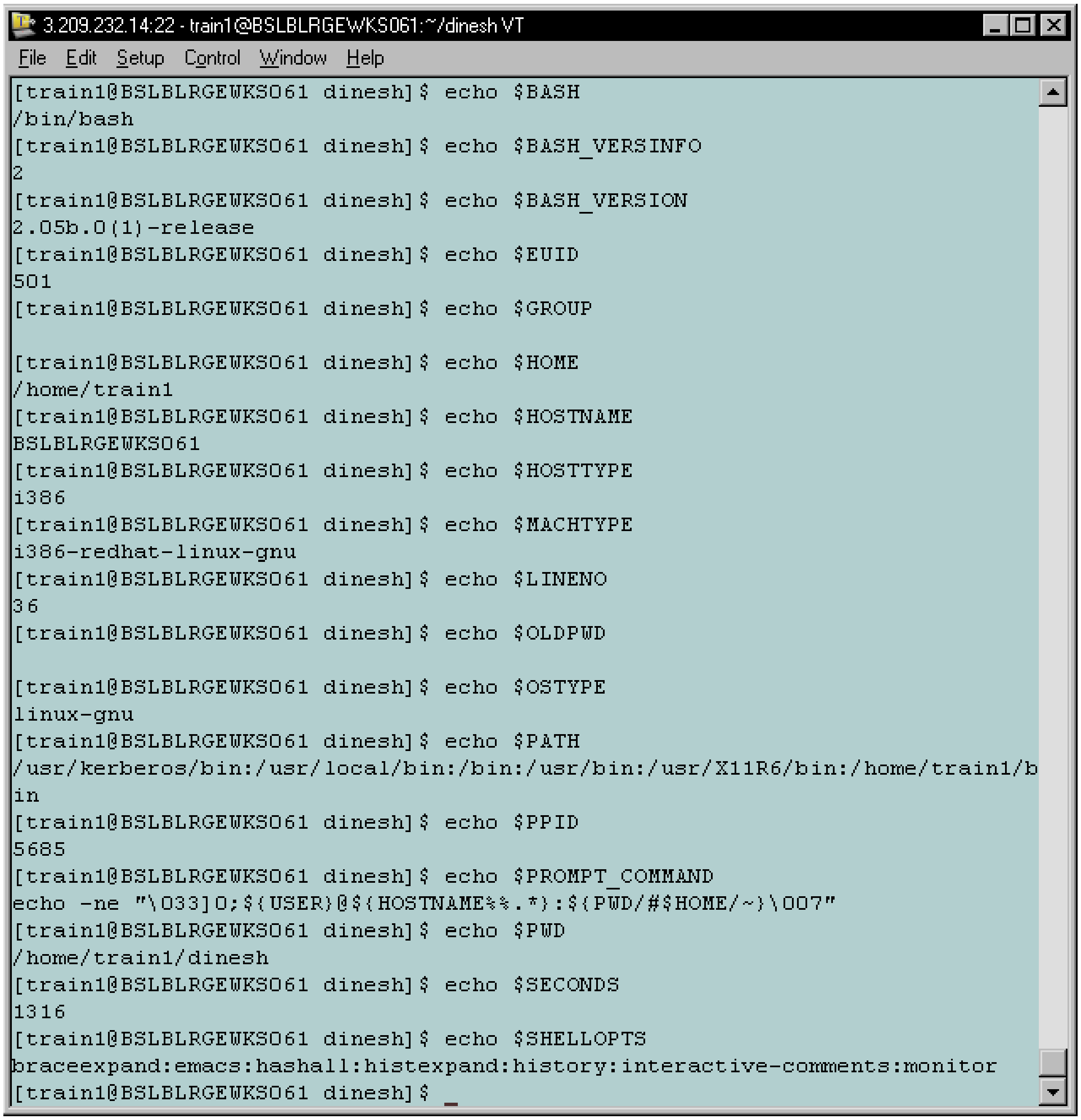
Variables Manipulation (Advance)

Builtin Variables:

These are variables which affect bash script behavior.

Variables	Description
$BASH	Displays bash path.
$BASH_ENV	Points to bash environment variables.
$BASH_VERSINFO	Displays bash shell version.
$BASH_VERSION	Displays bash version.
$EDITOR	Displays editor used by script.
$EUID	Displays user id.
$GROUP	Displays user group id.
$HOME	Displays home directory of user.
$HOSTNAME	Displays host name.
$HOSTTYPE	Displays host type.
$MACHTYPE	Displays hardware type.
$IGNOREOF	Ignore EOF.
$LINENO	Displays line number in script.
$OLDPWD	Displays old directory which user worked.
$OSTYPE	Displays Operating System type.
$PATH	Displays all path of user.
$PPID	Displays process ID.
$PROMPT_COMMAND	Displays variable holding command to execute.
$PWD	Displays present working directory.
$REPLY	Default value when value is not given to read variable.
$SECONDS	Displays no. of second's script being executed.
$SHELLOPTS	Displays list of shell options.

Note:
I have not discussed all the internal variables but most commonly used.
All the built-in variable command should be in UPPER CASE.


3.209.232.14:22 - train1@BSLBLRGEWKS061:~/dinesh VT
File Edit Setup Control Window Help

```
[train1@BSLBLRGEWKS061 dinesh]$ echo $BASH
/bin/bash
[train1@BSLBLRGEWKS061 dinesh]$ echo $BASH_VERSINFO
2
[train1@BSLBLRGEWKS061 dinesh]$ echo $BASH_VERSION
2.05b.0(1)-release
[train1@BSLBLRGEWKS061 dinesh]$ echo $EUID
501
[train1@BSLBLRGEWKS061 dinesh]$ echo $GROUP

[train1@BSLBLRGEWKS061 dinesh]$ echo $HOME
/home/train1
[train1@BSLBLRGEWKS061 dinesh]$ echo $HOSTNAME
BSLBLRGEWKS061
[train1@BSLBLRGEWKS061 dinesh]$ echo $HOSTTYPE
i386
[train1@BSLBLRGEWKS061 dinesh]$ echo $MACHTYPE
i386-redhat-linux-gnu
[train1@BSLBLRGEWKS061 dinesh]$ echo $LINENO
36
[train1@BSLBLRGEWKS061 dinesh]$ echo $OLDPWD

[train1@BSLBLRGEWKS061 dinesh]$ echo $OSTYPE
linux-gnu
[train1@BSLBLRGEWKS061 dinesh]$ echo $PATH
/usr/kerberos/bin:/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:/home/train1/b
in
[train1@BSLBLRGEWKS061 dinesh]$ echo $PPID
5685
[train1@BSLBLRGEWKS061 dinesh]$ echo $PROMPT_COMMAND
echo -ne "\033]0;${USER}@${HOSTNAME%%.*}:${PWD/#$HOME/~}\007"
[train1@BSLBLRGEWKS061 dinesh]$ echo $PWD
/home/train1/dinesh
[train1@BSLBLRGEWKS061 dinesh]$ echo $SECONDS
1316
[train1@BSLBLRGEWKS061 dinesh]$ echo $SHELLOPTS
braceexpand:emacs:hashall:histexpand:history:interactive-comments:monitor
[train1@BSLBLRGEWKS061 dinesh]$ _
```

Command Line Arguments:

Command Line arguments are nothing but passing a "**Parameter**" for a script to run. Such as,

Plsql:

SQL> Select f1 (**10**, **20**) from dual;

Here 10 & 20 are passed as argument to function f1 to be executed successfully.

In the same way we pass arguments to Shell Script at runtime to have user interaction with command line.

Syntax:

./Shell_Script_name.bash arg1 arg2 arg3 ….. arg9

Example:

. /**Dinesh** 10 2

Dinesh – Script name
10 – Parameter 1
2 - Parameter 2

Note: **We can pass maximum of 9 command line arguments only**.

Positional Parameters:

Positional Parameters are passed from command line to script or to a variable.

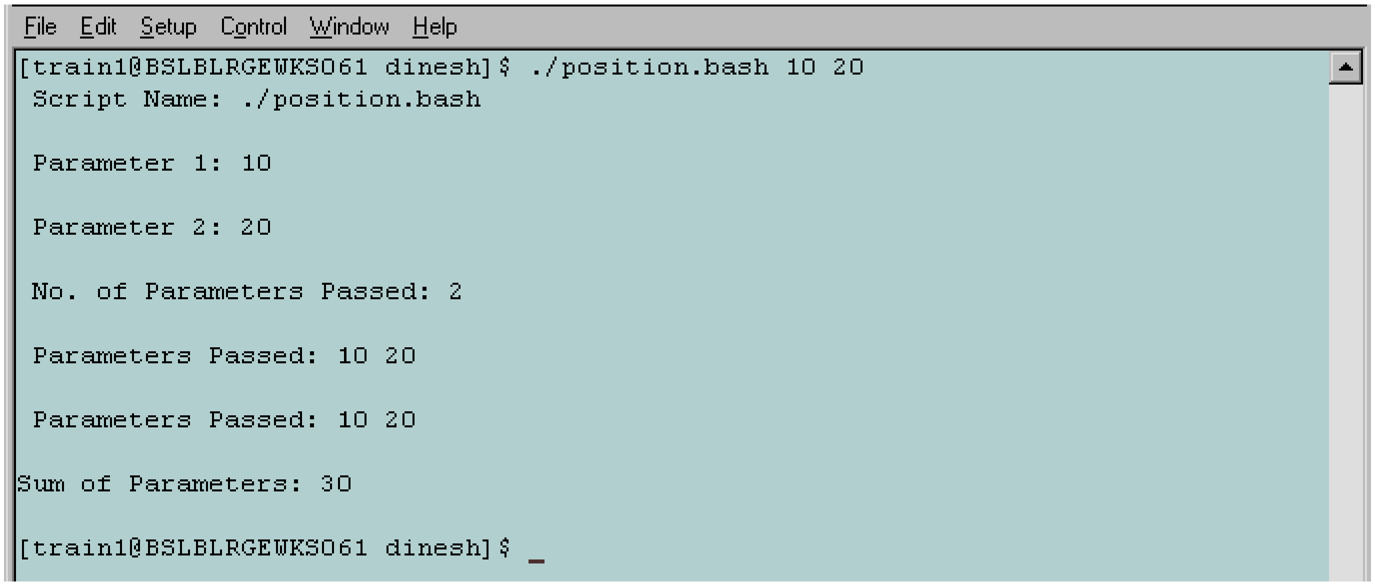
In a simple manner when arguments are passed from command line to script it is called 'command line arguments'. The same variables, when used inside the script are called as 'positional parameters'.

Positional Parameters	Description
$0	Script Name
$1 to $9	Arguments passed to the script
$#	No. of command line arguments
$*	Displays all parameters in single line
$@	Same as $* but considers each parameter as a single word

Consider the script below.

```
# This example to show the usage of positional parameters

echo -e " Script Name: $0 \n"
echo -e " Parameter 1: $1 \n"
echo -e " Parameter 2: $2 \n"
echo -e " No. of Parameters Passed: $# \n"
echo -e " Parameters Passed: $* \n"
echo -e " Parameters Passed: $@ \n"


sum=$[$[$1]+$[$2]]

echo -e "Sum of Parameters: $sum \n"
```

Output:



```
[train1@BSLBLRGEWKS061 dinesh]$ ./position.bash 10 20
 Script Name: ./position.bash

 Parameter 1: 10

 Parameter 2: 20

 No. of Parameters Passed: 2

 Parameters Passed: 10 20

 Parameters Passed: 10 20

Sum of Parameters: 30

[train1@BSLBLRGEWKS061 dinesh]$ _
```

Chapter 8 <u>Conditional Statements</u>

Conditional Statements:

if condition
if else if condition
Nested **if**
if elif Condition

I. if condition:

If the condition is satisfied then statements inside body is executed.

Syntax:

```
if $var1 Comparision_operator $var2
then
          Statements…..
fi

(or)

if $var1 test_operator $var2
then
     Statements…..
fi
```

II. if else if Condition:

If 1st condition is not satisfied then statements in else body will be executed.

Syntax:

```
if $var1 Comparision_operator $var2
then
     Statement_1
else
     Statement_2
fi
```

III. Nested if:

A condition is defined within a condition statement.

```
if $var1 Comparision_operator $var2
then
        if condition
        then
            Statement_2

        else
            if condition
            then
                Statement_3
            else
                Statement_4
            fi
        fi
```

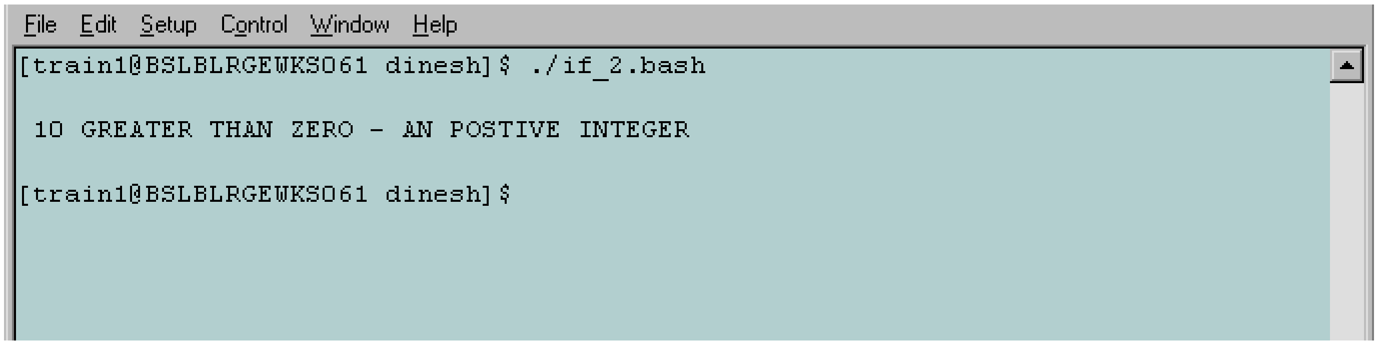
IV. Multi if else if:

If we want to check multiple conditions we will use multilevel if else statement.

```
if condition
then
      statement_1

elif condition1
then
      statement_2

elif condition2
then
       statement_3
else

        statement_4
fi
```

Method I:

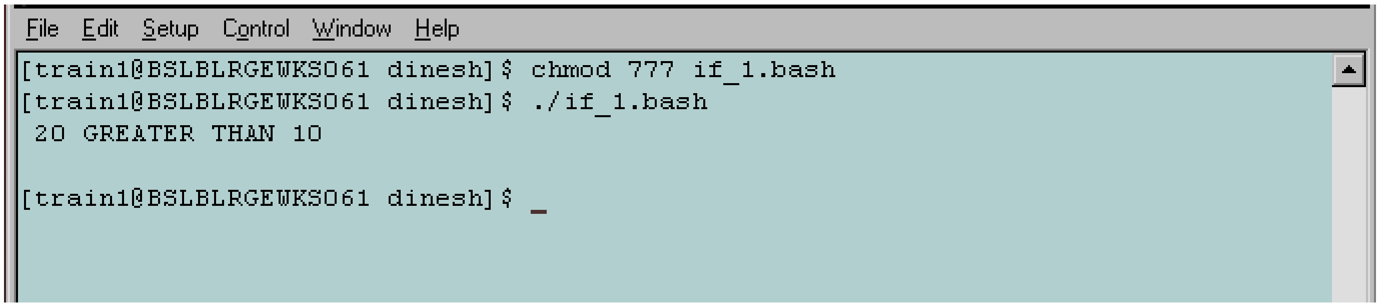
```
# This script to show usage of if statement Method 1

a=10

if (( "$a" > 0 ))
then
        echo -e "\n $a GREATER THAN ZERO - AN POSTIVE INTEGER \n"
fi
```



```
[train1@BSLBLRGEWKS061 dinesh]$ ./if_2.bash

 10 GREATER THAN ZERO - AN POSTIVE INTEGER

[train1@BSLBLRGEWKS061 dinesh]$
```

Method II:

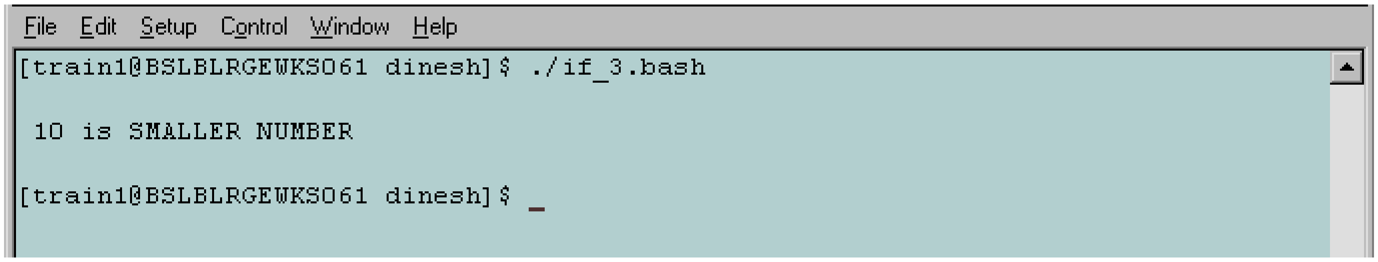
```
# This script to show the usage of simple if condition statement

a=10
b=20

if [ "$a" -gt "$b" ]
then
        echo -e " $a GREATER THAN $b \n"
else
        echo -e " $b GREATER THAN $a \n"
fi
```



```
[train1@BSLBLRGEWKS061 dinesh]$ chmod 777 if_1.bash
[train1@BSLBLRGEWKS061 dinesh]$ ./if_1.bash
 20 GREATER THAN 10

[train1@BSLBLRGEWKS061 dinesh]$ _
```

Method III:

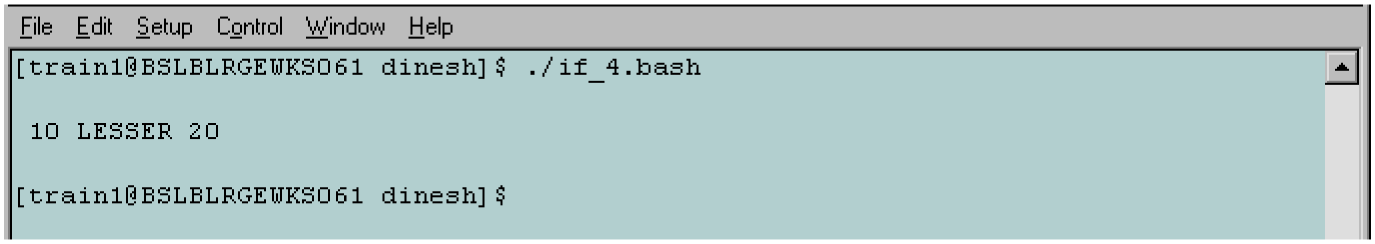
```
# This script to show the usage of nested if condition statement Method 3

a=10
b=20

if (( "$a" > 0 ))
then

        if [ "$a" -ge "$b" ]
        then
                echo -e " \n $a is GREATER NUMBER \n"
        else
                echo -e " \n $a is SMALLER NUMBER \n"
        fi
fi
```



```
[train1@BSLBLRGEWKS061 dinesh]$ ./if_3.bash

 10 is SMALLER NUMBER

[train1@BSLBLRGEWKS061 dinesh]$ _
```

Method IV:

```
# This script to show the usage of multi level if statement method 4

a=10
b=20

if [[ "$a" = "$b" ]]
then
        echo -e " \n $a EQUALS $b \n"
elif [[ "$a" > "$b" ]]
then
        echo -e " \n $a GREATER $b" \n"
elif [[ "$a" < "$b" ]]
then
        echo -e " \n $a LESSER $b \n"
else
        echo -e "\n CANNOT COMPARE \n"
fi
```



```
[train1@BSLBLRGEWKS061 dinesh]$ ./if_4.bash

 10 LESSER 20

[train1@BSLBLRGEWKS061 dinesh]$
```

Chapter 9

Looping Statements

Loop:

A Loop is a block of code which repeatedly executes statements until loop condition is satisfied.

Bash Scripting Supports

- **for Loop**
- **while Loop**
- **until Loop**

Points to note:

Variables in loop conditions should be initialized.
Before executing the loop body test condition should be satisfied.
In body of loop the test variable should be modified.

For Loop:

Syntax:

```
For Variable_name in [List]
Do
        Statements…..
Done
```

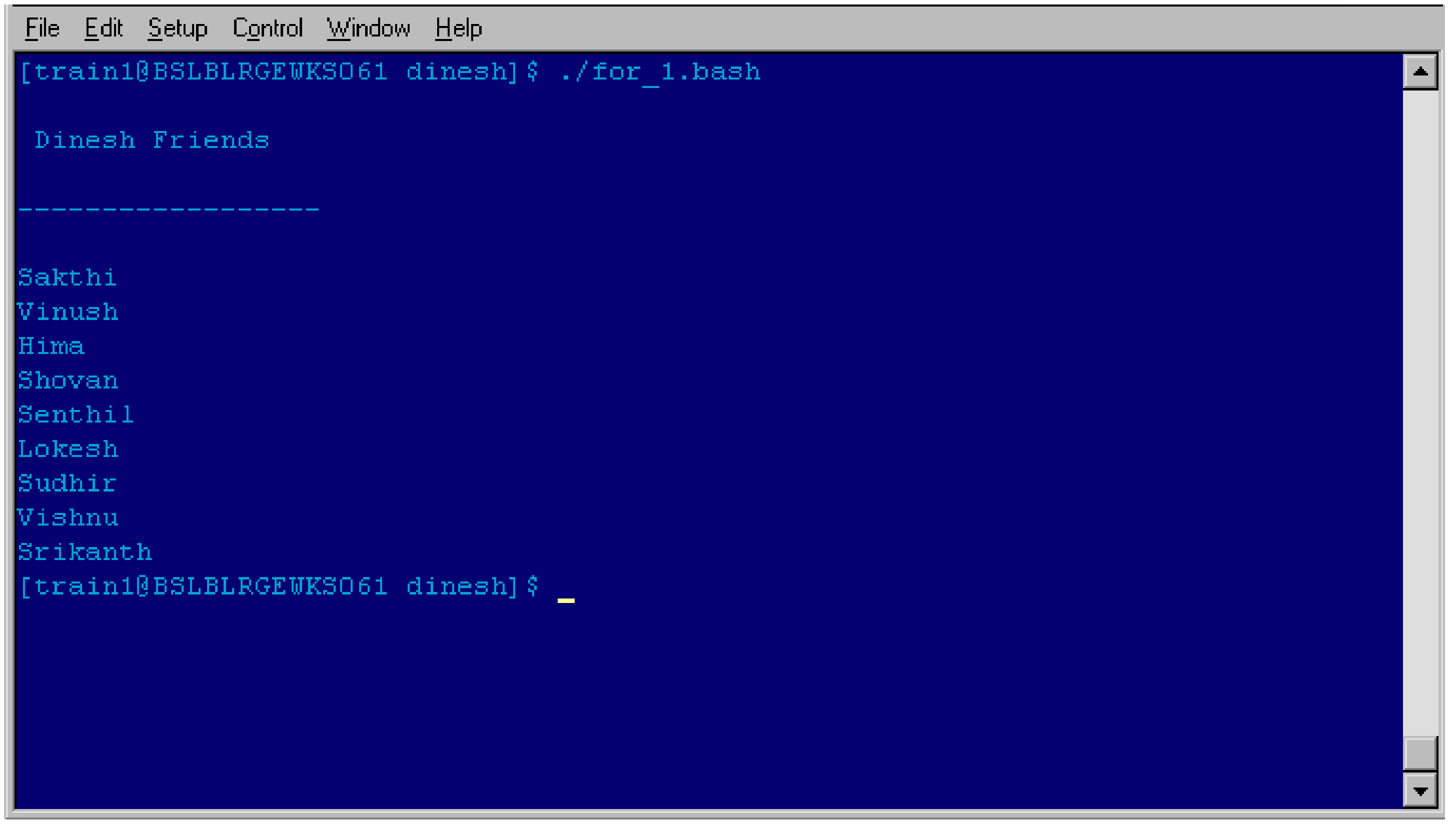
(Or)

```
For Variable_name in [List]; Do
        Statements…..
Done
```

Note: If "**do**" & "**for**" in same line then separate it by using semicolon "**;**".

Simple for Loop:

```
# This script to show the usage of simple for loop

  echo -e "\n Dinesh Friends \n"
  echo -e "------------------\n"

  for friends in Sakthi Vinush Hima Shovan Senthil Lokesh Sudhir Vishnu Srikanth
  do
      echo $friends
  done
```



```
[train1@BSLBLRGEWKS061 dinesh]$ ./for_1.bash

 Dinesh Friends

------------------

Sakthi
Vinush
Hima
Shovan
Senthil
Lokesh
Sudhir
Vishnu
Srikanth
[train1@BSLBLRGEWKS061 dinesh]$ _
```

Note:

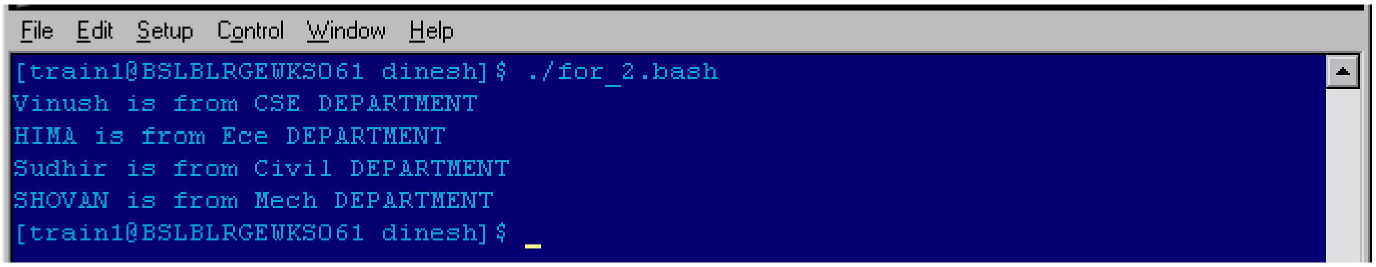
"Sakthi Vinush Hima Shovan Senthil Lokesh Sudhir Vishnu Srikanth"

If you give the list within "" then it becomes **single string**.

For loop with two Parameters:

Example 1:

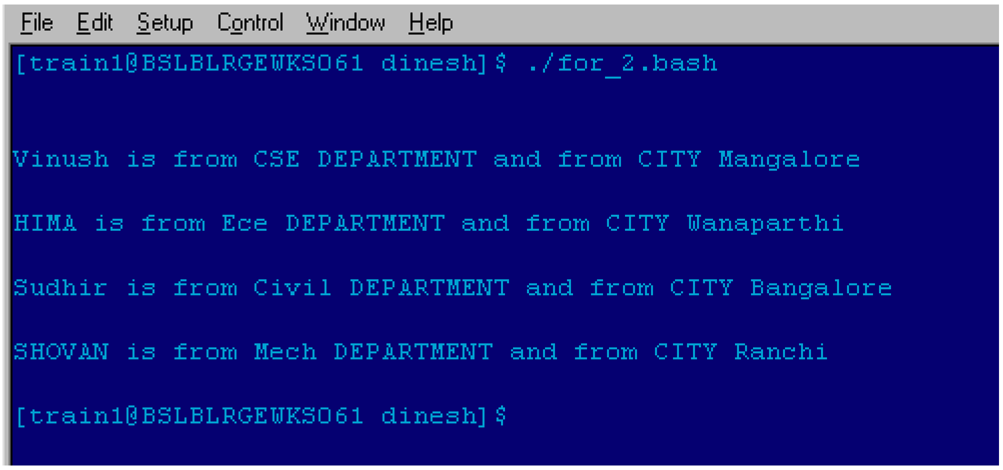
```
#This Script to show the usage of for lop with two parameters

for friends in "Vinush CSE" "HIMA Ece " "Sudhir Civil" "SHOVAN Mech"
do
    set -- $friends  # To Parses variable "friends"

    echo "$1 is from $2 DEPARTMENT"
done
```



```
[train1@BSLBLRGEWKS061 dinesh]$ ./for_2.bash
Vinush is from CSE DEPARTMENT
HIMA is from Ece DEPARTMENT
Sudhir is from Civil DEPARTMENT
SHOVAN is from Mech DEPARTMENT
[train1@BSLBLRGEWKS061 dinesh]$ _
```

Example 2:

```
#This Script to show the usage of for lop with two parameters

echo -e "\n"

for friends in "Vinush CSE Mangalore" "HIMA Ece Wanaparthi" "Sudhir Civil Bangalore" "SHOVAN Mech Ranchi"
do
    set -- $friends  # To Parses variable "friends"

    echo -e "$1 is from $2 DEPARTMENT and from CITY $3 \n"

done
```

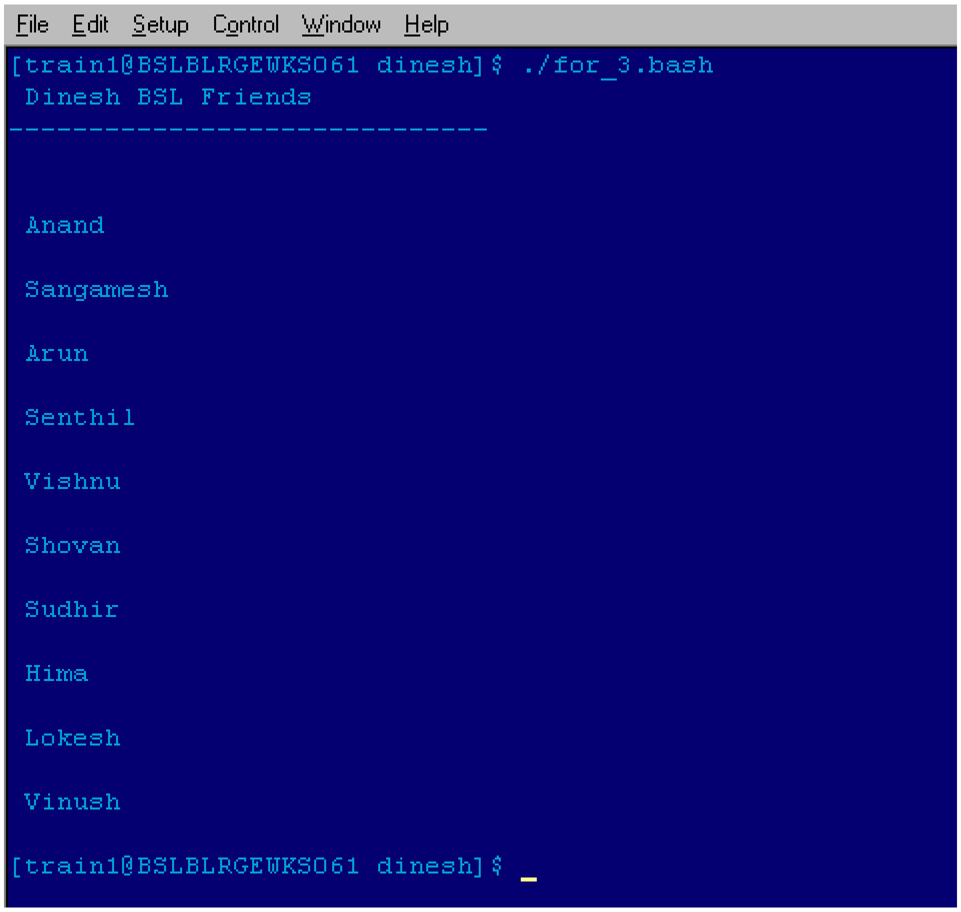


```
[train1@BSLBLRGEWKS061 dinesh]$ ./for_2.bash


Vinush is from CSE DEPARTMENT and from CITY Mangalore

HIMA is from Ece DEPARTMENT and from CITY Wanaparthi

Sudhir is from Civil DEPARTMENT and from CITY Bangalore

SHOVAN is from Mech DEPARTMENT and from CITY Ranchi

[train1@BSLBLRGEWKS061 dinesh]$
```

Note: Use can access multiple values in list using Positional parameters such as $1, $2….
In Example_1 **$1 → Vinush** **$2→ CSE**

Storing for Loop LIST in a Variable:

```
# This script to show storing LIST in a variable

dini="Anand
Sangamesh
Arun
Senthil
Vishnu
Shovan
Sudhir
Hima
Lokesh
Vinush"

echo -e " Dinesh BSL Friends"
echo "------------------------------"
echo -e "\n"

for friends in $dini
do

    echo -e " $friends \n"

done
```



```
[train1@BSLBLRGEWKS061 dinesh]$ ./for_3.bash
 Dinesh BSL Friends
------------------------------


 Anand

 Sangamesh

 Arun

 Senthil

 Vishnu

 Shovan

 Sudhir

 Hima

 Lokesh

 Vinush

[train1@BSLBLRGEWKS061 dinesh]$ _
```

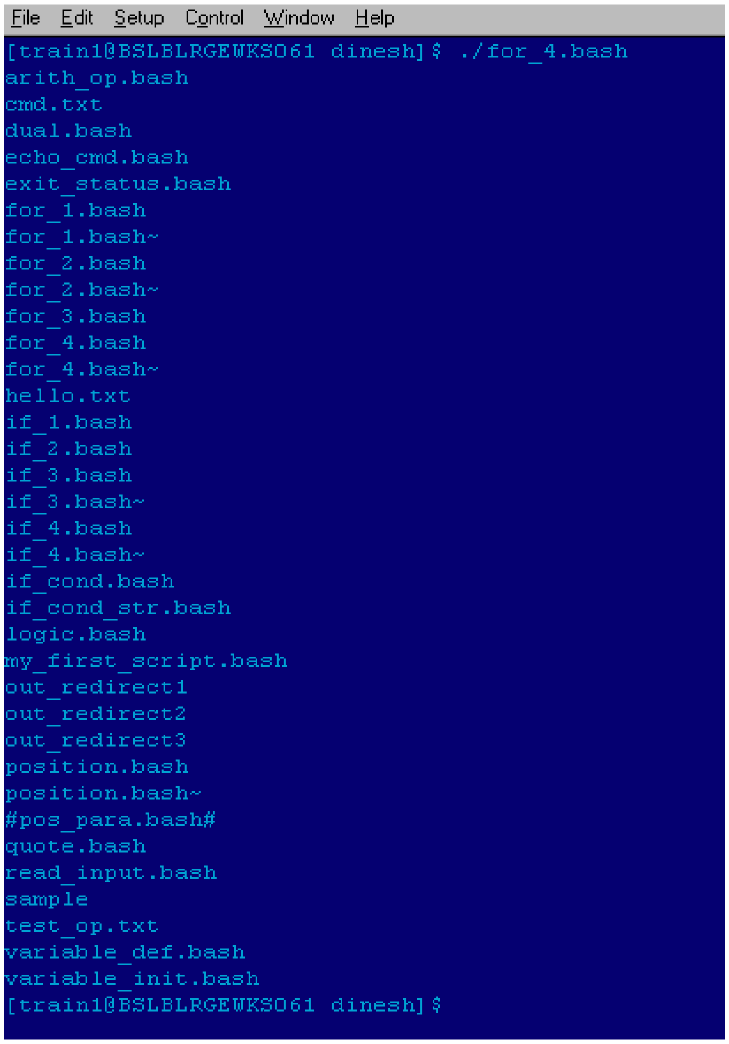
Operating files in for loops List:

Pattern Recognition:

Pattern	Description
*	Recognizes all file formats
[ab]*	Recognizes files beginning with '**a**' or '**b**'

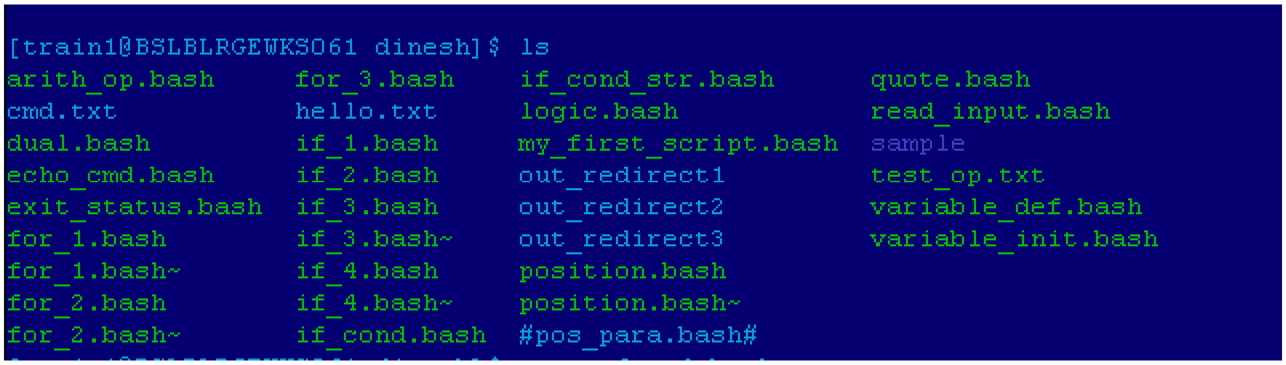
Example 1:

```
# This script to show the usage of files in for loop LIST
# * Displays all recoginized file formats

for ptr in *
do

    echo $ptr

done
```



```
[train1@BSLBLRGEWKS061 dinesh]$ ./for_4.bash
arith_op.bash
cmd.txt
dual.bash
echo_cmd.bash
exit_status.bash
for_1.bash
for_1.bash~
for_2.bash
for_2.bash~
for_3.bash
for_4.bash
for_4.bash~
hello.txt
if_1.bash
if_2.bash
if_3.bash
if_3.bash~
if_4.bash
if_4.bash~
if_cond.bash
if_cond_str.bash
logic.bash
my_first_script.bash
out_redirect1
out_redirect2
out_redirect3
position.bash
position.bash~
#pos_para.bash#
quote.bash
read_input.bash
sample
test_op.txt
variable_def.bash
variable_init.bash
[train1@BSLBLRGEWKS061 dinesh]$
```

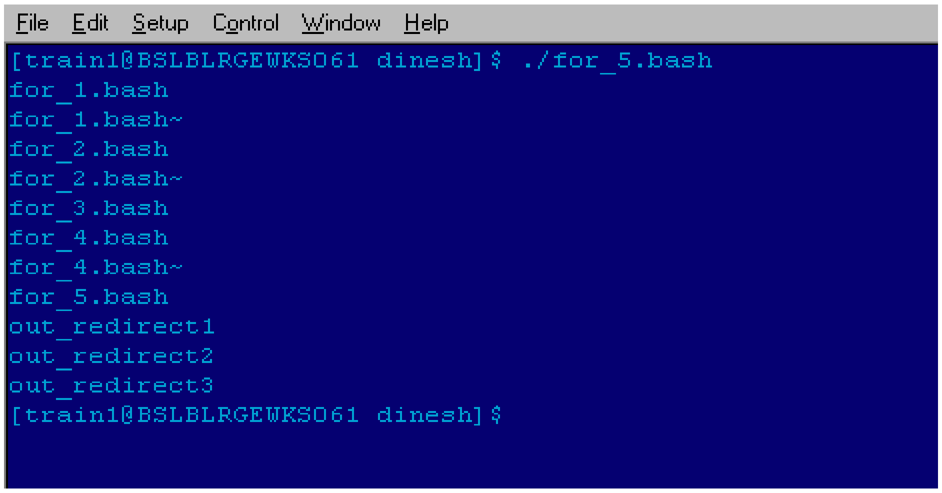
Files in the Present working Directory:



```
[train1@BSLBLRGEWKS061 dinesh]$ ls
arith_op.bash      for_3.bash     if_cond_str.bash      quote.bash
cmd.txt            hello.txt      logic.bash            read_input.bash
dual.bash          if_1.bash      my_first_script.bash  sample
echo_cmd.bash      if_2.bash      out_redirect1         test_op.txt
exit_status.bash   if_3.bash      out_redirect2         variable_def.bash
for_1.bash         if_3.bash~     out_redirect3         variable_init.bash
for_1.bash~        if_4.bash      position.bash
for_2.bash         if_4.bash~     position.bash~
for_2.bash~        if_cond.bash   #pos_para.bash#
```

Example 2:

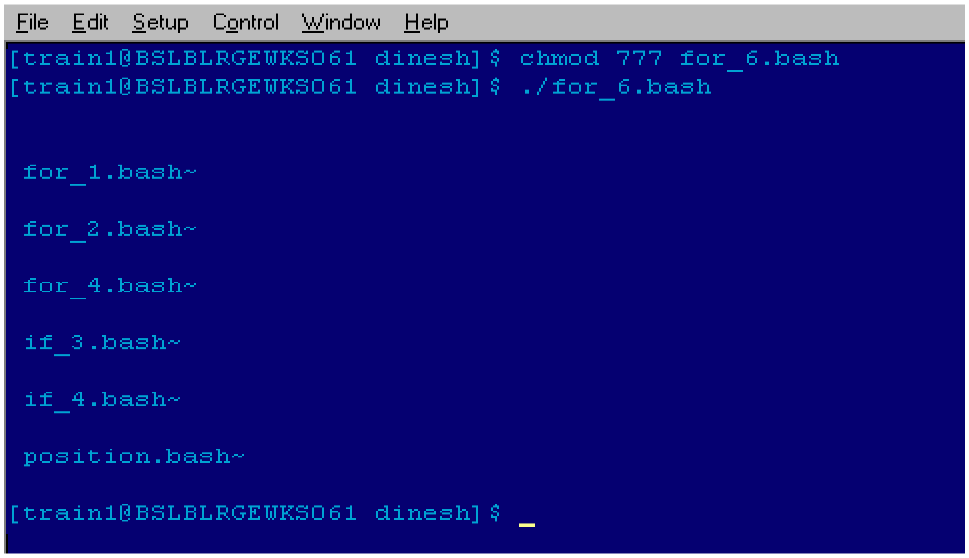
```
# This script to show the usage of files in for loop LIST
# * Displays all recognized file formats

for ptr in [fo]*
do

    echo $ptr

done
```



```
[train1@BSLBLRGEWKS061 dinesh]$ ./for_5.bash
for_1.bash
for_1.bash~
for_2.bash
for_2.bash~
for_3.bash
for_4.bash
for_4.bash~
for_5.bash
out_redirect1
out_redirect2
out_redirect3
[train1@BSLBLRGEWKS061 dinesh]$
```

Example 3:

```
# This script to show the usage of files in for loop LIST
# * Displays all recognized file formats

echo -e "\n"
for ptr in *[~]    #files ends with ~
do
    rm -f $ptr
    echo -e " $ptr \n"

done
```



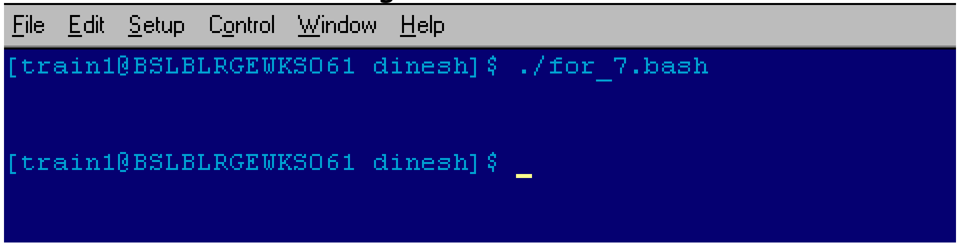
```
[train1@BSLBLRGEWKS061 dinesh]$ chmod 777 for_6.bash
[train1@BSLBLRGEWKS061 dinesh]$ ./for_6.bash


 for_1.bash~

 for_2.bash~

 for_4.bash~

 if_3.bash~

 if_4.bash~

 position.bash~

[train1@BSLBLRGEWKS061 dinesh]$ _
```

Excluding in [List] in for loop:

If we exclude it the loops works with positional parameter "**$@**" therefore script executes successfully.

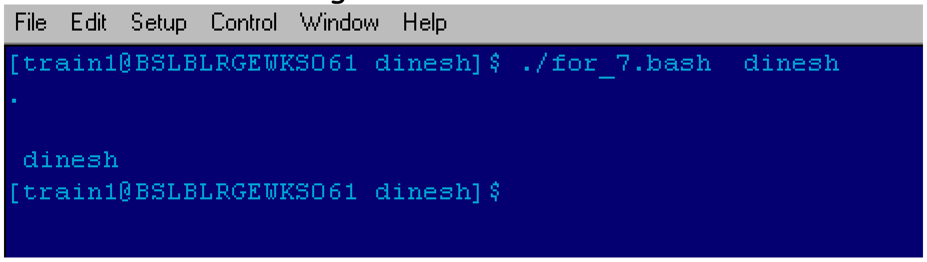
```
# This script to show behavihour of for loop in excluding in [list]

echo -e ".\n"
for ptr
do

    echo -ne " $ptr "

done

echo -e "\n"
```

Without Command line arguments:



```
[train1@BSLBLRGEWKS061 dinesh]$ ./for_7.bash



[train1@BSLBLRGEWKS061 dinesh]$ _
```

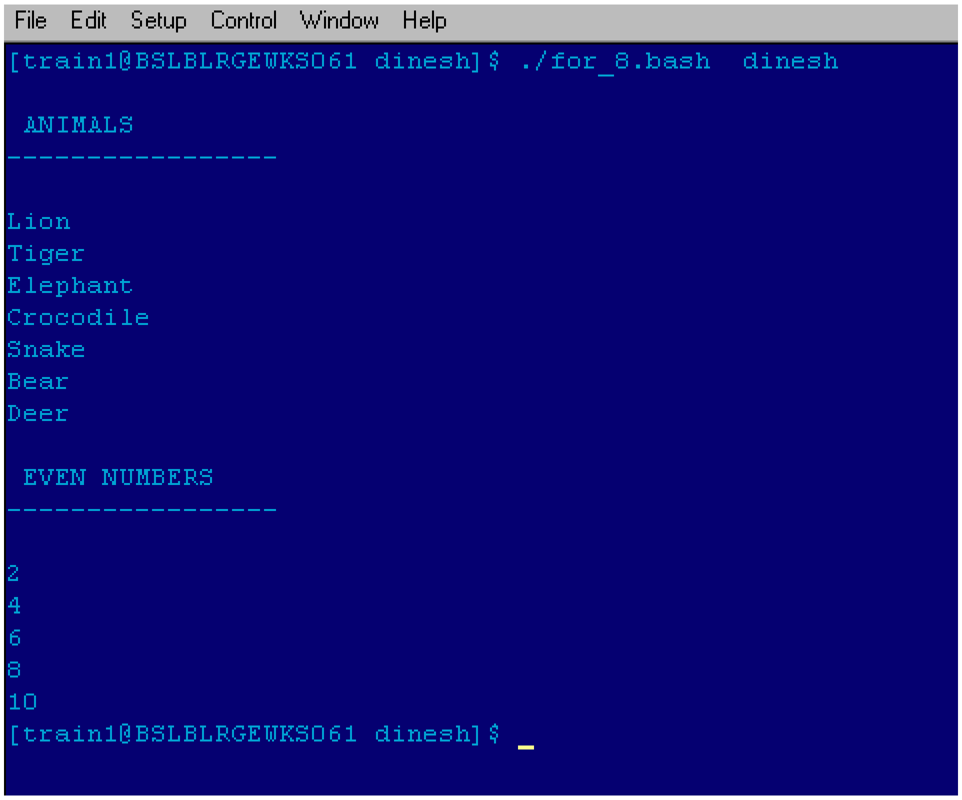
With Command line arguments:



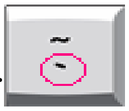
```
[train1@BSLBLRGEWKS061 dinesh]$ ./for_7.bash  dinesh
.

 dinesh
[train1@BSLBLRGEWKS061 dinesh]$
```

Command Substitution for [List] in for loop:

```
# This script to show how to substitute COMMAND instead of LIST in for loop

animals="Lion Tiger Elephant Crocodile Snake Bear Deer"

even_no="2 4 6 8 10"

echo -e "\n ANIMALS"
echo -e "-----------------\n"

for dini in `echo $animals`
do

    echo -e "$dini"

done

echo -e "\n EVEN NUMBERS"
echo -e "-----------------\n"

for dini2 in `echo $even_no`
do

    echo -e "$dini2"

done
```

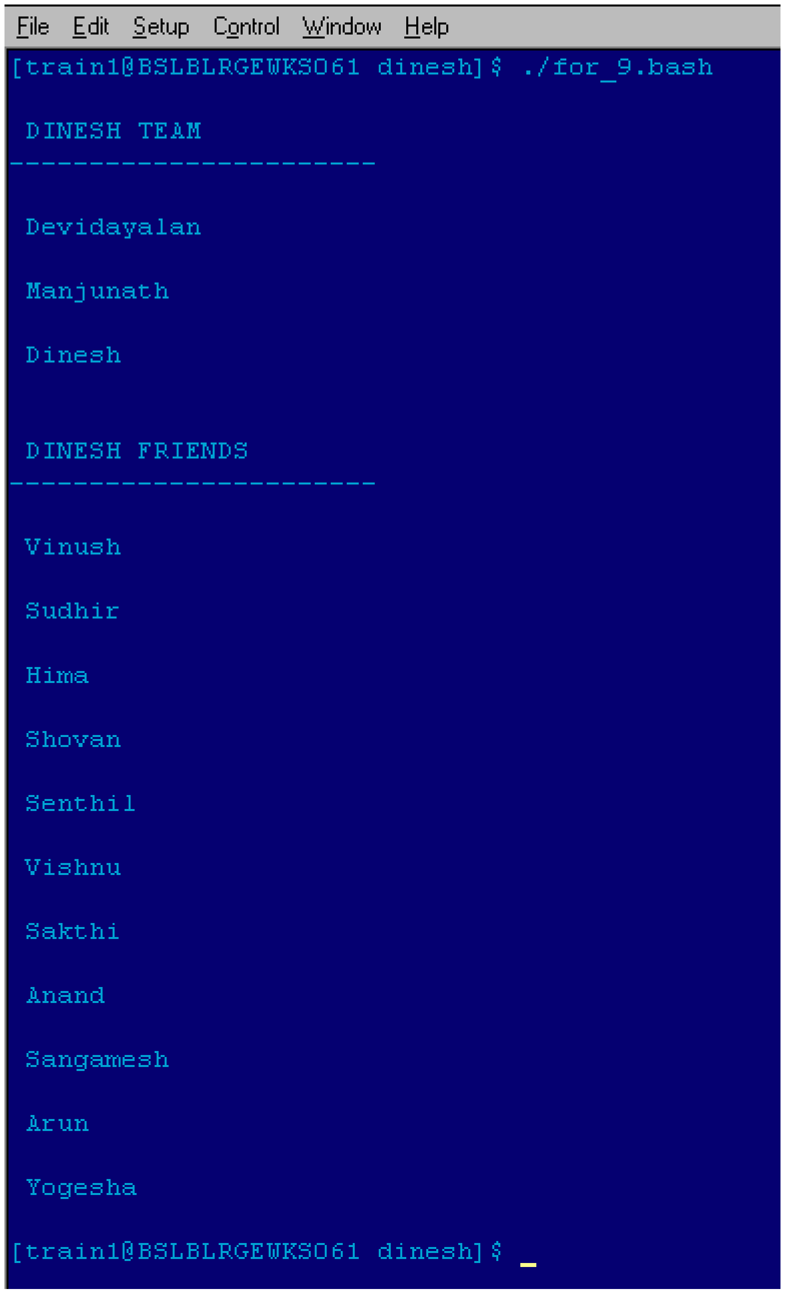


```
[train1@BSLBLRGEWKS061 dinesh]$ ./for_8.bash  dinesh

 ANIMALS
-----------------


Lion
Tiger
Elephant
Crocodile
Snake
Bear
Deer

 EVEN NUMBERS
-----------------


2
4
6
8
10
[train1@BSLBLRGEWKS061 dinesh]$ _
```

Note: While executing command use back quote. 

Function Substitution for [List] in for loop:

```
# This script to show how to subtitute function instead of LIST in for loop

dinesh_team ()
{
    echo "Devidayalan Manjunath Dinesh"
}

dinesh_friends ()
{
    echo "Vinush Sudhir Hima Shovan Senthil Vishnu Sakthi Anand Sangamesh Arun Yogesha"
}

echo -e "\n DINESH TEAM"
echo -e "------------------------\n"

for team in $(dinesh_team)
do

    echo -e " $team \n"

done

echo -e "\n DINESH FRIENDS"
echo -e "------------------------\n"

for friends in $(dinesh_friends)
do

    echo -e " $friends \n"

done
```



```
[train1@BSLBLRGEWKS061 dinesh]$ ./for_9.bash

 DINESH TEAM
-----------------------

 Devidayalan

 Manjunath

 Dinesh


 DINESH FRIENDS
-----------------------

 Vinush

 Sudhir

 Hima

 Shovan

 Senthil

 Vishnu

 Sakthi

 Anand

 Sangamesh

 Arun

 Yogesha

[train1@BSLBLRGEWKS061 dinesh]$ _
```

II. While loop:

While loop checks the condition first & execute the loop statements till the condition is satisfied. This loop is compliment to **for loop**. This loop is mainly used when the **loop repetition** is not known.

Syntax:

```
While [ condition ]
do
        Statements….
done
```

(Or)

```
While [ condition ]; do

        Statements….
done
```

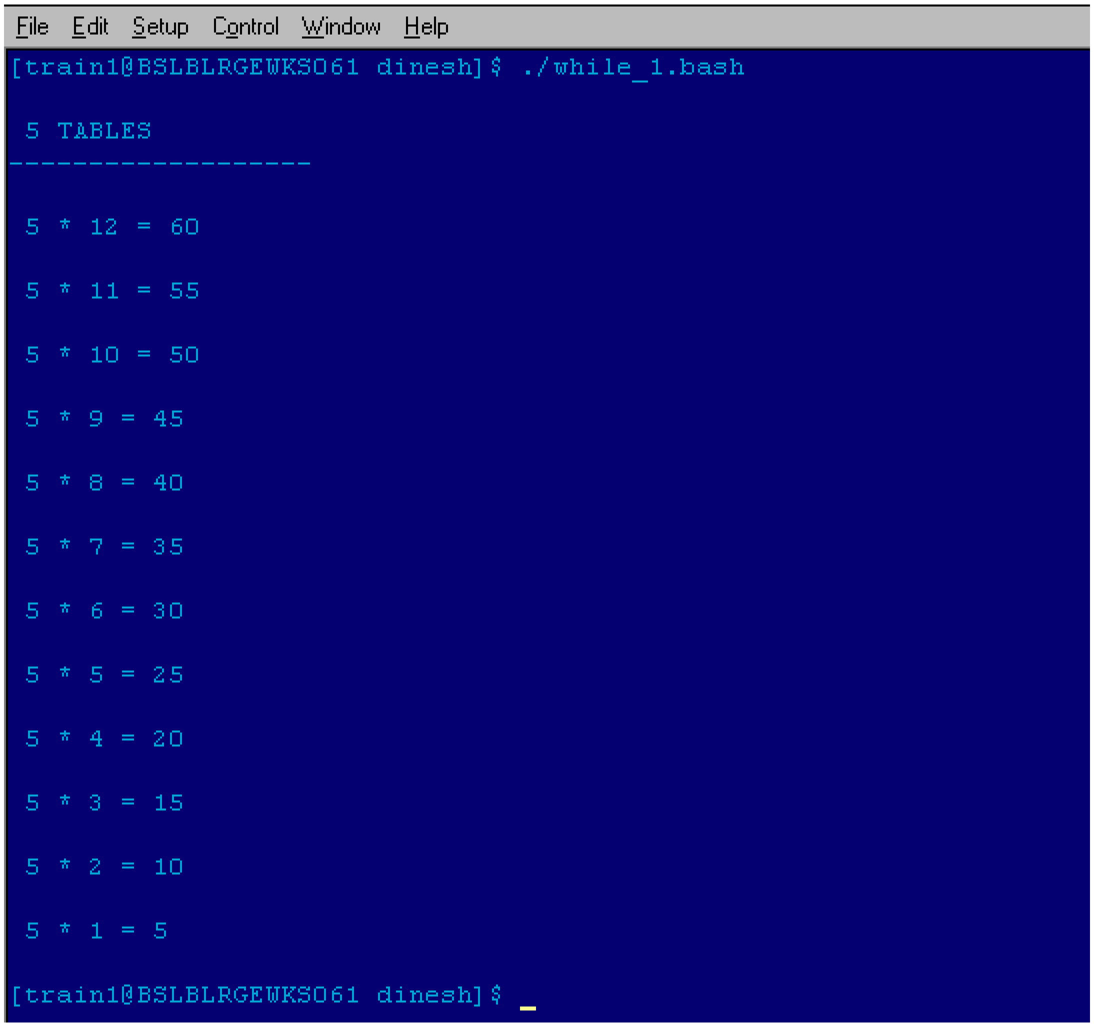
Note: If "**do**" & "**for**" in same line then separate it by using semicolon "**;**".

(Or)

```
While [[ condition ]]; do

        Statements….
done
```

Simple while loop:

```
# This script shows how to write a simple while statement
# Script to manipulate 5 tables

max=12
echo -e " \n 5 TABLES"
echo -e "--------------------\n"

while [ "$max" -gt 0 ]
do
    let "result=5*$max"
    echo -e " 5 * $max = $result \n"

    let "max=max-1"
done
```



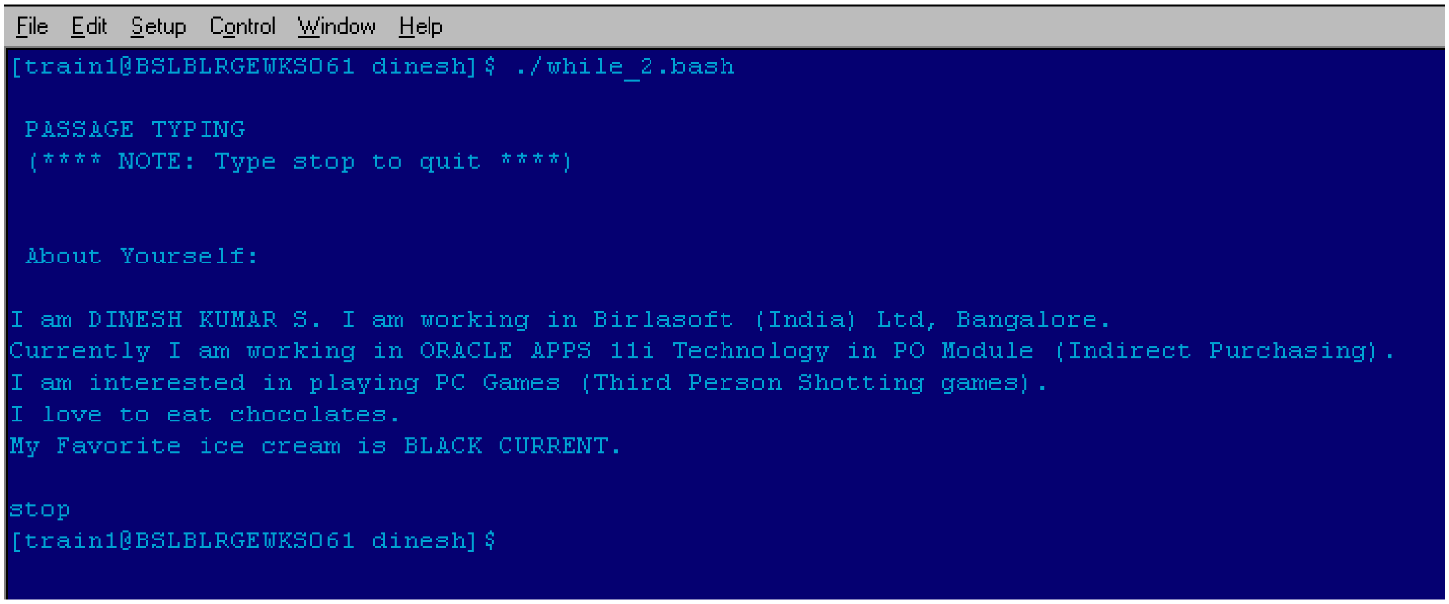
```
[train1@BSLBLRGEWKS061 dinesh]$ ./while_1.bash

 5 TABLES
--------------------

 5 * 12 = 60

 5 * 11 = 55

 5 * 10 = 50

 5 * 9 = 45

 5 * 8 = 40

 5 * 7 = 35

 5 * 6 = 30

 5 * 5 = 25

 5 * 4 = 20

 5 * 3 = 15

 5 * 2 = 10

 5 * 1 = 5

[train1@BSLBLRGEWKS061 dinesh]$ _
```

Using string in while loop condition:

I can use strings in while test condition to compare with other string.

In the below example I am going to read variables from user until he/she types the word "**stop**".

```
# This script shows how to use STRINGS in whil test conditions

echo -e "\n PASSAGE TYPING "
echo -e " (**** NOTE: Type stop to quit ****) \n"
echo -e "\n About Yourself:  "
echo

while [ "$var1" != "stop" ]
do

   read var1

done
```



```
[train1@BSLBLRGEWKS061 dinesh]$ ./while_2.bash

 PASSAGE TYPING
 (**** NOTE: Type stop to quit ****)


 About Yourself:

I am DINESH KUMAR S. I am working in Birlasoft (India) Ltd, Bangalore.
Currently I am working in ORACLE APPS 11i Technology in PO Module (Indirect Purchasing).
I am interested in playing PC Games (Third Person Shotting games).
I love to eat chocolates.
My Favorite ice cream is BLACK CURRENT.

stop
[train1@BSLBLRGEWKS061 dinesh]$
```

Multiple Statements in while loop:

We can give multiple statements; each statement should be given in **separate line**. In this case the loop control is decided by **last statement**.

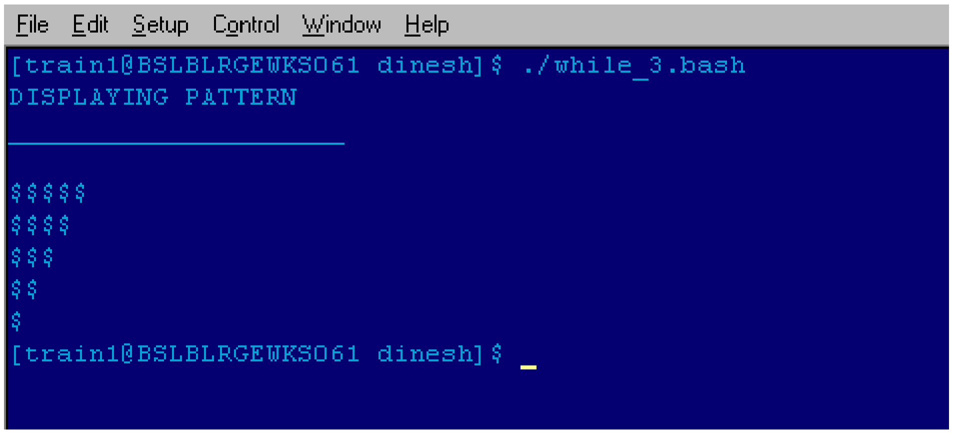
Syntax:

```
While statement_1
        statement_2
        statement_3
        statement_4                  # this statement takes loop control (i.e. Test condition)
do
        body_statements….
done
```

(Or)

```
While statement_1
        statement_2
        statement_3
        statement_4; do

body_statements….

done
```

```
# This script to show usage of multiple statements in WHILE LOOP

chr='$'
max=6
echo -e "DISPLAYING PATTERN"
echo -e "_____________________"

while   let "max=max-1"          # Decrementing the max value
        echo                     # echo statement
        [ $max -gt 0 ]           # Condition statement which control while loop
do
            max2=$max

            while [ $max2 != 0 ] # this while loop to display $ no. of times
            do
                echo -n "$chr"
                let "max2=max2-1"
            done
done
```



```
[train1@BSLBLRGEWKS061 dinesh]$ ./while_3.bash
DISPLAYING PATTERN
_____________________

$$$$$
$$$$
$$$
$$
$
[train1@BSLBLRGEWKS061 dinesh]$ _
```

Statement 1: `let "max=max-1"`

Statement 2: `echo`

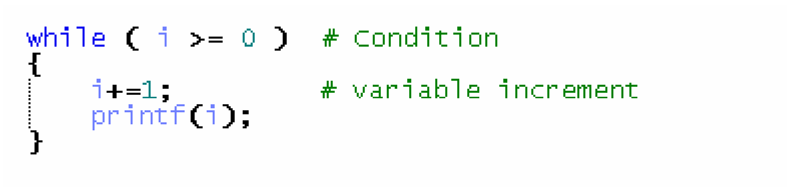
Statement 3: `[ $max -gt 0 ]` # this condition controls the loop

C Programming syntax in while loop:

C Syntax:

While [Condition)]
{
Statements…
}

Example:



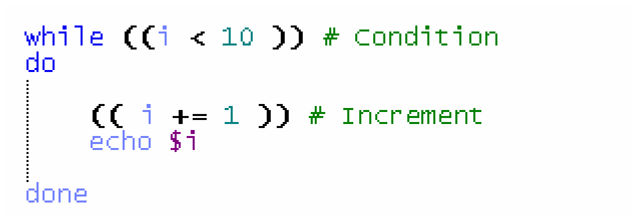
```
while ( i >= 0 )   # condition
{
    i+=1;          # variable increment
    printf(i);
}
```

Unix Syntax:

To execute condition & variable increment in same way UNIX uses the syntax below.

While ((Condition))
do
Statements….
done

Example:



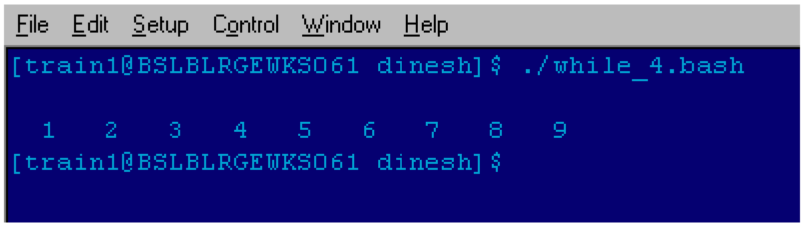
```
while ((i < 10 )) # condition
do

    (( i += 1 )) # Increment
    echo $i

done
```

Note: When we use **(( ))** double parenthesis two points to be noted:

1. It helps in making a mathematical calculation without using **'$' or 'let'** or **'expr'** command.
2. Also it helps in incrementing variable like **C** syntax.

Example 1:

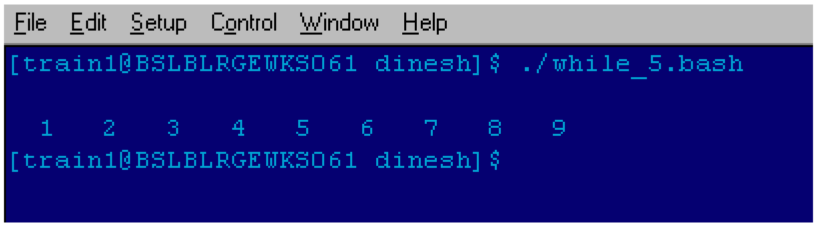
```
# This script to show how to use C type syntax in while loop

(( min = 1))
     echo

while (( min < 10 ))
do
     echo -n "  $min "
     ((min += 1))

done
     echo
```


File Edit Setup Control Window Help

```
[train1@BSLBLRGEWKS061 dinesh]$ ./while_4.bash

  1    2    3    4    5    6    7    8    9
[train1@BSLBLRGEWKS061 dinesh]$
```

Example 2:

```
# This script to show how to use C type syntax in while loop Method 2

(( min = 1))
     echo

while (( min < 10 ))
do
     echo -n "  $min "
     ((min++))                  # C Syntax

done
     echo
```


File Edit Setup Control Window Help

```
[train1@BSLBLRGEWKS061 dinesh]$ ./while_5.bash

  1    2    3    4    5    6    7    8    9
[train1@BSLBLRGEWKS061 dinesh]$
```

While loop calling a function inside test brackets:

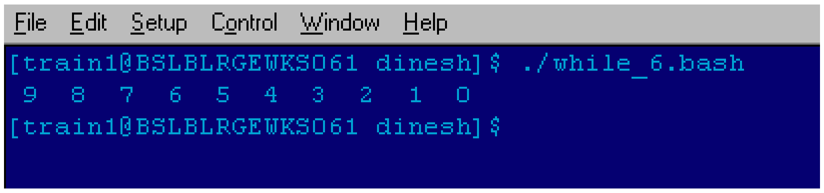
Syntax:

```
Function_name ()
{
        Statements….
}

While function_name
Do
        Statements….
Done
```

(Or)

```
While function_name; do
        Statements….
done
```

```
# This script to show how to call a function inside while loop test brackets

(( t=10))    # intialize

check ()
{
  if [[ t != 0 ]]    # Condition checked in IF statement instead of in WHILE loop
  then
       (( t-- ))     # Decrement value
  fi
}

while check
do
    echo -n " $t "

done

echo
```



```
[train1@BSLBLRGEWKS061 dinesh]$ ./while_6.bash
 9  8  7  6  5  4  3  2  1  0
[train1@BSLBLRGEWKS061 dinesh]$
```

Reading a file using while loop:

We can read a file with the help of while loop. For reading we can use either '**cat**' or '**More**' command.

Ways to read a file:

1. **Line by line**
2. **Value by value**

Method 1:

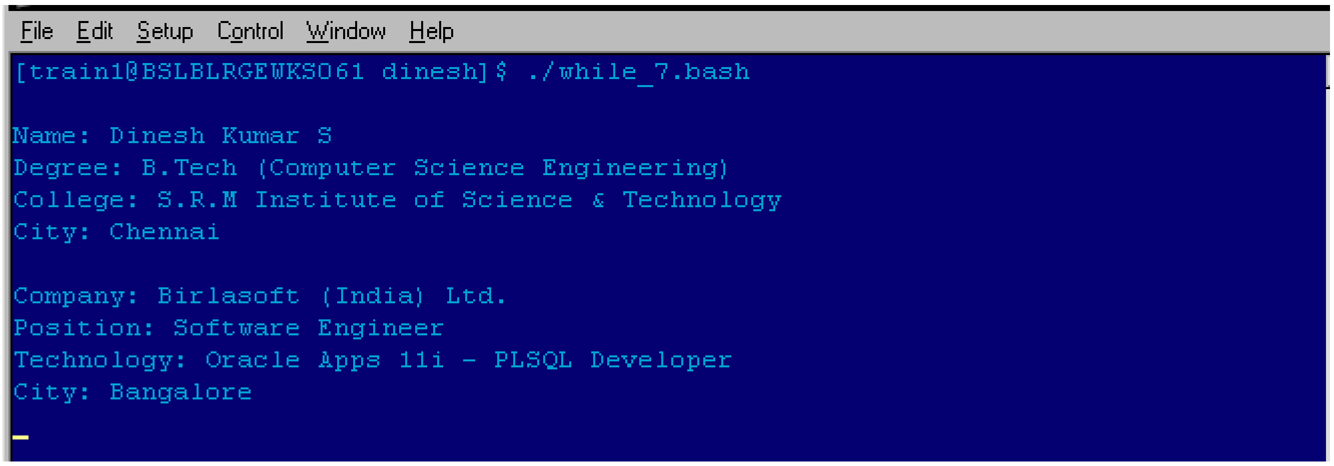
```
While read line
do
        echo "$line"
done
```

Method 2:

```
While read Value
do
        echo "$value"
done
```

Example 1:

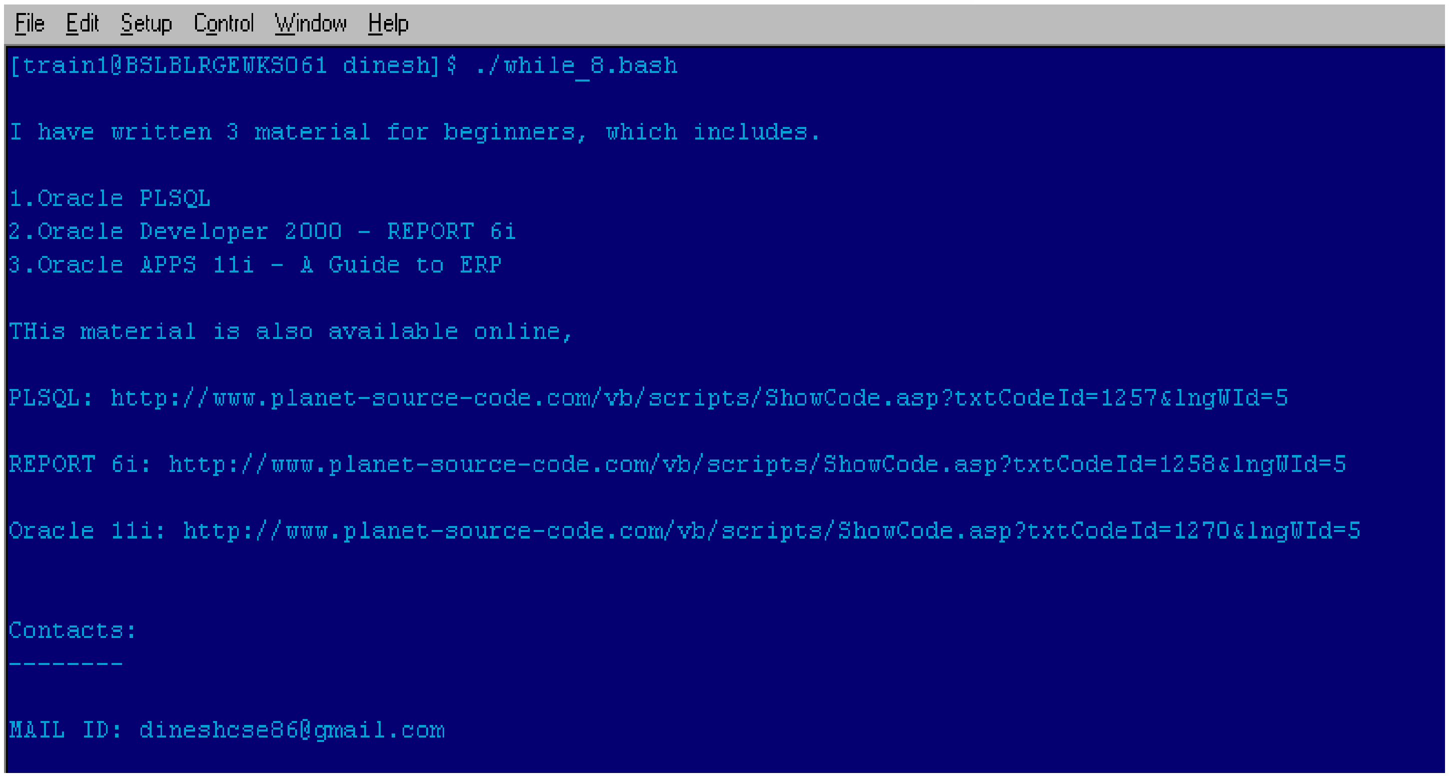
```
# This script to show how to read a file using while loop
# read line reads the file line by line as a whole

more about_me.txt

while read line
do
        echo $line
done
exit 0
```



```
[train1@BSLBLRGEWKS061 dinesh]$ ./while_7.bash

Name: Dinesh Kumar S
Degree: B.Tech (Computer Science Engineering)
College: S.R.M Institute of Science & Technology
City: Chennai

Company: Birlasoft (India) Ltd.
Position: Software Engineer
Technology: Oracle Apps 11i - PLSQL Developer
City: Bangalore

_
```

Example 2:

```
# This script to show how to read a file using while loop
# read line reads the file value by value as a whole

more dini_material.txt

while read value
do
     echo $value
done
exit 0
```



```
[train1@BSLBLRGEWKS061 dinesh]$ ./while_8.bash

I have written 3 material for beginners, which includes.

1.Oracle PLSQL
2.Oracle Developer 2000 - REPORT 6i
3.Oracle APPS 11i - A Guide to ERP

THis material is also available online,

PLSQL: http://www.planet-source-code.com/vb/scripts/ShowCode.asp?txtCodeId=1257&lngWId=5

REPORT 6i: http://www.planet-source-code.com/vb/scripts/ShowCode.asp?txtCodeId=1258&lngWId=5

Oracle 11i: http://www.planet-source-code.com/vb/scripts/ShowCode.asp?txtCodeId=1270&lngWId=5


Contacts:
--------

MAIL ID: dineshcse86@gmail.com
```

III.Until loop:

Until loop check condition at first and executes the statements in loop body till the condition is '**false**'.

This is **compliment** to '**while loop**'. Also it checks for '**Termination Condition**' at top of the loop.

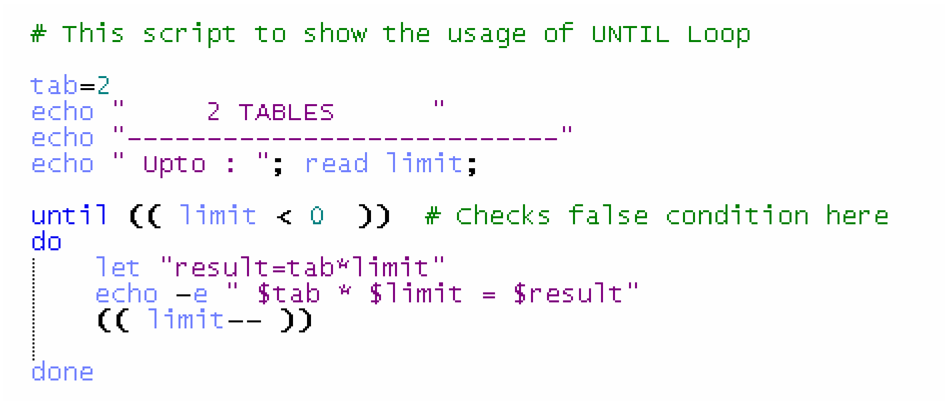
Syntax:

Until [Checks True_Condition]
do
        statements…
done

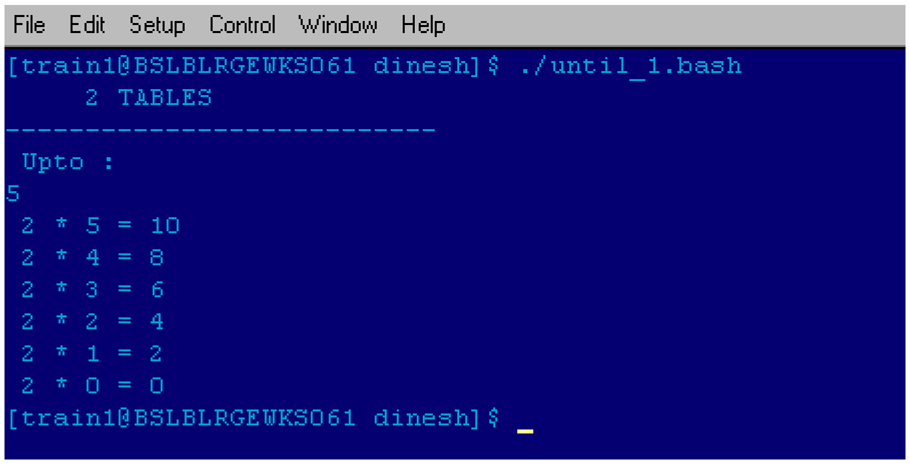
(Or)

Until [Checks True_Condition]; do

        statements…
done



```
# This script to show the usage of UNTIL Loop

tab=2
echo "      2 TABLES       "
echo "----------------------------"
echo " Upto : "; read limit;

until (( limit < 0  ))  # Checks false condition here
do
    let "result=tab*limit"
    echo -e " $tab * $limit = $result"
    (( limit-- ))

done
```



```
[train1@BSLBLRGEWKS061 dinesh]$ ./until_1.bash
     2  TABLES
---------------------------
 Upto :
5
 2  *  5  =  10
 2  *  4  =  8
 2  *  3  =  6
 2  *  2  =  4
 2  *  1  =  2
 2  *  0  =  0
[train1@BSLBLRGEWKS061 dinesh]$ _
```

Explanation:

Iteration 1: 5 < 0 False { 2 * 5 =10 }

Iteration 2: 4 < 0 False { 2 * 4 =8 }

Iteration 3: 3 < 0 False { 2 * 3 =6 }

Iteration 4: 2 < 0 False { 2 * 2 =4 }

Iteration 5: 1 < 0 False { 2 * 1 =2 }

Iteration 6: 0 < 0 False { 2 * 0 =0 }

Iteration 7: -1 < 0 True Exit since condition not satisfied

Note:

The statements in body is executed only when condition is 'false' than is when checking the condition it should be compliment or opposite one. Then only control flow passes into loop.

Nested Loops:

Loop within a loop is called 'Nested loop'.

Syntax:

```
For Variable_name in [List]
Do
        Statements…..

        For Variable_name in [List]
        Do
                Statements…..
        Done
Done
```

(Or)

```
While [ condition ]
do
        Statements….

        While [ condition ]
        do
                Statements….
        done
done
```

(Or)

```
Until [Checks True_Condition]
do
        statements…
        Until [Checks True_Condition]
        do
                statements…
        done
done
```

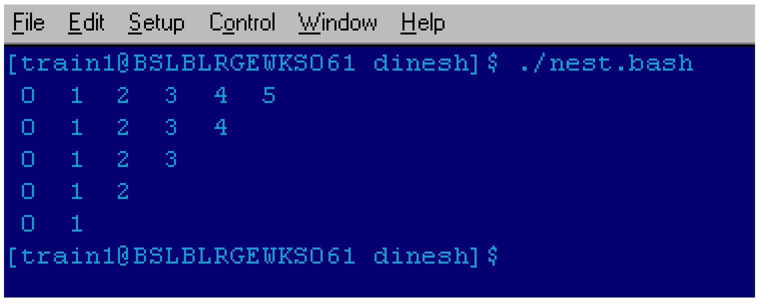
Example:

Script to display the below pattern:

```
0  1  2  3  4  5
0  1  2  3  4
0  1  2  3
0  1  2
0  1
```

```
# This Script to show how to write nested loop

min=0
max=5

    while (( max > 0 ))
    do
        while (( min <= max ))
        do
            echo -n " $min "

            (( min++ ))
        done

        echo
        min=0

        (( max-- ))
    done
```

Expected output:



```
[train1@BSLBLRGEWKS061 dinesh]$ ./nest.bash
 0  1  2  3  4  5
 0  1  2  3  4
 0  1  2  3
 0  1  2
 0  1
[train1@BSLBLRGEWKS061 dinesh]$
```

Commands affecting Loop Control & behavior:

There are two commands which affect the loop behavior,

1. break
2. continue

I. Break:

Break command terminates the loop that is comes to end of program.

Example:

```
if  [ condition ]
then
        break;
fi
```

II. Continue:

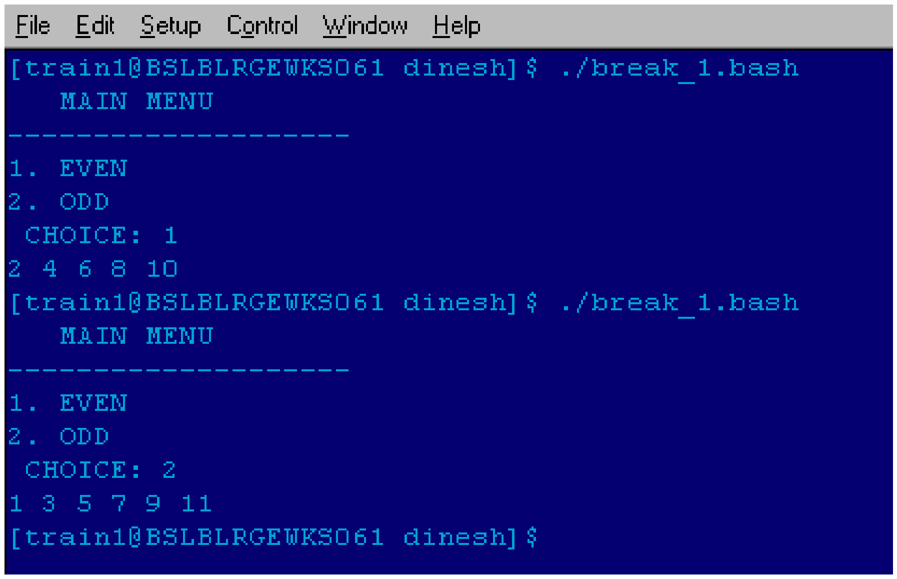
Jump command jumps to next iteration of the loop, by skipping remaining commands in the loop.

Example:

```
if  [ condition ]
then
        Continue;
fi
```

Example 1: This script to demonstrate how to use continue statement.

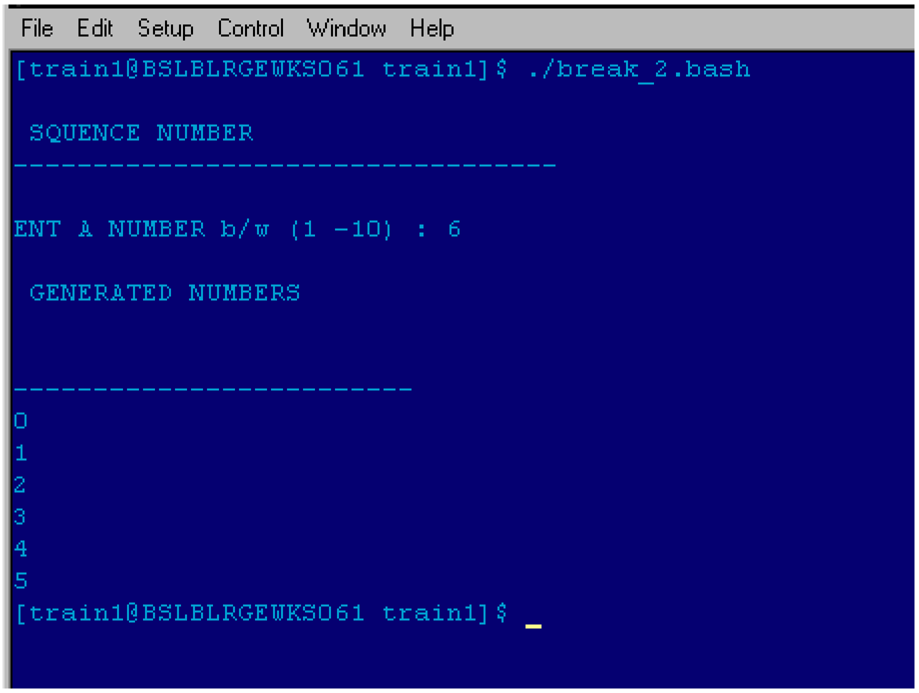
```
LIMIT=10
a=0

echo "   MAIN MENU   "
echo "---------------------"
echo "1. EVEN "; echo "2. ODD"
echo -n " CHOICE: "; read cho

if [ "$cho" -eq 1 ]
then
while [ $a -le "$LIMIT" ]
 do
 (( a++ ))

 let "c=a%2"

 if [ "$c" -ne 0 ]
 then
 continue
 fi

 echo -n "$a "

 done
 echo

else
while [ $a -le "$LIMIT" ]
 do
 (( a++ ))

 let "c=a%2"

 if [ "$c" -eq 0 ]
 then
 continue
 fi

 echo -n "$a "
 done
 echo
fi
```



```
[train1@BSLBLRGEWKS061 dinesh]$ ./break_1.bash
   MAIN MENU
--------------------
1. EVEN
2. ODD
 CHOICE: 1
2 4 6 8 10
[train1@BSLBLRGEWKS061 dinesh]$ ./break_1.bash
   MAIN MENU
--------------------
1. EVEN
2. ODD
 CHOICE: 2
1 3 5 7 9 11
[train1@BSLBLRGEWKS061 dinesh]$
```

Example 2: This script to demonstrate how to use Break statement.

```
# This script to show how to use break statement

i=0
echo -e " \n SQUENCE NUMBER "
echo -e "----------------------------------\n"

echo -n "ENT A NUMBER b/w (1 -10) : "; read num
echo -e "\n GENERATED NUMBERS \n"
echo -e "\n-------------------------"
while [ "$i" > 0 ]
do
    if [ "$i" -eq "$num" ]
    then
        break;
    fi

echo "$i"

(( i++ ))

done
```



```
[train1@BSLBLRGEWKS061 train1]$ ./break_2.bash

 SQUENCE NUMBER
----------------------------------

ENT A NUMBER b/w (1 -10) : 6

 GENERATED NUMBERS


-------------------------
0
1
2
3
4
5
[train1@BSLBLRGEWKS061 train1]$ _
```

Chapter 10

Control Statements

Control Statements direct the program flow based on the condition.

- **Case statement**
- **Select statement**

Case Statement:

Case statement is an alternative to multi level if then else statement. It is similar to switch case in C++.

Syntax:

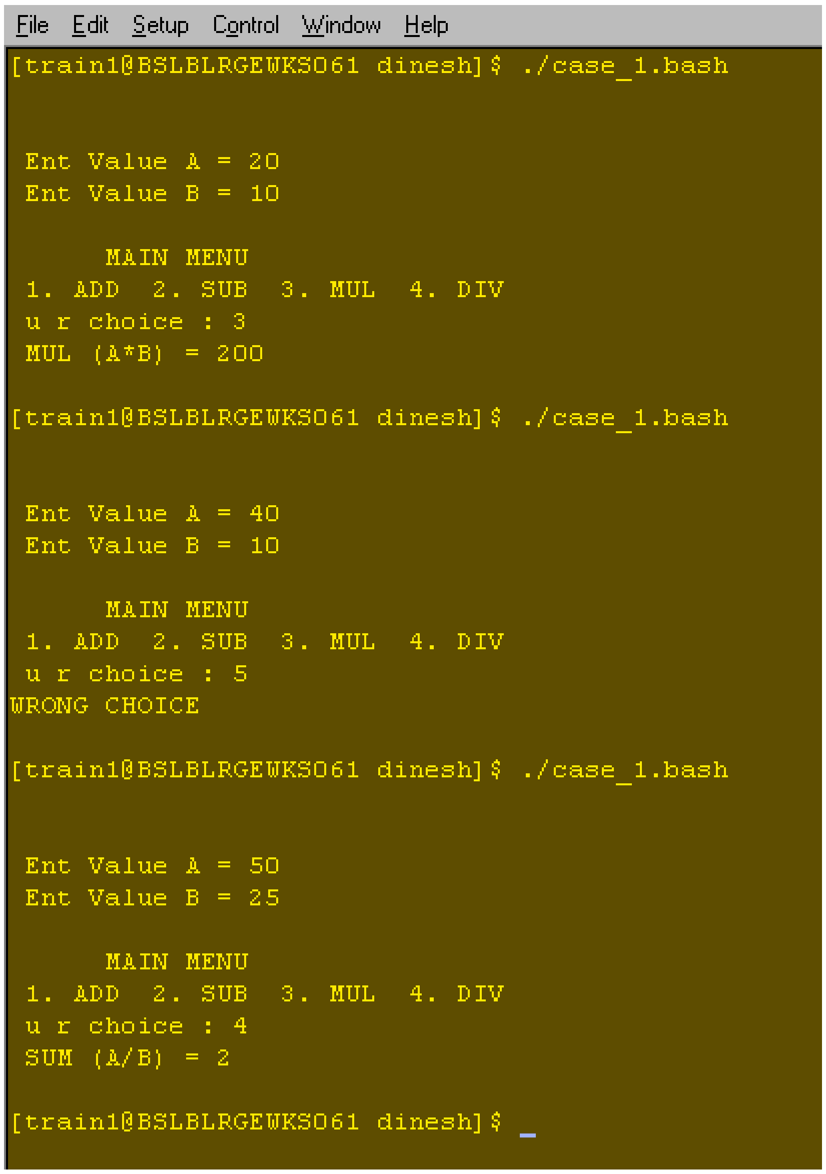
```
Case "$variable" in

"Condition_1") Command;;
"Condition_2") Command;;
.
"Condition_n") Command;;
* ) Command                          # Default Option
esac
```

(Or)

```
Case "$variable" in
"Condition_1") Command;;
"Condition_2") Command;;
.
"Condition_n") Command;;
esac
```

Example 1: Menu Program

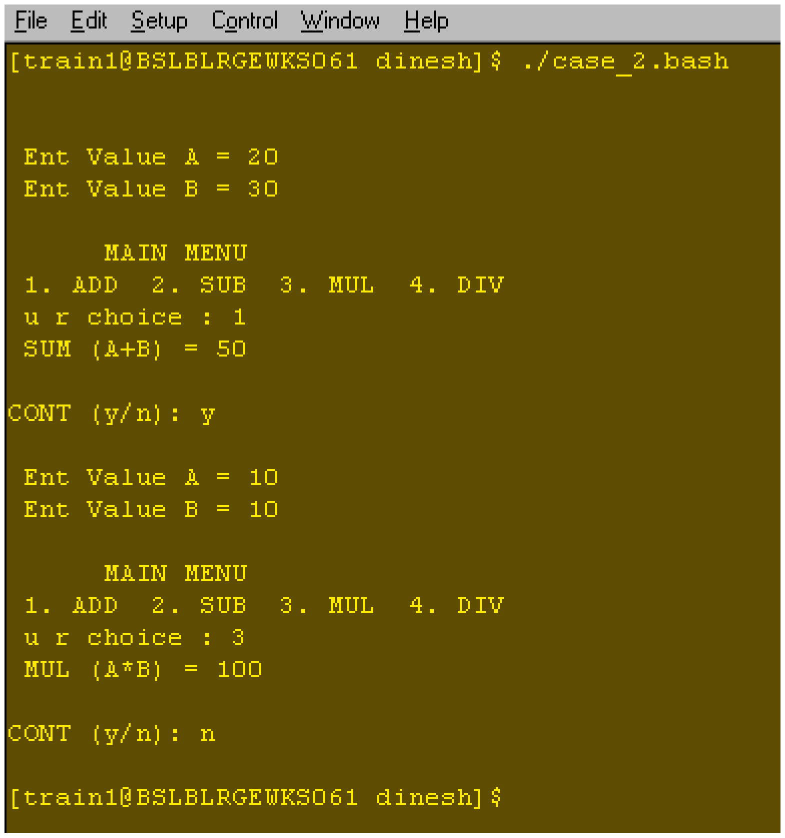
```
# This script to show how to use case statement

echo
echo
echo -n " Ent Value A = "; read a
echo -n " Ent Value B = "; read b

echo -e "\n       MAIN MENU     "
echo " 1. ADD  2. SUB  3. MUL  4. DIV "
echo -n " u r choice : "; read cho

case "$cho" in
1 ) let "result=a+b"; echo " SUM (A+B) = $result ";;
2 ) let "result=a-b"; echo " SUB (A-B) = $result ";;
3 ) let "result=a*b";echo " MUL (A*B) = $result ";;
4 ) let "result=a/b";echo " SUM (A/B) = $result ";;
* ) echo "WRONG CHOICE"
esac
echo
```



```
[train1@BSLBLRGEWKS061 dinesh]$ ./case_1.bash


 Ent Value A = 20
 Ent Value B = 10

      MAIN MENU
 1. ADD  2. SUB  3. MUL  4. DIV
 u r choice : 3
 MUL (A*B) = 200

[train1@BSLBLRGEWKS061 dinesh]$ ./case_1.bash


 Ent Value A = 40
 Ent Value B = 10

      MAIN MENU
 1. ADD  2. SUB  3. MUL  4. DIV
 u r choice : 5
WRONG CHOICE

[train1@BSLBLRGEWKS061 dinesh]$ ./case_1.bash


 Ent Value A = 50
 Ent Value B = 25

      MAIN MENU
 1. ADD  2. SUB  3. MUL  4. DIV
 u r choice : 4
 SUM (A/B) = 2

[train1@BSLBLRGEWKS061 dinesh]$ _
```

Example 2: Performing Menu Program till user wants

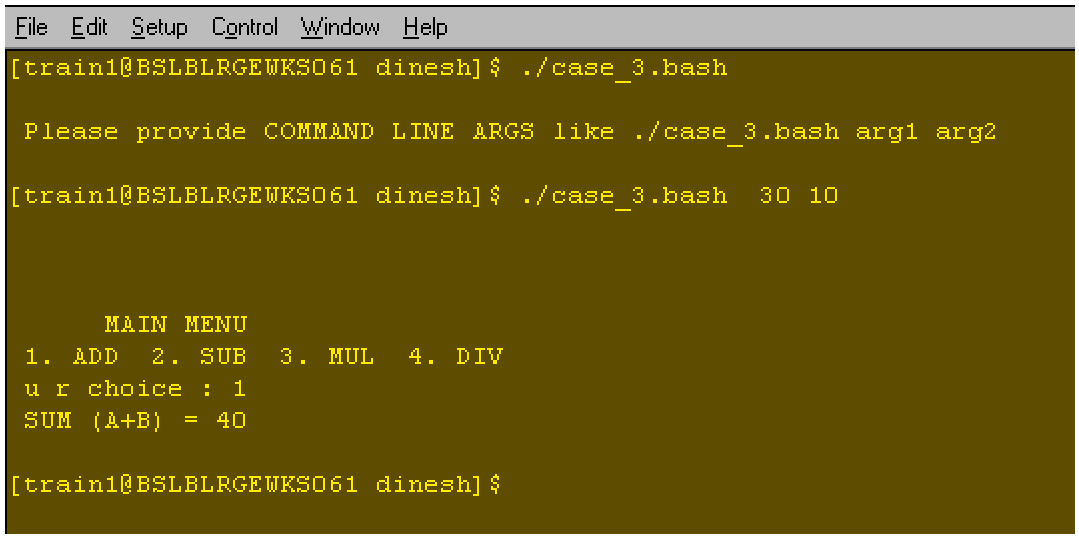
```
# This script to show how to use case statement

    ch="y"
    echo
    echo

    while [ "$ch" = y ]
    do

    echo -n " Ent Value A = "; read a
    echo -n " Ent Value B = "; read b

    echo -e "\n      MAIN MENU     "
    echo " 1. ADD  2. SUB  3. MUL  4. DIV "
    echo -n " u r choice : "; read cho

    case "$cho" in
    1 ) let "result=a+b"; echo " SUM (A+B) = $result "; (( ch="n" )); echo; echo -n "CONT (y/n): "; read ch;;
    2 ) let "result=a-b"; echo " SUB (A-B) = $result "; (( ch="n" )); echo; echo -n "CONT (y/n): "; read ch;;
    3 ) let "result=a*b";echo " MUL (A*B) = $result "; (( ch="n" )); echo; echo -n "CONT (y/n): "; read ch;;
    4 ) let "result=a/b";echo " SUM (A/B) = $result "; (( ch="n" )); echo; echo -n "CONT (y/n): "; read ch;;
    * ) echo "WRONG CHOICE"
    esac
    echo

    done
```



```
[train1@BSLBLRGEWKS061 dinesh]$ ./case_2.bash


 Ent Value A = 20
 Ent Value B = 30

      MAIN MENU
 1. ADD  2. SUB  3. MUL  4. DIV
 u r choice : 1
 SUM (A+B) = 50

CONT (y/n): y

 Ent Value A = 10
 Ent Value B = 10

      MAIN MENU
 1. ADD  2. SUB  3. MUL  4. DIV
 u r choice : 3
 MUL (A*B) = 100

CONT (y/n): n

[train1@BSLBLRGEWKS061 dinesh]$
```

Example3: Menu Program with command line arguments

```
# This script to show how to use case statement
# with command Line Argument


    if [ $# -ne 0 ]
    then

    echo
    echo
    echo -e "\n        MAIN MENU        "
    echo " 1. ADD  2. SUB  3. MUL  4. DIV "
    echo -n " u r choice : "; read cho

    case "$cho" in
    1 ) let "result=$1+$2"; echo " SUM (A+B) = $result ";;
    2 ) let "result=$1-$2"; echo " SUB (A-B) = $result ";;
    3 ) let "result=$1*$2";echo " MUL (A*B) = $result ";;
    4 ) let "result=$1/$2";echo " SUM (A/B) = $result ";;
    * ) echo "WRONG CHOICE"
    esac
    echo

    else
        echo
        echo " Please provide COMMAND LINE ARGS like ./case_3.bash arg1 arg2 "
        echo
    fi
```



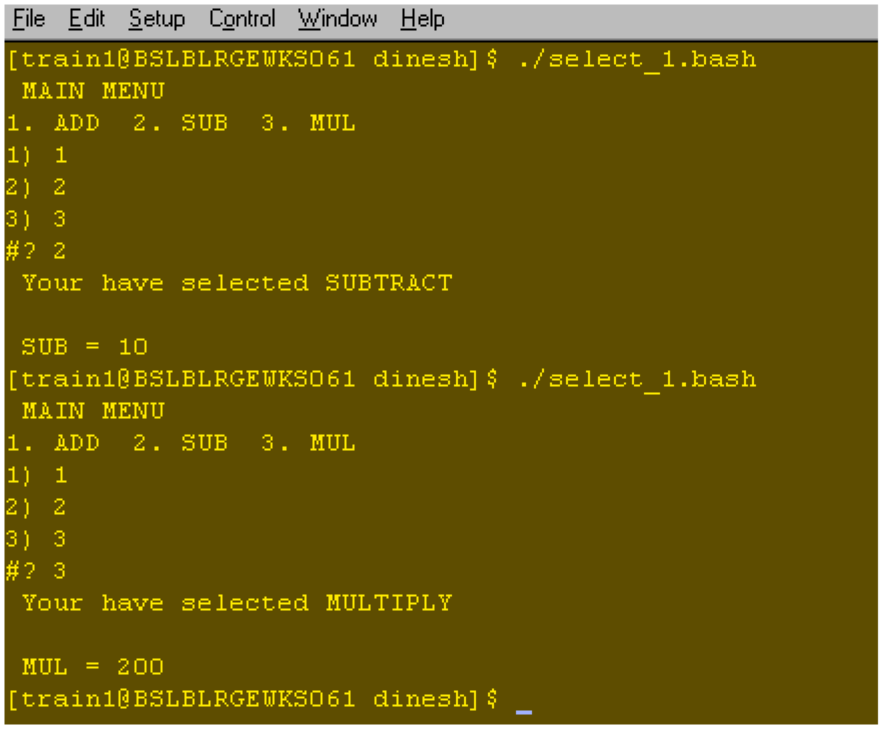
```
[train1@BSLBLRGEWKS061 dinesh]$ ./case_3.bash

 Please provide COMMAND LINE ARGS like ./case_3.bash arg1 arg2

[train1@BSLBLRGEWKS061 dinesh]$ ./case_3.bash  30 10



      MAIN MENU
 1. ADD  2. SUB  3. MUL  4. DIV
 u r choice : 1
 SUM (A+B)  = 40

[train1@BSLBLRGEWKS061 dinesh]$
```

Select:

Select statement is adopted from Korn shell. This is also used to build menus.

Select variable [list]
do
        Command
        Break
done

```
# This script to show how to use SELECT statement

a=20
b=10
choice()
{
        case $1 in
        1 ) echo " Your have selected ADDITION"; let "res=a+b"; echo -e " \n SUM = $res ";;
        2 ) echo " Your have selected SUBTRACT";let "res=a-b"; echo -e " \n SUB = $res ";;
        3 ) echo " Your have selected MULTIPLY";let "res=a*b"; echo -e " \n MUL = $res "
        esac
}

echo " MAIN MENU "
echo "1. ADD  2. SUB  3. MUL"

select maths in 1 2 3
do
break           # if no break statement here continuously it will ask for choice
done

choice $maths
```



```
[train1@BSLBLRGEWKS061 dinesh]$ ./select_1.bash
 MAIN MENU
1. ADD  2. SUB  3. MUL
1) 1
2) 2
3) 3
#? 2
 Your have selected SUBTRACT

 SUB = 10
[train1@BSLBLRGEWKS061 dinesh]$ ./select_1.bash
 MAIN MENU
1. ADD  2. SUB  3. MUL
1) 1
2) 2
3) 3
#? 3
 Your have selected MULTIPLY

 MUL = 200
[train1@BSLBLRGEWKS061 dinesh]$ _
```

Chapter 11

Functions

A function is a subroutine, which when executed implements some set of operations or task.

Syntax:

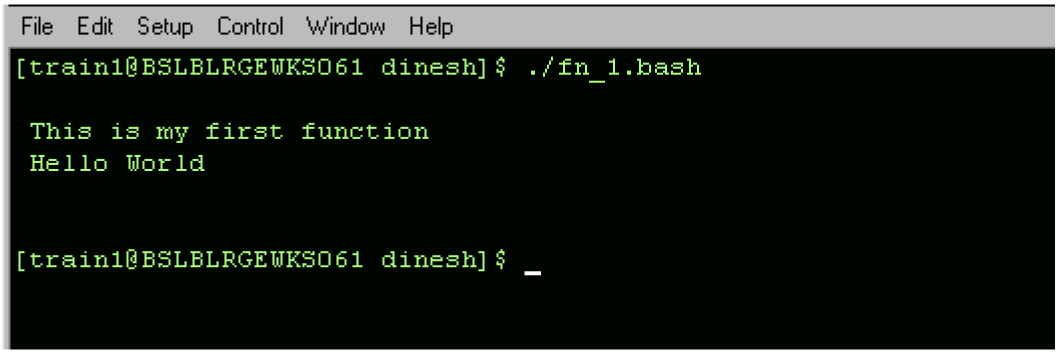
```
function func_name ()
{
   Command
}
```

(Or)

```
func_name ()
{
   Command
}
```

Example 1: Simple Function

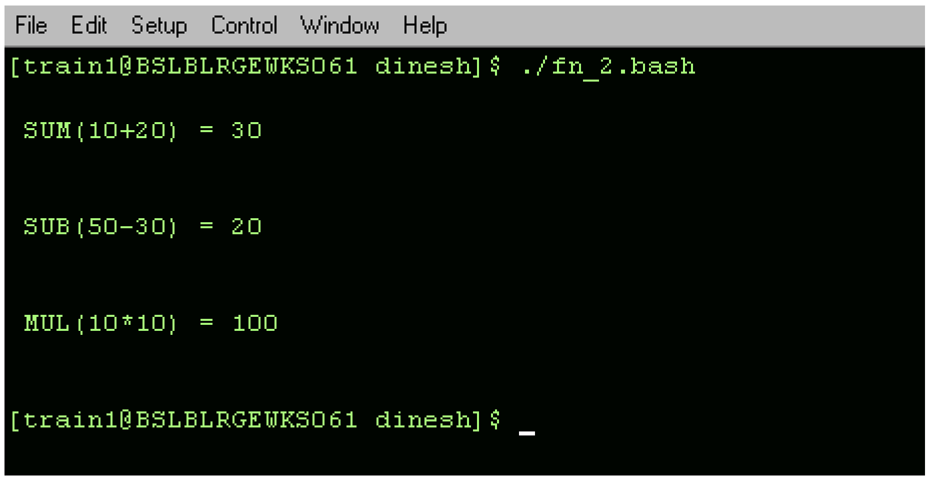
```
# This script shows how to write a simple function

function f1()
{
	echo -e " \n This is my first function"

}

f2()
{
	echo -e " Hello World \n "
}

f1	# Calling a function
f2

echo
```



```
[train1@BSLBLRGEWKS061 dinesh]$ ./fn_1.bash

 This is my first function
 Hello World


[train1@BSLBLRGEWKS061 dinesh]$ _
```

Example 2: Passing argument to a Function

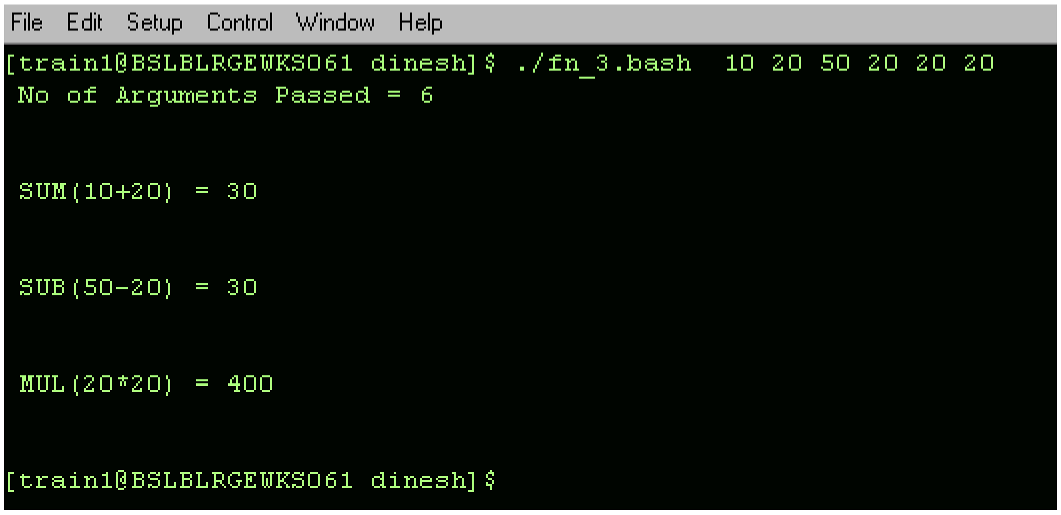
```
# This script shows how to pass argument to a function inside shell script

add()
{
    let "res=$1+$2"
    echo -e " \n SUM($1+$2) = $res \n"
}

sub()
{
    let "res=$1-$2"
    echo -e " \n SUB($1-$2) = $res \n"
}

mul()
{
    let "res=$1*$2"
    echo -e " \n MUL($1*$2) = $res \n"
}

add 10 20              # parameter values can be reterieved by using POSITIONAL PARAMETERS

sub 50 30

mul 10 10

echo
```



```
[train1@BSLBLRGEWKS061 dinesh]$ ./fn_2.bash

 SUM(10+20) = 30


 SUB(50-30) = 20


 MUL(10*10) = 100


[train1@BSLBLRGEWKS061 dinesh]$ _
```

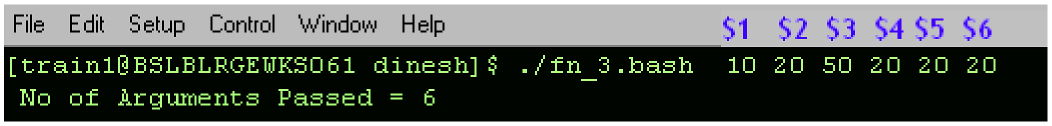
Example 3: Passing argument to a Function from Command Line

```
# THis script to show how to pass argument from command line to a function
    add()
    {
            let "res=$1+$2"
            echo -e " \n SUM($1+$2) = $res \n"
    }

    sub()
    {
            let "res=$1-$2"
            echo -e " \n SUB($1-$2) = $res \n"
    }

    mul()
    {
            let "res=$1*$2"
            echo -e " \n MUL($1*$2) = $res \n"
    }

    echo -e " No of Arguments Passed = $# \n"

    add $1 $2

    sub $3 $4

    mul $5 $6

    echo
```



```
[train1@BSLBLRGEWKS061 dinesh]$ ./fn_3.bash  10 20 50 20 20 20
 No of Arguments Passed = 6


 SUM(10+20) = 30


 SUB(50-20) = 30


 MUL(20*20) = 400


[train1@BSLBLRGEWKS061 dinesh]$
```

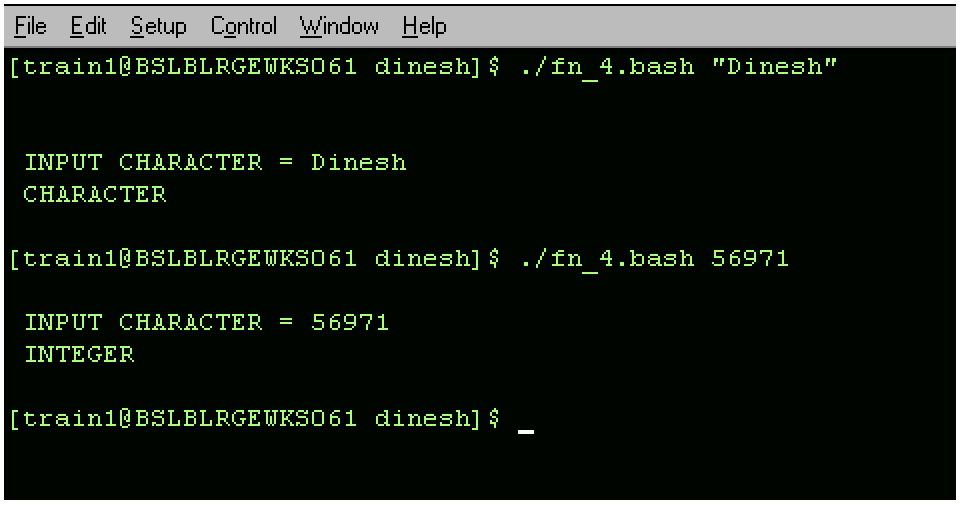
Note:



```
[train1@BSLBLRGEWKS061 dinesh]$ ./fn_3.bash  10 20 50 20 20 20
 No of Arguments Passed = 6
```

Example 4: Function calling another function

```
# This script to show how to call function inside another function

echo
echo " INPUT CHARACTER = $1 "

check_arg()
{
    if [ "$#" -lt 1 ]
    then
        echo -e " Please Enter ARGUMENTS !!!!! \n"

    fi

}


check_input()
{
    check_arg $1                    # calling a function

    case $1 in

        *[0-9]* ) echo " INTEGER ";;
        *[a-zA-Z]* ) echo " CHARACTER ";;
        * ) echo " UNKNOWN INPUT !!!!!"

    esac
}

check_input $1      # Main Function
echo
```

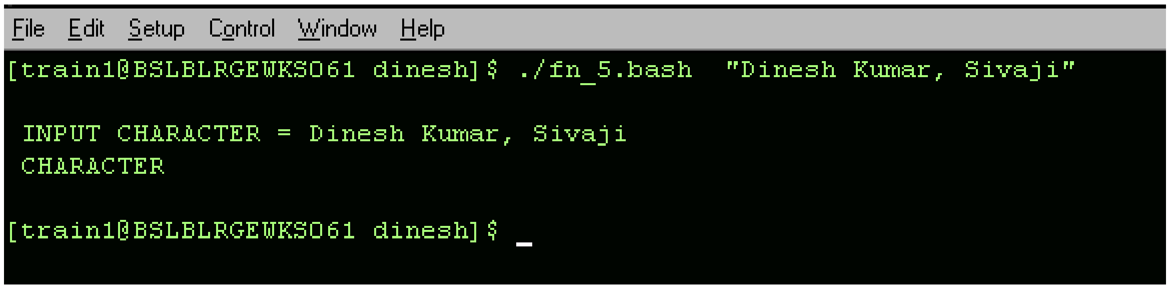


```
[train1@BSLBLRGEWKS061 dinesh]$ ./fn_4.bash "Dinesh"


 INPUT CHARACTER = Dinesh
 CHARACTER

[train1@BSLBLRGEWKS061 dinesh]$ ./fn_4.bash 56971

 INPUT CHARACTER = 56971
 INTEGER

[train1@BSLBLRGEWKS061 dinesh]$ _
```

Example 5: Defining Function inside another function

We are going to modify **example 4** to explain this concept.

```
# This script to show how to call function inside another function
# LOCAL function cannot be used anywhere in SHELL script

echo
echo " INPUT CHARACTER = $1 "

check_input()
{
    check_arg()         # Function Defined inside another function
    {                   # This is a LOCAL function & can be used within check_input only

        if [ "$#" -lt 1 ]
        then
            echo -e " Please Enter ARGUMENTS !!!!! \n"
        fi
    }

    case $1 in

        *[0-9]* ) echo " INTEGER ";;
        *[a-zA-Z]* ) echo " CHARACTER ";;
        * ) echo " UNKNOWN INPUT !!!!!"

    esac
}

check_input $1     # Main Function
echo
```

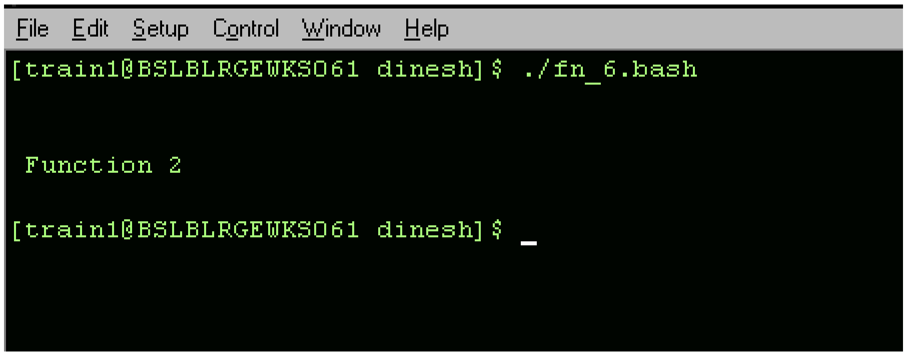


```
[train1@BSLBLRGEWKS061 dinesh]$ ./fn_5.bash  "Dinesh Kumar, Sivaji"

 INPUT CHARACTER = Dinesh Kumar, Sivaji
 CHARACTER

[train1@BSLBLRGEWKS061 dinesh]$ _
```

Example 6: Multiple Functions with same name inside shell script

When **multiple functions** with same name is defined inside a shell script only **second** function i.e. **lastly** declared function will be active.

```
# THis script shows what happens when multiple functions with same name
#+ is declared inside a shell script

echo

function dinesh()
{
    echo -e " \n Function 1 "
}

dinesh()
{
    echo -e " \n Function 2 "
}


main()                        # main () calling function dinesh
{
  dinesh
}

main                        # Calling Main()

echo
```



```
[train1@BSLBLRGEWKS061 dinesh]$ ./fn_6.bash


 Function 2

[train1@BSLBLRGEWKS061 dinesh]$ _
```

Thank You!