Course	EDITORS AND SHELL SCRIPTING	
Module Day 2		**3 Hours**
	At the end of this module you will be able to know: ▪ Editors and their uses. ▪ Vi editors and modes. ▪ Commands in vi editor. ▪ Customizing vi editor. ▪ Shell and its applications. ▪ Writing and running a shell script. ▪ Commands in shell script. ▪ Basic shell scripting. ▪ Advanced shell scripting.	

Session Plan	
1	Editors and their uses, vi editor modes and commands, customizing vi editor - environment variables: Path, PS1, operations in editor.
2	What is a shell, its significance, pipes and redirections, creating and executing a script, commenting and documenting the scripts
3	shell syntax :- Variables: strings, numbers, environment and parameter, Conditions: shell Booleans, Program Control: if, elif, for, while, until, case ,Lists

Module Overview

- ✓ Introduction to editors
- ✓ Vi text editor and modes
- ✓ Commands in vi editor
- ✓ Environment variables and usage.
- ✓ Shell and its applications
- ✓ Writing and executing a shell script.
- ✓ Shell script syntax of variables
- ✓ Program control commands in shell.

Introduction to editors

If I were to choose one of the main reasons why people use PCs, I would definitely say for writing. With a computer and a word processing program, cross outs, white out and crumpled up paper has disappeared forever. Linux is just as well suited for word processing as any other operating system. There are several excellent word processing programs for Linux like AbiWord, KWord, part of the KOffice suite and the OpenOffice.org suite's word processor. First, we should talk about the terminal mode text editors that are available for Linux.

A text editor is just like a word processor without a lot of features. All operating systems come with a basic text editor. Linux comes with several. The main use of a text editor is for writing something in plain text with no formatting so that another program can read it. Based on the information it gets from that file, the program will run one way or another.

Vi text editor

The most popular text editor for Linux is called 'vi'. This is a program that comes from UNIX. There is a more recent version called 'vim' which means 'vi improved'.

Vi was the very first full-screen text editor. In fact, Vi is abbreviation for visual editor.

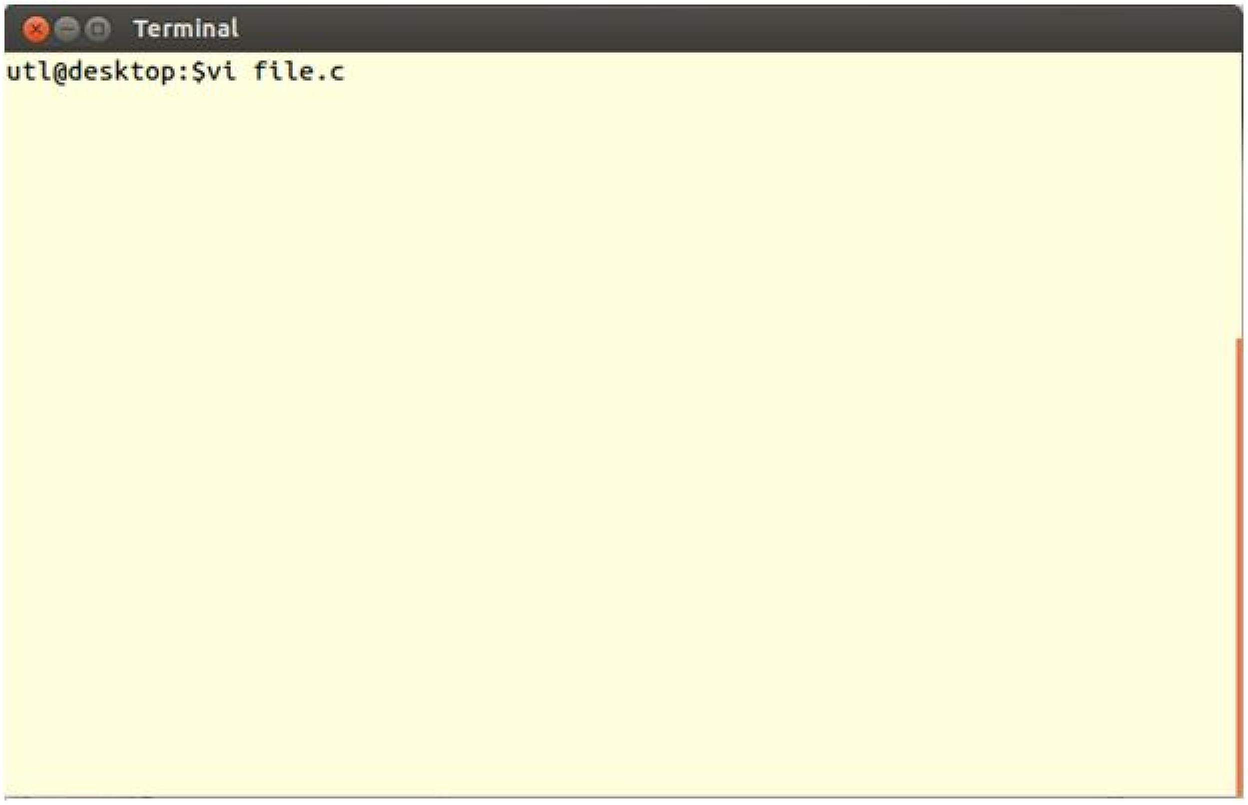
Vi is generally considered the defacto standard in UNIX editors because:

- It’s usually available on any UNIX system. Once you have learned how to use it, there is no learning curve when you’re working on another UNIX system, which allows for faster recovery or adjustments of files.

- Vi implementations are very similar across the board (from Linux, to Mac OS X, to Sun Solaris and so on).

- It requires very few resources.

To start, create a new file by issuing vi command on the prompt as shown below:



```
utl@desktop:$vi file.c
```

There are several ways to start ("open an instance of") vi, all of which you type in a console window:

Command	**Description**	**Results**
Vi	Start vi without any filename	vi starts with an empty palette. You must save to a new file before exiting.
vi filename	1. Use an existing file name as the argument. 2. Use the new filename as the argument(file doesn't exist until saved in vi)	1. The exiting file opens in vi. When you save it, the file is updated with the changes you make. 2. When you save the file, you create a new file with the file name specified in the argument.

vi's Modes:

Before you start working with vi, you need to understand one very critical concept.
vi has two modes of operation:

1. Command mode—Enables you to perform administrative tasks such as saving files, executing commands, moving the cursor, cutting (yanking) and pasting lines

or words, and finding and replacing. In this mode, anything that's typed—even a single character—is interpreted as a command.

2. Insert mode—Enables you to insert text into the file. Everything that's typed in this mode is interpreted as input.

Vi always starts in command mode. To enter text, you must be in insert mode, Simply type **I**. To get out of insert mode, press the **Esc** key, which will put you back into command mode.

Table below shows several options to save and quit:

Command	Action
:w	Saves(writes) File
:q	Quits vi
:wq	Saves file and quits vi
:q!	Forces a quit (allows you to quit on an unsaved file)
ZZ	Saves and quits(same as :wq)

At this point, you can use vi's built-in commands to edit your file.

Working with files

Vim command	**Action**
:e filename	Open a new file. You can use the Tab key for automatic file name completion, just like at the shell command prompt.
:w filename	Save changes to a file. If you don't specify a file name, Vim saves as the file name you were editing. For saving the file under a different name, specify the file name
:q	Quit vim. If you have any unsaved changes, Vim refuses to exit.
:q!	Exit Vim without saving changes.
:wq	Write the file and exit.
:x	Almost the same as :wq, write the file and exit if you've made changes to the file. If you haven't made any changes to the file, Vim exits without writing the file.

Moving around in the file

These Vim commands and keys work both in command mode and visual mode.

Vim command	Action
k or Up Arrow	Move the cursor up one line.
j or Down Arrow	Down one line.
h or Left Arrow	Left one character.
l or Right Arrow	Right one character.
e	To the end of a word.
E	To the end of a whitespace-delimited word.
b	To the beginning of a word.
B	To the beginning of a whitespace-delimited word.
0	To the beginning of a line.
^	To the first non-whitespace character of a line.
$	To the end of a line.
H	To the first line of the screen.
M	To the middle line of the screen.
L	To the last line of the screen.
:n	Jump to line number n. For example, to jump to line 42, you'd type :42

Inserting and overwriting text

Vim command	Action
i	Insert before cursor.
I	Insert to the start of the current line.
a	Append after cursor.
A	Append to the end of the current line.
o	Open a new line below and insert.
O	Open a new line above and insert.
C	Change the rest of the current line.
r	Overwrite one character. After overwriting the single character, go back to command mode.
R	Enter insert mode but replace characters rather than inserting.
The ESC key	Exit insert/overwrite mode and go back to command mode.

Deleting text

Vim command	Action
x	Delete characters under the cursor.
X	Delete characters before the cursor.
dd or :d	Delete the current line.

Entering visual mode

Vim command	Action
v	Start highlighting characters. Use the normal movement keys and commands to select text for highlighting.
V	Start highlighting lines.
The ESC key	Exit visual mode and return to command mode.

Editing blocks of text

Note: the Vim commands marked with (V) work in visual mode, when you've selected some text. The other commands work in the command mode, when you haven't selected any text.

Vim command	Action
~	Change the case of characters. This works both in visual and command mode. In visual mode, change the case of highlighted characters. In command mode, change the case of the character under cursor.
> (V)	Shift right (indent).
< (V)	Shift left (de-indent).
c (V)	Change the highlighted text.
y (V)	Yank the highlighted text. In Windows terms, "copy the selected text to clipboard."
d (V)	Delete the highlighted text. In Windows terms, "cut the selected text to clipboard."
yy or :y or Y	Yank the current line. You don't need to highlight it first.
dd or :d	Delete the current line. Again, you don't need to highlight it first.
p	Put the text you yanked or deleted. In Windows terms, "paste the contents of the clipboard". Put characters after the cursor. Put lines below the current line.
P	Put characters before the cursor. Put lines above the current line.

Undo and redo

Vim command	Action
u	Undo the last action.
U	Undo all the latest changes that were made to the current line.
Ctrl + r	Redo.

Search commands

Vim command	**Action**
/pattern	Search the file for pattern.
n	Scan for next search match in the same direction.
N	Scan for next search match but opposite direction.

Replace

Vim command	**Action**
:rs/foo/bar/a	Substitute foo with bar. r determines the range and a determines the arguments.

The range (r) can be:

nothing	Work on current line only.
number	Work on the line whose number you give.
%	The whole file.

Arguments (a) can be:

g	Replace all occurrences in the line. Without this, Vim replaces only the first occurrences in each line.
i	Ignore case for the search pattern.
I	Don't ignore case.
c	Confirm each substitution. You can type y to substitute this match, n to skip this match, a to substitute this and all the remaining matches ("Yes to all"), and q to quit substitution.

Examples:

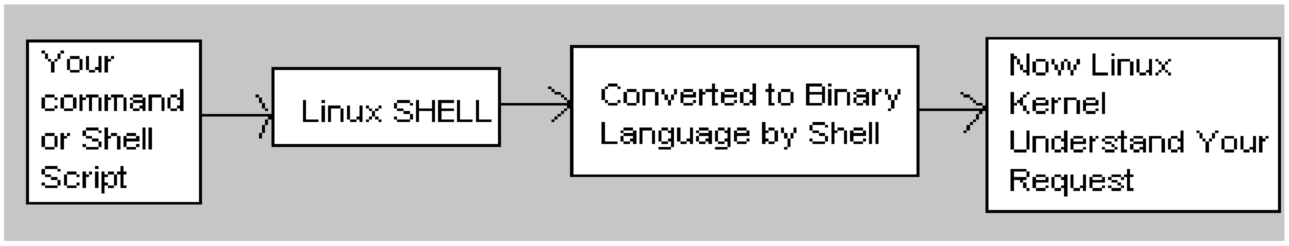
:452s/foo/bar/	Replace the first occurrence of the word foo with bar on line number 452.
:s/foo/bar/g	Replace every occurrence of the word foo with bar on current line.
:%s/foo/bar/g	Replace every occurrence of the word foo with bar in the whole file.
:%s/foo/bar/gi	The same as above but ignore the case of the pattern you want to substitute. This replaces foo, FOO, Foo, and so on.
:%s/foo/bar/gc	Confirm every substitution.
:%s/foo/bar/c	For each line on the file, replace the first occurrence of foo with bar and confirm every substitution

Introduction to Shell

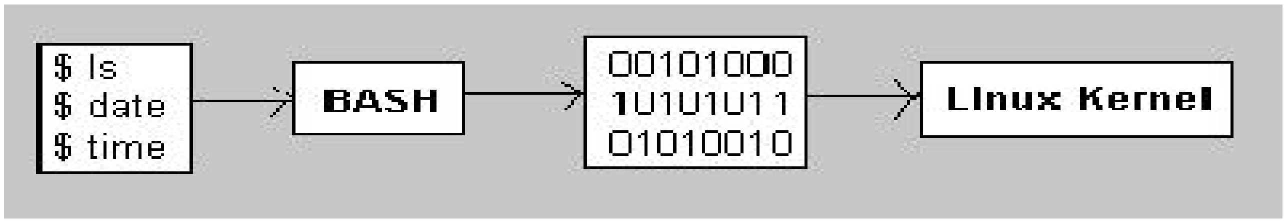
Computer understand the language of 0's and 1's called binary language, In early days of computing, instruction are provided using binary language, which is difficult for all of us, to read and write. So in O/s there is special program called Shell.

Shell accepts your instruction or commands in English and translates it into computers native binary language.

This is what Shell Does for US


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

You type your command and shell converts 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.

Several shells are available for Linux including:

- o **BASH** (Bourne-Again SHell) - Most common shell in Linux. It's Open Source.
- o **CSH** (C SHell) - The C shell's syntax and usage are very similar to the C programming language.
- o **KSH** (Korn SHell) - Created by David Korn at AT & T Bell Labs. The Korn Shell also was the base for the POSIX Shell standard specifications.
- o **TCSH** - It is an enhanced but completely compatible version of the Berkeley UNIX C shell (CSH).

NOTE: To find your shell type following command
$ echo $SHELL

Pipes and redirection

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

There are three main redirection symbols >, >>, <

(1) > (Redirector Symbol)

```
Syntax: Linux-command > filename
```

To output Linux-commands result to file. If file already exist, it will be overwritten else new file is created.

For e.g. to send output of ls command give **`$ ls > myfiles`**
Now if myfiles file exist in your current directory it will be overwritten without any type of warning.

(2) >> (Redirector Symbol)

```
Syntax: Linux-command >> filename
```

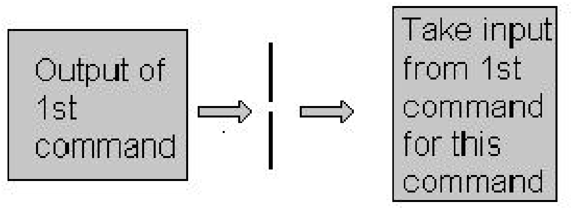
To output Linux-commands result to END of file. If file exist, it will be opened and new information / data will be written to END of file, without losing previous information/data, and if file is not exist, then new file is created.
For e.g. To send output of date command to already exist file give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

(3) < (Redirector Symbol)

```
Syntax: Linux-command < filename
```

This symbol takes input to a linux command from file instead of keyboard.
For e.g. To take input for cat command give **$ cat < myfiles**

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



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

```
Syntax: command1 | command2
```

Examples:

$ ls | more: Here the output of ls command is given as input to more command So that output is printed one screen full page at a time.

$ who | sort: Here output of who command is given as input to sort command So that it will print sorted list of users.

$ who | wc –l: Here output of who command is given as input to wc command So that it will number of user who logon to system.

$ ls -l | wc –l: Here output of ls command is given as input to wc command So that it will print number of files in current directory.

$ who | grep admin: Here output of who command is given as input to grep command So that it will print if particular user name if he is logon or nothing is printed (To see for particular user logon).

Filter
If a Linux command accepts its input from the standard input and produces its output on standard output is known as a filter. A filter performs some kind of process on the input and gives output.

For e.g.. Suppose we have file called **hotel.txt** with 100 lines data, And from 'hotel.txt' we would like to print contains from line number 20 to line number 30 and store this result to file called **hlist** then give command

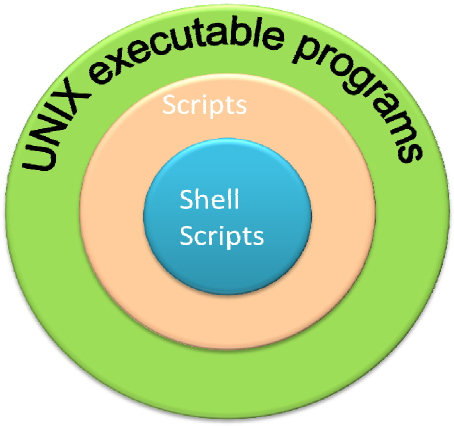
$ tail +20 < hotel.txt | head -n30 >hlist

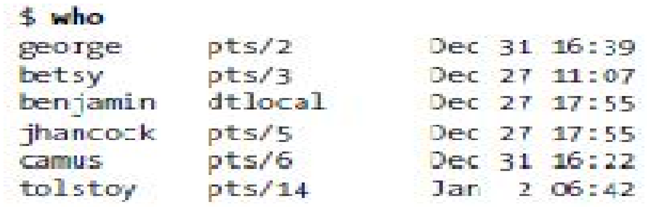
Here head is filter which takes its input from tail command (tail command start selecting from line number 20 of given file i.e. hotel.txt) and passes this lines to input to head, whose output is redirected to 'hlist' file.

Basic Shell Script

The basic concept of a shell script is a list of commands, which are listed in the order of execution.

Shell scripts and functions are both *interpreted*. This means they are not compiled.
Both shell scripts and functions are ASCII text that is read by the shell command interpreter.
When we execute a shell script, or function, a command interpreter goes through the ASCII text line-by-line, loop-by-loop, test-by-test, and executes each statement as each line is reached from the top to the bottom.





```
$ who
george      pts/2       Dec 31 16:39
betsy       pts/3       Dec 27 11:07
benjamin    dtlocal     Dec 27 17:55
jhancock    pts/5       Dec 27 17:55
camus       pts/6       Dec 31 16:22
tolstoy     pts/14      Jan  2 06:42
```

Why shell??
Because the shell is universal among UNIX systems, and the language is standardized by POSIX. Thus, the reasons to use a shell script are:

Simplicity:
The shell is a high-level language; you can express complex operations clearly and simply using it.

Portability:
By using just POSIX-specified features, you have a good chance of being able to move your script, unchanged, to different kinds of systems.
Ease of development:
You can often write a powerful, useful script in little time.

Shell Script Application

- Combine lengthy and repetitive sequences of commands into a single, simple command.
- Generalize a sequence of operations on one set of data, into a procedure that can be applied to any similar set of data.
- Create new commands using combinations of utilities in ways the original authors never thought of.
- Simple shell scripts might be written as shell aliases, but the script can be made available to all users and all processes. Shell aliases apply only to the current shell.
- Wrap programs over which you have no control inside an environment that you can control.
- Create customized datasets on the fly, and call applications (e.g. matlab, sas, idl, gnuplot) to work on them, or create customized application commands/procedures.
- Rapid prototyping (but avoid letting prototypes become production)

Typical uses

- System boot scripts (/etc/init.d)
- System administrators, for automating many aspects of computer maintenance, user accounts creation etc.
- Application package installation tools
- Application startup scripts, especially unattended applications (e.g. started from **cron** or **at**).
- Any user needing to automate the process of setting up and running commercial applications, or their own code.

Variables

Sometimes to process our data/information, it must be kept in computers RAM memory. RAM memory is divided into small locations, and each location had unique number called memory location/address, which is used to hold our data. Programmer can give a unique name to this memory location/address called memory variable or variable (Its a named storage location that may take different values, but only one at a time).

In Linux, there are two types of variable
1) Environment variables - Created and maintained by Linux itself. This type of variable defined in CAPITAL LETTERS. These are also refered as system variables.
2) User defined variables (UDV) - Created and maintained by user. This type of variable defined in lower LETTERS.

Environment variables list

Variable	Description
HISTFILE	The name of the file in which command history is saved
HISTFILESIZE	The maximum number of lines contained in the history file
HOSTNAME	The system's host name
PS1	Your default (first) shell prompt
USER	Current logged in user's name.
PATH	Colon separated list of directories to search for binaries.
DISPLAY	Network name of the X11 display to connect to, if available.
SHELL	The current shell.
TERM	The name of the user's terminal. Used to determine the capabilities of the terminal.
TERMCAP	Database entry of the terminal escapes codes to perform various terminal functions.
OSTYPE	Type of operating system.
MACHTYPE	The CPU architecture that the system is running on.

You can print any of the above variables contain as follows
$ echo $USERNAME
$ echo $HOME
Caution: Do not modify System variable this can some time create problems.

User defined variables (UDV)
To define UDV use following syntax

```
Syntax: variablename=value
```

Here 'value' is assigned to given 'variablename' and Value must be on right side = sign
For e.g.
$ no=10 # this is ok
$ 10=no # Error, NOT Ok, Value must be on right side of = sign.
To define variable called 'vech' having value Bus
$ vech=Bus
To define variable called n having value 10
$ n=10

Rules for naming a variable

1. Variable name must begin with Alphanumeric character or underscore character (_), followed by one or more Alphanumeric character.
   For e.g. Valid shell variable are as follows
   HOME , SYSTEM_VERSION , vech.
2. Don't put spaces on either side of the equal sign when assigning value to variable.
   For e.g. In following variable declaration there will be no error
   ```
   $ no=10
   ```
   But here there will be problem for following
   ```
   $ no =10
   $ no= 10
   $ no = 10
   ```
3. Variables are case-sensitive, just like filename in Linux.
   For e.g.
   ```
   $ no=10
   $ No=11
   $ NO=20
   $ nO=2
   ```
   Above all are different variable name, so to print value 20 we have to use **$ echo** `$NO` and Not any of the following
   ```
   $ echo $no   # will print 10 but not 20
   $ echo $No   # will print 11 but not 20
   $ echo $nO   # will print 2 but not 20
   ```
4. You can define NULL variable as follows (NULL variable is variable which has no value at the time of definition)
   For e.g.
   ```
   $ vech=
   $ vech=""
   ```
   Try to print it's value `$ echo $vech` , Here nothing will be shown because variable has no value i.e. NULL variable.

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

Comments in shell script

Take look at the following shell script:

```
#!/bin/bash

# A Simple Shell Script To Get Linux Network Information

# Vivek Gite - 30/Aug/2009

echo "Current date : $(date) @ $(hostname)"

echo "Network configuration"

/sbin/ifconfig
```

The first line is called a "bang" line. The following are the next two lines of the program:

```
# A Simple Shell Script To Get Linux Network Information
# Vivek Gite - 30/Aug/2009
```

- A word or line beginning with # causes that word and all remaining characters on that line to be ignored.
- These lines aren't statements for the bash to execute. In fact, the bash totally ignores them.
- These notes are called comments.
- It is nothing but explanatory text about script.
- It makes source code easier to understand.
- These notes are for humans and other sys admins.
- It helps other sys admins to understand your code, logic and it helps them to modify the script you wrote.

Multiple Line Comment

You can use HERE DOCUMENT feature as follows to create multiple line comment:

```
#!/bin/bash
echo "Adding new users to LDAP Server..."
<<COMMENT1
    Master LDAP server : dir1.nixcraft.net.in
    Add user to master and it will get sync to backup server too
    Profile and active directory hooks are below
COMMENT1
echo "Searching for user..."
```

Writing the first shell script

Now we write our first script that will execute the commands pwd, date and cal on screen. To write shell script you can use in of the Linux's text editor such as vi or mcedit or even you can use vim editor.

```
$ vim myscript

pwd

date

cal
```

Press :wq to save and quit. Now our script is ready.

Running a Shell Script

Because of security of files, in Linux, the creator of Shell Script does not get execution permission by default.
So if we wish to run shell script we have to do two things as follows.

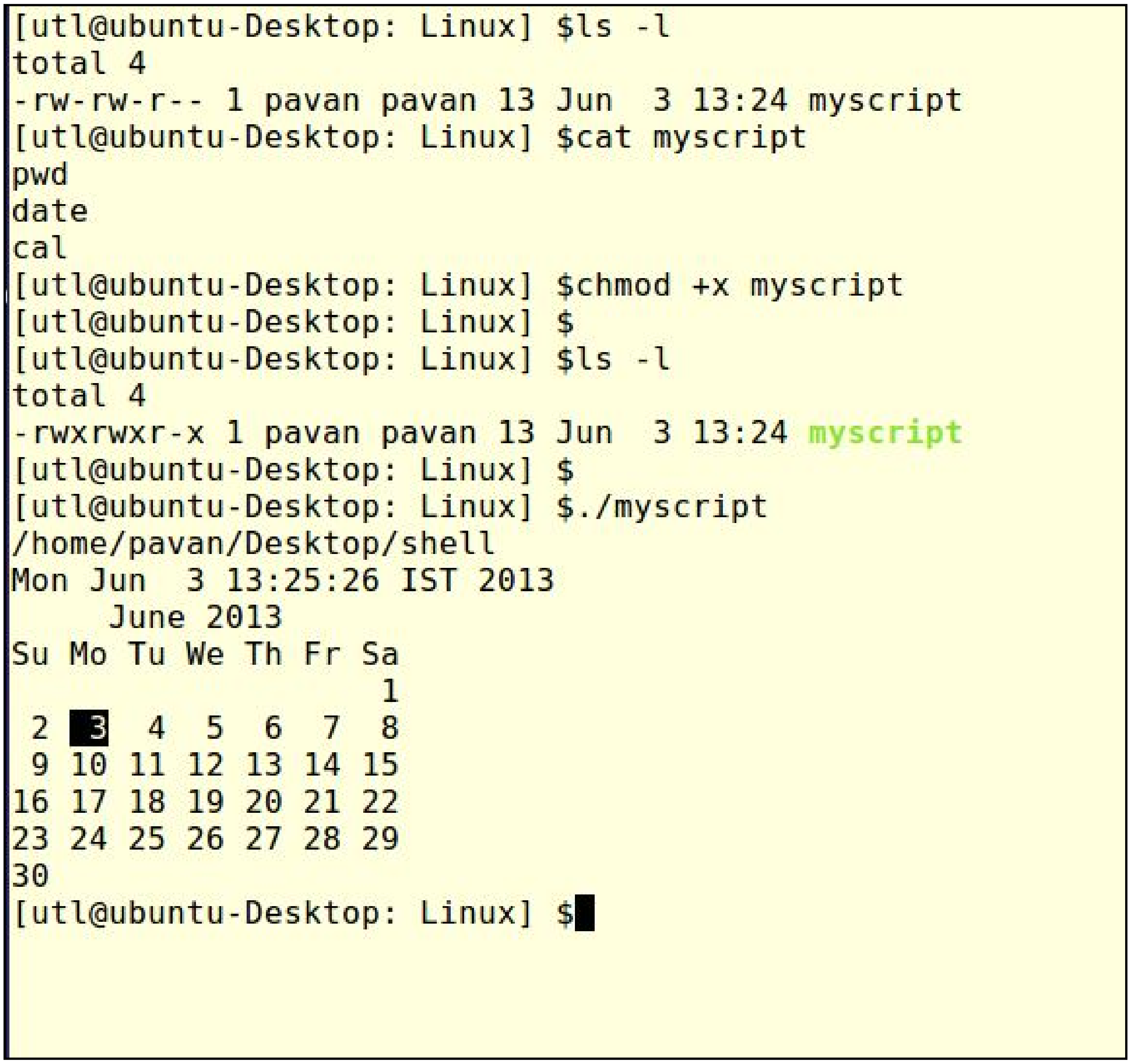
Use `chmod` command as follows to give execution permission to our script

```
Syntax: chmod +x shell-script-name
Syntax: chmod 777 shell-script-name
```

Run our script as: `./your-shell-program-name`

For e.g.   `$ ./first`

Here '.'(dot) is command, and used in conjunction with shell script. The dot(.) indicates to current shell that the command following the dot(.) has to be executed in the same shell i.e. without the loading of another shell in memory.



```
[utl@ubuntu-Desktop: Linux] $ls -l
total 4
-rw-rw-r-- 1 pavan pavan 13 Jun  3 13:24 myscript
[utl@ubuntu-Desktop: Linux] $cat myscript
pwd
date
cal
[utl@ubuntu-Desktop: Linux] $chmod +x myscript
[utl@ubuntu-Desktop: Linux] $
[utl@ubuntu-Desktop: Linux] $ls -l
total 4
-rwxrwxr-x 1 pavan pavan 13 Jun  3 13:24 myscript
[utl@ubuntu-Desktop: Linux] $
[utl@ubuntu-Desktop: Linux] $./myscript
/home/pavan/Desktop/shell
Mon Jun  3 13:25:26 IST 2013
     June 2013
Su Mo Tu We Th Fr Sa
                   1
 2  3  4  5  6  7  8
 9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30
[utl@ubuntu-Desktop: Linux] $
```

We can execute the script without execution permissions by using sh (shell) command.

Syntax:

```
sh shellprogramname
sh first
```

Commands related with shell programming

`1. echo [options] [string, variables...]`

Displays text or variables value on screen.

Options

-n	Do not output the trailing new line.
-e	Enable interpretation of the following backslash escaped characters in the strings
\a	alert (bell)
\b	backspace
\c	suppress trailing new line
\n	new line
\r	carriage return
\t	horizontal tab
\\	backslash

For eg. `$ echo -e "An apple a day keeps away \a\t\tdoctor\n"`

`2. Quotes`

There are three types of quotes

"	Double Quotes
'	Single quotes
`	Back quote

Single quotes

In Unix shell scripting, there are many special characters used by the shell, such as $, [, *, and #. If text in a shell script is surrounded by single quotes, then *every character* in that text is considered to be a normal, non-special character (except another quote). This includes spaces and even newline characters

For example, the shell command:

```
$ echo 'The total is nearly $750'
```

will cause the following output to appear on the screen:

```
The total is nearly $750
```

Double quotes

Single quotes remove all of the shell's special-character features. Sometimes this is excessive – we may prefer some of the special characters to work, specifically:
$ (for variable substitution, e.g. $PATH), ` (see the next section). Also, we may want the use of certain constructs, like \" or \$.
In these situations we can surround the text with double quotes. Other characters are still treated as special

For example:
```
echo "$LOGNAME made \$1000 in `date +%B`"
```
produces
```
peter made $1000 in November
```

Back Quotes

Unlike single and double quotes, the back quotes have nothing to do with special characters
Any text enclosed in back quotes is treated as a UNIX command, and is executed in its own shell. Any output from the command is substituted into the script line, replacing the quoted text.
For example
```
     list=`who | sort`
     echo $list
```
produces
```
fred tty02 Aug 21 11:01 peter tty01 Aug 22 09:58 tony tty05 Aug
22 10:32
```

Shell Arithmetic

To print sum of two numbers, let's say 6 and 3
```
     $ echo 6 + 3
```
This will print 6 + 3, not the sum 9, to do sum or math operations in shell use `expr`.

Syntax: *expr op1 operator op2*
Where, op1 and op2 are any Integer Number (Number without decimal point) and operator can be

+	Addition
-	Subtraction
/	Division
%	Modular, to find remainder
*	Multiplication

```
$ expr 6 + 3
```

Now It will print sum as 9, But **$ expr 6+3** will not work because space is required between number and operator used to perform arithmetic operations.

Examples:

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

For the last statement note the following points

1) First, before expr keyword we used ` (back quote) sign not the (single quote) sign.
2) Second, expr is also end with ` i.e. back quote.
3) Here expr 6 + 3 is evaluated to 9, then echo command prints 9 as sum
4) Here if you use double quote or single quote, it will NOT work.

For e.g.

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

Command line processing

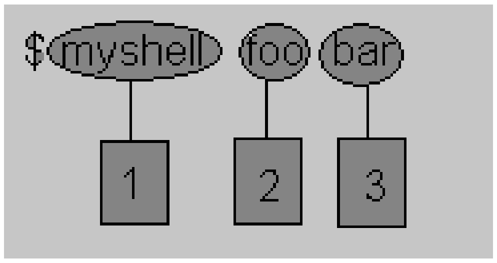
Let's take rm command, which is used to remove file, But which file you want to remove and how you will you tail this to rm command (Even rm command does not ask you name of file that would like to remove).

So what we do is we write as command as follows

$ rm {file-name}

Here rm is command and file-name is file which you would like to remove. This way you tail to rm command which file you would like to remove. So we are doing one way communication with our command by specifying file-name. Also you can pass command line arguments to your script to make it more users friendly. But how we address or access command line argument in our script.

For shell script,
$ myshell foo bar



1 Shell Script name i.e. myshell

2 First command line argument passed to myshell i.e. foo

3 Second command line argument passed to myshell i.e. bar

1 myshell it is $0

2 foo it is $1

3 bar it is $2

Here $# will be 2 (Since foo and bar only two Arguments), Please note At a time such 9 arguments can be used from $0 to $9, You can also refer all of them by using $* (which expand to `$0,$1,$2...$9`) Now try to write following for commands.
Now we will write script to print command ling argument and we will see how to access them.

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

Save the above script by pressing ctrl+d and execute the script.

Exit Status

By default in Linux if particular command is executed, it returns two types of values, if return value is zero (0), command is successful.
If return value is nonzero (>0), command is not successful or some sort of error executing command/shell script. This value is known as Exit Status of that command.

To determine this exit Status we use $? variable of shell.
For e.g.
$ rm unknow1file
It will show error as follows
rm: cannot remove `unkowm1file': No such file or directory

Now if we execute the command **$ echo $?**
it will print nonzero value(>0) to indicate error.

Now give command **$ ls**
$ echo $?
It will print 0 to indicate command is successful.

Try the following commands and not down there exit status
$ expr 1 + 3
$ echo $?
$ echo Welcome
$ echo $?
$ wildwest canwork?
$ echo $?
$ date
$ echo $?

Conditional command execution

The control operators are && (read as AND) and || (read as OR).

AND operator (&&):
```
Syntax: command1 && command2
```
Here command2 is executed if, and only if, command1 returns an exit status of zero.

OR operator (||):
```
Syntax: command1 || command2
```
Here command2 is executed if and only if command1 returns a non-zero exit status.

You can use both as follows:

```
command1 && comamnd2 || command3
```

Here if command1 is executed successfully then shell will run command2 and if command1 is not successful then command3 is executed.
For example:

```
$ rm myfile && echo “File is removed successfully” || echo “File
is not removed”
```

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

Using structured commands

Till now the shell processed each individual command in the shell script in the order it appeared. This works out fine for sequential operations, where you want all of the commands to process in the proper order. However, this isn’t how all programs operate. Many programs require some sort of logic flow control between the commands in the script. This means that the shell executes certain commands given one set of circumstances, but it has the ability to execute other commands given a different set of circumstances. There is a whole class of commands that allows the script to skip over or loop through commands based on conditions of variable values, or the result of other commands. These commands are generally referred to as structured commands.
The structured commands allow you to alter the flow of operation of the program, executing some commands under some conditions, while skipping others under other conditions. There are quite a few structured commands available in the bash shell, so we’ll look at them individually. First we’ll look at the if-then statement.

If then statement

The most basic type of structured command is the if-then statement.
The if-then statement has the following format:

```
if command
     then
     commands
fi
```

If you’re use to using if-then statements in other programming languages, this format may be somewhat confusing. In other programming languages, the object after the if statement is an equation that is evaluated for a TRUE or FALSE value. That’s not how the bash shell if statement works.

The bash shell if statement runs the command defined on the if line. If the exit status of the command is zero, the commands listed under the then section are executed. If the exit status of the command is anything else, the then commands aren’t executed, and the bash shell moves on to the next command in the script.

Here’s a simple example to demonstrate this concept:

```
#!/bin/bash

# testing the if statement

if date

     then echo “Inside if condition”

     echo "it worked"

fi
```

This script uses the date command on the if line. If the command completes successfully, the echo statement should display the text string. When you run this script from the command line, you’ll get the following results:

```
$ ./test1

Sat Sep 29 14:09:24 EDT 2007

Inside if condition

it worked

$
```

The shell executed the date command listed on the if line. Since the exit status was zero, it also executed the echo statement listed in the then section.

If then else statement

In the if-then statement, you only have one option of whether or not a command is successful. If the command returns a non-zero exit status code, the bash shell just moves on to the next command in the script. In this situation, it would be nice to be able to execute an alternate set of commands. That's exactly what the if-then-else statement is for.

The if-then-else statement provides another group of commands in the statement:

```
if command

then

commands

else

commands

fi
```

If the command in the if statement line returns with an exit status code of zero, the commands listed in the then section are executed, just as in a normal if-then statement. If the command in the if statement line returns a non-zero exit status code, the bash shell executes the commands in the else section. Now we can see the demonstration of if else statements

```
     testuser=badtest

     if grep $testuser /etc/passwd

     then

     echo The files for user $testuser are:

     ls -a /home/$testuser/.b*

     else

     echo "The user name $testuser doesn’t exist on this system"

     fi
```

```
$. /test4

The user name badtest doesn’t exist on this system
```

That’s more user-friendly. Just like the then section, the else section can contain multiple commands. The fi statement delineates the end of the else section.

Nesting if’s

Sometimes you must check for several situations in your script code. Instead of having to write separate if-then statements, you can use an alternative version of the else section, called elif.

The elif statement line provides another command to evaluate, similarly to the original if statement line. If the exit status code from the elif command is zero, bash executes the commands in the second then statement section.

The elif continues an else section with another if-then statement:

```
if command1

then

     command set 1

elif command2

then

     command set 2

elif command3

then

     command set 3

elif command4

then

     command set 4

fi
```

Each block of commands is executed depending on which command returns the zero exit status code. Remember, the bash shell will execute the if statements in order, and only the first one that returns a zero exit status will result in the then section being executed.

The test command

So far all you've seen in the if statement line are normal shell commands. You might be wondering if the bash if-then statement has the ability to evaluate any condition other than the exit status code of a command. The answer is no, it can't. However, there's a neat utility available in the bash shell that helps us evaluate other things, using the if-then statement.

The `test` command provides a way to test different conditions in an if-then statement. If the condition listed in the test command evaluates to true, the test command exits with a zero exit status code, making the if-then statement behave in much the same way that if-then statements work in other programming languages. If the condition is false, the test command exits with a one, which causes the if-then statement to fail.

The format of the test command is pretty simple:

```
test condition
```

The *condition* is a series of parameters and values that the test command evaluates. When used in an if-then statement, the test command looks like this:

```
if test condition

then

     commands

fi
```

The bash shell provides an alternative way of declaring the test command in an if-then statement:

```
if [ condition ]

then

commands

fi
```

The square brackets define the condition that's used in the test command. Be careful; you must have a space after the first bracket, and a space before the last bracket or you'll get an error message.

There are three classes of conditions the test command can evaluate:

1. Numeric comparisons
2. String comparisons
3. File comparisons

Numeric comparisons

The most common method for using the test command is to perform a comparison of two numeric values. The numeric test conditions can be used to evaluate both numbers and variables.

Comparison	**Description**
n1 -eq *n2*	Check if *n1* is equal to *n2*.
n1 -ge *n2*	Check if *n1* is greater than or equal to *n2*.
n1 -gt *n2*	Check if *n1* is greater than *n2*.
n1 -le *n2*	Check if *n1* is less than or equal to *n2*.
n1 -lt *n2*	Check if *n1* is less than *n2*.
n1 -ne *n2*	Check if *n1* is not equal to *n2*.

We will see an example script,

```
val1=10
val2=11
if [ $val1 -gt 5 ]
then
     echo "The test value $val1 is greater than 5"
fi
if [ $val1 -eq $val2 ]
then
     echo "The values are equal"
else
     echo "The values are different"
fi
```

The first test condition:

```
if [ $val1 -gt 5 ]
```

tests if the value of the variable val1 is greater than 5.

The second test condition:

```
if [ $val1 -eq $val2 ]
```

tests if the value of the variable val1 is equal to the value of the variable val2.

Run the script and watch the results:

```
$ ./test5
The test value 10 is greater than 5
The values are different
```

String comparison

The test command also allows you to perform comparisons on string values.

Comparison	**Description**
str1 = str2	Check if str1 is the same as string str2.
str1 != str2	Check if str1 is not the same as str2.
str1 < str2	Check if str1 is less than str2.
str1 > str2	Check if str1 is greater than str2.
-n str1	Check if str1 has a length greater than zero.
-z str1	Check if str1 has a length of zero.

```
testuser=rich
if [ $USER = $testuser ]
then
echo "Welcome $testuser"
fi
```

```
$ ./test7
Welcome rich
```

File comparisons

The last category of test comparisons is quite possibly the most powerful and most used comparisons in shell scripting. The test command allows you to test the status of files and directories on the Linux file system. These conditions give you the ability to check files in your file system within your shell scripts, and they are often used in scripts that access files.

equation	**condition to return logical true**
`-e <file>`	<file> exists
`-d <file>`	<file> exists and is a directory
`-f <file>`	<file> exists and is a regular file
`-w <file>`	<file> exists and is writable
`-x <file>`	<file> exists and is executable
`<file1> -nt <file2>`	<file1> is newer than <file2> (modification)
`<file1> -ot <file2>`	<file1> is older than <file2> (modification)
`<file1> -ef <file2>`	<file1> and <file2> are on the same device and the same inode number

Double parentheses

The *double parentheses* command allows you to incorporate advanced mathematical formulas in your comparisons. The test command only allows for simple arithmetic operations in the comparison.
The double parentheses command provides more mathematical symbols that programmers from other languages are used to using. The format of the double parentheses command is:

((*expression*))

The *expression* term can be any mathematical assignment or comparison expression. Besides the standard mathematical operators that the test command uses, additional operators are shown in the below table.

```
val1=10
if (( $val1 ** 2 > 90 ))
then
     (( val2 = $val1 ** 2 ))
     echo "The square of $val1 is $val2"
fi
$ ./test23
The square of 10 is 100
```

Symbol	Description
val++	post-increment
val--	post-decrement
++val	pre-increment
--val	pre-decrement
!	logical negation
□	bitwise negation
**	exponentiation
<<	left bitwise shift
>>	right bitwise shift
&	bitwise Boolean AND
|	bitwise Boolean OR
&&	logical AND
||	logical OR

Case command

Often you'll find yourself trying to evaluate the value of a variable, looking for a specific value within a set of possible values. In this scenario, you end up having to write a lengthy if-then else statement, like this

```
if [ $USER = "rich" ]
then
     echo "Welcome $USER"
     echo "Please enjoy your visit"
elif [ $USER = barbara ]
then
     echo "Welcome $USER"
     echo "Please enjoy your visit"
elif [ $USER = testing ]
then
     echo "Special testing account"
elif [ $USER = jessica ]
then
     echo "Don't forget to logout when you're done"
else
     echo "Sorry, you're not allowed here"
fi
```

The elif statements continue the if-then checking, looking for a specific value for the single comparison variable. Instead of having to write all of the elif statements to continue checking the same variable value, you can use the case command. The case command checks multiple values of a single variable in a list-oriented format:

```
case variable in

pattern1 | pattern2) commands1;;

pattern3) commands2;;

*) default commands;;

esac
```

The case command compares the variable specified against the different patterns. If the variable matches the pattern, the shell executes the commands specified for the pattern. You can list more than one pattern on a line, using the bar operator to separate each pattern. The asterisk symbol is the catch-all for values that don’t match any of the listed patterns.

```
case $USER in
rich)          echo "Welcome, $USER"
               echo "Please enjoy your visit";;
testing)       echo "Special testing account";;
jessica)       echo "Don’t forget to log off when you’re done";;
*)             echo "Sorry, you’re not allowed here";;
esac

$. /myscript
Welcome, rich
Please enjoy your visit
```

for command

Iterating through a series of commands is a common programming practice. Often you need to repeat a set of commands until a specific condition has been met, such as processing all of the files in a directory, all of the users on a system, or all of the lines in a text file.

The bash shell provides the `for` command to allow you to create a loop that iterates through a series of values. Each iteration performs a defined set of commands using one of the values in the series.
The basic format of the bash shell for command is:

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

You supply the series of values used in the iterations in the list parameter. There are several different ways that you can specify the values in the list. In each iteration, the variable var contains the current value in the list. The first iteration uses the first item in the list, the second iteration the second item, and so on until all of the items in the list have been used.
The commands entered between the do and done statements can be one or more standard bash shell commands. Within the commands the $var variable contains the current list item value for the iteration.

```
for test in Alabama Alaska Arizona Arkansas California Colorado
do
echo The next state is $test
done

 . /test1
The next state is Alabama
The next state is Alaska
The next state is Arizona
The next state is Arkansas
The next state is California
The next state is Colorado
```

The C-Style for Command

```
for (( variable assignment ; condition ; iteration process ))
```

The format of the C-style for loop can be confusing for bash shell script programmers, as it uses C-style variable references instead of the shell-style variable references.

Here's what a C-style for command looks like:
```
for (( a = 1; a ‹ 10; a++ ))
```

Notice that there are a couple of things that don't follow the standard bash shell for method:

■ The assignment of the variable value can contain spaces.
■ The variable in the condition isn't preceded with a dollar sign.
■ The equation for the iteration process doesn't use the expr command format.

Example:
```
for (( i=1; i ‹= 10; i++ ))
do
echo "The next number is $i"
done
```

Using multiple variables in for command

```
for (( a=1, b=10; a ‹= 10; a++, b-- ))
do
     echo "$a - $b"
done
```

while command

The while command is somewhat of a cross between the if-then statement and the for loop. The while command allows you to define a command to test, then loop through a set of commands for as long as the defined test command returns a zero exit status. It tests the test command at the start of each iteration. When the test command returns a non-zero exit status, the while command stops executing the set of commands.

The format of the while command is:
```
     while test command
     do
     other commands
     done
```

The test command defined in the while command is the exact same format as in if-then statements. Just as in the if-then statement, you can use any normal bash shell command, or you can use the test command to test for conditions, such as variable values.
The key to the while command is that the exit status of the test command specified must change, based on the commands run during the loop. If the exit status never changes, the while loop will get stuck in an infinite loop.
The most common situation uses the test command brackets to check a value of a shell variable that's used in the loop commands:

```
var1=10
while [ $var1 -gt 0 ]
do
echo $var1
var1=$[ $var1 - 1 ]
done
```

The while command defines the test condition to check for each iteration:

```
while [ $var1 -gt 0 ]
```

As long as the test condition is true, the while command continues to loop through the commands defined. Within the commands, the variable used in the test condition must be modified, or else you'll have an infinite loop. In this example, we use shell arithmetic to decrease the variable value by one:

```
var1=$[ $var1 - 1 ]
```

The while loop stops when the test condition is no longer true.

until command

Until command works exactly the opposite to the way while command works. The until command requires that you to specify a test command that normally produces a non-zero exit status. As long as the exit status of the test command is non-zero, the bash shell executes the commands listed in the loop. Once the test command returns a zero exit status and the loop stops.

Format:

```
until test commands
do
other commands
done
```

Example:

```
var1=100
until [ $var1 -eq 0 ]
do
echo $var1
var1=$[ $var1 - 25 ]
done
```

output:

```
$. /test12
100
75
50
25
```