

## VI editor

The VI editor is the most popular and classic text editor in the Linux family. Below, are some reasons which make it a widely used editor –

- 1) It is available in almost all Linux Distributions
- 2) It works the same across different platforms and Distributions
- 3) It is user-friendly. Hence, millions of Linux users love it and use it for their editing needs

Nowadays, there are advanced versions of the vi editor available, and the most popular one is **VIM** which is **Vi Improved**. Some of the other ones are Elvis, Nvi, Nano, and Vile. It is wise to learn vi because it is feature-rich and offers endless possibilities to edit a file.

To work on VI editor, you need to understand **its operation modes**. They can be divided into two main parts.

### Command mode:

- The vi editor opens in this mode, and it only **understands commands**
- In this mode, you can, **move the cursor and cut, copy, paste the text**
- This mode also saves the changes you have made to the file
- **Commands are case sensitive**. You should use the right letter case.

### Insert mode:

- This mode is for inserting text in the file.
- You can switch to the Insert mode from the command mode **by pressing 'i' on the keyboard**
- Once you are in Insert mode, any key would be taken as an input for the file on which you are currently working.
- To return to the command mode and save the changes you have made you need to press the Esc key

## Starting the vi editor

To launch the VI Editor -Open the Terminal (CLI) and type

```
vi <filename_NEW> or <filename_EXISTING>
```

And if you specify an existing file, then the editor would open it for you to edit. Else, you can create a new file.



### Vi editor commands

- a - Write after cursor (goes into insert mode)
- A - Write at the end of line (goes into insert mode)
- ESC - Terminate insert mode
- u - Undo last change
- U - Undo all changes to the entire line
- o - Open a new line (goes into insert mode)
- dd - Delete line
- 3dd - Delete 3 lines.
- D - Delete contents of line after the cursor
- C - Delete contents of a line after the cursor and insert new text. Press ESC key to end insertion.
- dw - Delete word
- 4dw - Delete 4 words
- cw - Change word
- x - Delete character at the cursor
- r - Replace character
- R - Overwrite characters from cursor onward
- s - Substitute one character under cursor continue to insert
- S - Substitute entire line and begin to insert at the beginning of the line
- ~ - Change case of individual character

**Note:** You should be in the "command mode" to execute these commands. VI editor is **case-sensitive** so make sure you type the commands in the right letter-case.

Make sure you press the right command otherwise you will end up making undesirable changes to the file. You can also enter the insert mode by pressing a, A, o, as required.

### Moving within a file

- k - Move cursor up
- j - Move cursor down
- h - Move cursor left
- l - Move cursor right

You need to be in the command mode to move within a file. The default keys for navigation are mentioned below else; You can **also use the arrow keys on the keyboard**.

### Saving and Closing the file

- Shift+zz - Save the file and quit
- :w - Save the file but keep it open
- :q - Quit without saving
- :wq - Save the file and quit

You should be in the **command mode to exit the editor and save changes** to the file.

---

## Introduction to Shell Scripting

A shell script is a computer program designed to be run by the Unix/Linux shell which could be one of the following:

- The Bourne Shell
- The C Shell
- The Korn Shell
- The GNU Bourne-Again Shell(BASH)

A shell is a command-line interpreter and typical operations performed by shell scripts include file manipulation, program execution, and printing text.

Usually shells are interactive that mean, they accept command as input from users and execute them. However, some time we want to execute a bunch of commands routinely, so we have type in all commands each time in terminal. As shell can also take commands as input from file we can write these commands in a file and can execute them in shell to avoid this repetitive work. These files are called **Shell Scripts** or **Shell Programs**. Shell scripts are similar to the **batch file** in MS-DOS. Each shell script is saved with **.sh** file extension eg. **myscript.sh**

A shell script have syntax just like any other programming language. If you have any prior experience with any programming language like Python, C/C++ etc. it would be very easy to get started with it.

*A shell script comprises following elements -*

- **Shell Keywords** - if, else, break etc.
- **Shell commands** - cd, ls, echo, pwd, touch etc.
- Functions
- **Control flow** - if..then..else, case and shell loops etc.

## Why do we need shell scripts

There are many reasons to write shell scripts -

- To avoid repetitive work and automation
- System admins use shell scripting for routine backups
- System monitoring
- Adding new functionality to the shell etc.

## Advantages of shell scripts

- The command and syntax are exactly the same as those directly entered in command line, so programmer do not need to switch to entirely different syntax
- Writing shell scripts are much quicker
- Quick start
- Interactive debugging etc.

## Disadvantages of shell scripts

- Prone to costly errors, a single mistake can change the command which might be harmful
- Slow execution speed
- Design flaws within the language syntax or implementation
- Not well suited for large and complex task
- Provide minimal data structure unlike other scripting languages. Etc

## About Bash

**Bash** ("Bourne Again Shell") shell as the main shell interpreter. Shell programming using other common shells such as sh, csh, tcsh, will also be referenced, as they sometime differ from bash.

Shell programming can be accomplished by directly executing shell commands at the shell prompt or by storing them in the order of execution, in a text file, called a shell script, and then executing the shell script. To execute, simply write the shell script file name, once the file has execute permission (chmod +x filename).

The first line of the shell script file begins with a "sha-bang" (!) which is not read as a comment, followed by the full path where the shell interpreter is located. This path, tells the operating system that this file is a set of commands to be fed into the interpreter indicated. Note that if the path given at the "sha-bang" is incorrect, then an error message e.g. "Command not found.", may be the result of the script execution. It is common to name the shell script with the ".sh" extension. The first line may look like this:

```
#!/bin/bash
```

Adding comments: any text following the "#" is considered a comment

To find out what is currently active shell, and what is its path, type the highlighted command at the shell prompt (sample responses follow):

```
ps | grep $$
```

```
987 tty1 00:00:00 bash
```

This response shows that the shell you are using is of type 'bash'. next find out the full path of the shell interpreter

```
which bash
```

```
/bin/bash
```

This response shows the full execution path of the shell interpreter. Make sure that the "sha-bang" line at the beginning of your script, matches this same execution path.

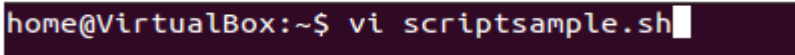
## Steps in creating a Shell Script

1. Create a file using a vi editor(or any other editor). Name script file with **extension .sh**
2. **Start** the script with **#!/bin/sh**
3. Write some code.
4. Save the script file as filename.sh
5. For **executing** the script type **bash filename.sh**

"#!" is an operator called shebang which directs the script to the interpreter location. So, if we use"#! /bin/sh" the script gets directed to the bourne-shell.

Let's see the steps to create it -

Creating a new script file scriptsample.sh



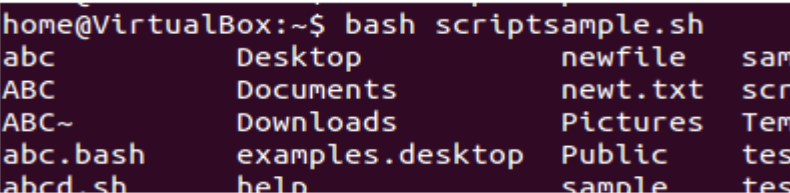
```
home@VirtualBox:~$ vi scriptsample.sh
```

Adding the command 'ls' after #!/bin/sh



```
#!/bin/sh
ls
~
```

Executing the script file



```
home@VirtualBox:~$ bash scriptsample.sh
abc      Desktop      newfile     sam
ABC      Documents   newt.txt    scr
ABC~     Downloads   Pictures    Temp
abc.bash examples.desktop Public      test
abcd.sh  help       sample     test
```

Command 'ls' is executed when we execute the scrip sample.sh file.

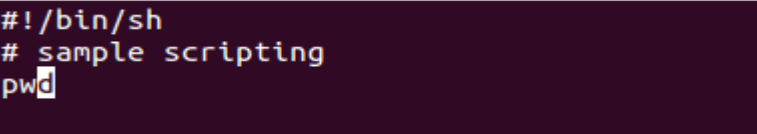
## Adding shell comments

Commenting is important in any program. In Shell programming, the syntax to add a comment is

**#comment**

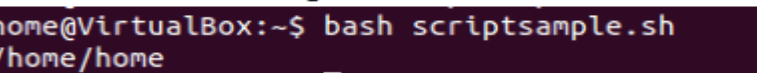
Let understand this with an example.

Adding a comment



```
#!/bin/sh
# sample scripting
pwd
```

Shell executes only the command



```
home@VirtualBox:~$ bash scriptsample.sh
/home/home
```

It ignores the comment **# sample scripting**

## To compile and run the program

**Chmod 777 filename.sh or ./filename.sh or bash filename.sh**

### Variable

Shell variable is a value, which changes during the execution of program. Shell variables are created once they are assigned a value. A variable can contain a number, a character or a string of characters. A variable is nothing more than a pointer to the actual data. The shell enables you to create, assign, and delete variables.

### System Defined Variables:

These are the variables which are created and maintained by **Operating System(Linux) itself**. Generally these variables are defined in **CAPITAL LETTERS**. We can see these variables by using the command “\$ set”. Some of the system defined variables are given below :

System Defined Variables	Meaning
BASH=/bin/bash	Shell Name
BASH_VERSION=4.1.2(1)	Bash Version
COLUMNS=80	No. of columns for our screen
HOME=/home/linuxtechi	Home Directory of the User
LINES=25	No. of columns for our screen
LOGNAME=LinuxTechi	LinuxTechi Our logging name
OSTYPE=Linux	OS type
PATH=/usr/bin:/sbin:/bin:/usr/sbin	Path Settings
PS1=[\u@\h \W]\\$	Prompt Settings
PWD=/home/linuxtechi	Current Working Directory
SHELL=/bin/bash	Shell Name
USERNAME=linuxtechi	User name who is currently login to system

To Print the value of above variables, use **echo command** as shown below :

```
# echo $HOME
```

```
# echo $USERNAME
```

We can tap into these environment variables from within your scripts by using the environment variable's name preceded by a dollar sign. This is demonstrated in the following script:

```
$ cat myscript
#!/bin/bash
# display user information from the system.
echo "User info for userid: $USER"
echo UID: $UID
echo HOME: $HOME
```

Notice that the **environment variables** in the echo commands are replaced by their current values when the script is run. Also notice that we were able to place the `$USER` system variable within the double quotation marks in the first string, and the shell script was still able to figure out what we meant. There is a **drawback** to using this method, however. Look at what happens in this example:

```
$ echo "The cost of the item is $15"
The cost of the item is 5
```

That is obviously not what was intended. Whenever the script sees a dollar sign within quotes, it assumes you're referencing a variable. In this example the script attempted to display the **variable \$1** (which was not defined), and then the number 5. To display an actual dollar sign, you **must precede** it with a **backslash character**:

```
$ echo "The cost of the item is \$15"
The cost of the item is $15
```

That's better. The backslash allowed the shell script to interpret the **dollar sign** as an actual dollar sign, and not a variable.

### User Defined Variables:

These variables are defined by **users**. A shell script allows us to set and use our **own variables** within the script. Setting variables allows you to **temporarily store data** and use it throughout the script, making the shell script more like a real computer program.

**User variables** can be any text string of up to **20 letters, digits, or an underscore character**.

### Rules for declaring variables

- Variable name is case sensitive
- The name of a variable can contain only letters (a to z or A to Z), numbers (0 to 9) or the underscore character (\_). Value assignment is done using the "=" sign.
- Note that no space permitted on either side of = sign when initializing variables.



## Defining Variables

Variables are defined as follows –

**variable\_name=variable\_value**

Here are a few examples of assigning values to user variables:

```
var1=10
var2=-57
var3=testing
var4="still more testing"
```

The shell script **automatically determines the data type** used for the variable value. Variables defined within the shell script maintain their values throughout the life of the shell script but are deleted when the shell script completes.

## Accessing Values

To access the value stored in a variable, prefix its name with the dollar sign (\$)–

For example, the following script will access the value of defined variable NAME and print it on STDOUT –

```
#!/bin/bash
NAME="BCA VI SEM"
echo $NAME
```

The above script will produce the following value –

BCA VI SEM

## Read-only Variables or Constants

Shell provides a way to mark variables as read-only by using the read-only command. After a variable is marked read-only, its value cannot be changed.

For example, the following script generates an error while trying to change the value of NAME –

```
#!/bin/sh
NAME="PEARLS"
readonly NAME
NAME="BCA VI SEM"
```

The above script will generate the following result –

/bin/sh: NAME: This variable is read only.

## Unsetting Variables

Unsetting or deleting a variable directs the shell to remove the variable from the list of variables that it tracks. Once you unset a variable, you cannot access the stored value in the variable.

Following is the syntax to unset a defined variable using the **unset** command –

**unset variable\_name**

The above command unsets the value of a defined variable. Here is a simple example that demonstrates how the command works –

```
#!/bin/sh
```

```
NAME="BCA VI SEM"
unset NAME
echo $NAME
```

The above example does not print anything. You cannot use the unset command to **unset** variables that are marked **readonly**.

## Variable Types

When a shell is running, three main types of variables are present –

- **Local Variables** – A local variable is a variable that is present within the current instance of the shell. It is not available to programs that are started by the shell. They are set at the command prompt.
- **Environment Variables** – An environment variable is available to any child process of the shell. Some programs need environment variables in order to function correctly. Usually, a shell script defines only those environment variables that are needed by the programs that it runs.
- **Shell Variables** – A shell variable is a special variable that is set by the shell and is required by the shell in order to function correctly. Some of these variables are environment variables whereas others are local variables.

## EXPR COMMAND

The **expr** command in UNIX evaluates a given expression and displays its corresponding output. It is used for:

- Basic operations like addition, subtraction, multiplication, division, and modulus on integers.
- Evaluating regular expressions, string operations like substring, length of strings etc.

**Syntax:**

```
$expr expression
```

**Options:**

- **Option -version** : It is used to show the version information.

**Syntax:**

```
$expr --version
```

**Example:**

```
anshul@anshul-VirtualBox:~/Desktop$ expr --version
expr (GNU coreutils) 8.28
Copyright (C) 2017 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>.
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

Written by Mike Parker, James Youngman, and Paul Eggert.
anshul@anshul-VirtualBox:~/Desktop$
```

- **Option -help** : It is used to show the help message and exit.

**Syntax:**

```
$expr --help
```

**Example:**

```
anshul@anshul-VirtualBox:~/Desktop$ expr --help
Usage: expr EXPRESSION
      or:  expr OPTION

      --help      display this help and exit
      --version   output version information and exit

Print the value of EXPRESSION to standard output.  A blank line below
separates increasing precedence groups.  EXPRESSION may be:

ARG1 | ARG2      ARG1 if it is neither null nor 0, otherwise ARG2

ARG1 & ARG2      ARG1 if neither argument is null or 0, otherwise 0

ARG1 < ARG2      ARG1 is less than ARG2
ARG1 <= ARG2     ARG1 is less than or equal to ARG2
ARG1 = ARG2      ARG1 is equal to ARG2
ARG1 != ARG2     ARG1 is unequal to ARG2
ARG1 >= ARG2     ARG1 is greater than or equal to ARG2
ARG1 > ARG2      ARG1 is greater than ARG2

ARG1 + ARG2      arithmetic sum of ARG1 and ARG2
ARG1 - ARG2      arithmetic difference of ARG1 and ARG2

ARG1 * ARG2      arithmetic product of ARG1 and ARG2
ARG1 / ARG2      arithmetic quotient of ARG1 divided by ARG2
ARG1 % ARG2      arithmetic remainder of ARG1 divided by ARG2

STRING : REGEXP  anchored pattern match of REGEXP in STRING

match STRING REGEXP  same as STRING : REGEXP
substr STRING POS LENGTH  substring of STRING, POS counted from 1
index STRING CHARS      index in STRING where any CHARS is found, or 0
length STRING           length of STRING
+ TOKEN                 interpret TOKEN as a string, even if it is a
                        keyword like 'match' or an operator like '/'

( EXPRESSION )         value of EXPRESSION

Beware that many operators need to be escaped or quoted for shells.
Comparisons are arithmetic if both ARGs are numbers, else lexicographical.
Pattern matches return the string matched between \( and\) or null; if
\( and\) are not used, they return the number of characters matched or 0.

Exit status is 0 if EXPRESSION is neither null nor 0, 1 if EXPRESSION is null
or 0, 2 if EXPRESSION is syntactically invalid, and 3 if an error occurred.
```

Below are some examples to demonstrate the use of "expr" command:

### 1. Using expr for basic arithmetic operations :

**Example: Addition**

```
$expr 12 + 8
```

**Example: Multiplication**

```
$expr 12 \* 2
```

**Output**

```
root@genesis101: ~
File Edit View Search Terminal Help
root@genesis101:~# expr 12 + 8
20
root@genesis101:~# expr 12 \* 2
24
root@genesis101:~#
```

**Note:**The multiplication operator \* must be escaped when used in an arithmetic expression with *expr*.

## 2. Performing operations on variables inside a shell script

**Example:** Adding two numbers in a script

```
echo "Enter two numbers"

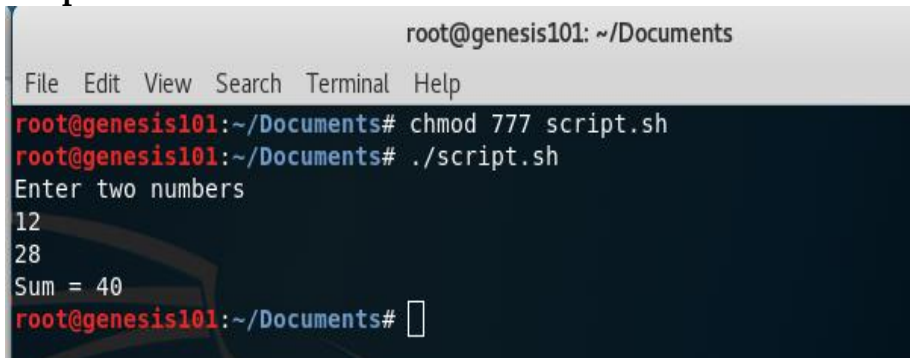
read x

read y

sum=expr $x + $y

echo "Sum = $sum"
```

**Output:**



```
root@genesis101: ~/Documents
File Edit View Search Terminal Help
root@genesis101:~/Documents# chmod 777 script.sh
root@genesis101:~/Documents# ./script.sh
Enter two numbers
12
28
Sum = 40
root@genesis101:~/Documents#
```

**Note:** *expr* is an external program used by Bourne shell. It uses *expr* external program with the help of backtick. The *backtick*(```) is actually called command substitution.

## 3. Comparing two expressions

**Example:**

```
x=10

y=20

# matching numbers with '='
res=`expr $x = $y`

echo $res

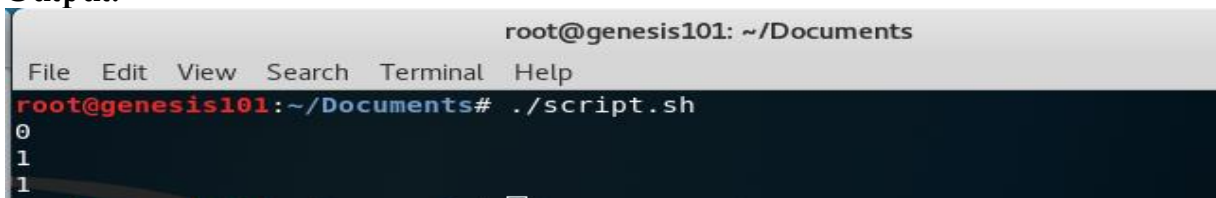
# displays 1 when arg1 is less than arg2
res=`expr $x \< $y`

echo $res

# display 1 when arg1 is not equal to arg2
res=`expr $x \!= $y`

echo $res
```

**Output:**



```
root@genesis101: ~/Documents
File Edit View Search Terminal Help
root@genesis101:~/Documents# ./script.sh
0
1
1
```

**Example:** Evaluating boolean expressions

# OR operation

```
$expr length "geekss" "<" 5 "|" 19 - 6 ">" 10
```

**Output:**

```

root@genesis101: ~/Documents
File Edit View Search Terminal Help
root@genesis101:~/Documents# expr length "geekss" "<" 5 "|" 19 - 6 ">" 10
1
root@genesis101:~/Documents#

```

# AND operation

```
$expr length "geekss" "<" 5 "&" 19 - 6 ">" 10
```

**Output:**

```

root@genesis101: ~/Documents
File Edit View Search Terminal Help
root@genesis101:~/Documents# expr length "geekss" "<" 5 "&" 19 - 6 ">" 10
0
root@genesis101:~/Documents#

```

**4. For String operations**

**Example:** Finding length of a string

```
x=geeks
```

```
len=`expr length $x`
```

```
echo $len
```

**Output:**

```

root@genesis101: ~/Documents
File Edit View Search Terminal Help
root@genesis101:~/Documents# ./script.sh
5
root@genesis101:~/Documents#

```

**Example:** Finding substring of a string

```
x=geeks
```

```
sub=`expr substr $x 2 3`
```

```
#extract 3 characters starting from index 2
```

```
echo $sub
```

**Output:**

```

root@genesis101: ~/Documents
File Edit View Search Terminal Help
root@genesis101:~/Documents# ./script.sh
eek
root@genesis101:~/Documents# █

```

**Example:** Matching number of characters in two strings

```
$ expr geeks : geek
```

**Output:**

```

root@genesis101: ~/Documents
File Edit View Search Terminal Help
root@genesis101:~/Documents# expr geeks : geek
4
root@genesis101:~/Documents# █

```

## READ AND ECHO STATEMENTS OR COMMANDS

**read** is a built-in command of the Bash shell. It reads a line of text from standard input and splits it into words. These words can then be used as the input for other commands.

The built in command reads a line of input and separates the line into individual words using the "IFS" inter field separator. (see IFS). By default, the "IFS" is set to a space. Each word in the line is stored in a variable from left to right. The first word is stored in the first variable, the second word to the second variable and so on. If there are fewer variables than words, then all remaining words are then assigned to the last variable. If you have more variables than words defined, then any excess variables are set to null. If no variable names are supplied to the read line, then the read uses the default variable `REPLY`.

### read example

```

#!/bin/bash
echo "Please enter 3 words followed by ENTER:"
read first middle last
echo "Hello $first $middle $last"

```

### Output from above read example

```

john@john-desktop:~/scripts$ ./read2.sh
Please enter 3 words followed by ENTER:
land of Unix
Hello land of Unix

```

We could also use the "-p" flag with the read command:

**read -p example**

```
#!/bin/bash
read -p "Please Enter first word followed by ENTER: " first
read -p "Please Enter second word followed by ENTER: " middle
read -p "Please Enter last word followed by ENTER: " last
echo "Hello $first $middle $last"
```

**Output from above read -p example**

```
john@john-desktop:~/scripts$ ./read4.sh
Please Enter first word followed by ENTER: land
Please Enter second word followed by ENTER: of
Please Enter last word followed by ENTER: Unix
Hello land of Unix
```

**Options available to the read command**

Below is a table containing other parameters that may be used with the read built in in command:

Option	Description
-a ANAME	Words are assigned sequentially to the array variable ANAME
-d DELIM	The first character of DELIM us used to terminate the input line
-e	readline is used to get line
-n NCHARS	read returns after reading NCHARS
-p PROMPT	Display PROMPT without a trailing newline. Prompt is only displayed if coming from a terminal.
-r	Backslash does not act as an escape character
-s	Silent Mode. Characters are not echoed coming from a Terminal
-t TIMEOUT	read will timeout after TIMEOUT seconds. Only from a Terminal
-u FD	read input from Filed Descriptor FD

**ECHO COMMAND OR STATEMENT**

*echo* is a built-in *command* in the *bash* and *C shells* that writes its *arguments* to *standard output*.

A shell is a program that provides the command line (i.e., the all-text display user interface) on Linux and other Unix-like operating systems. It also executes (i.e., runs) commands that are typed into it and displays the results. *bash* is the default shell on Linux.

A command is an instruction telling a computer to do something. An argument is input data for a command. Standard output is the display screen by default, but it can be redirected to a file, printer, etc.

The syntax for echo is

**echo [option(s)] [string(s)]**

The items in square brackets are optional. A *string* is any finite sequence of *characters* (i.e., letters, numerals, symbols and punctuation marks).

When used without any options or strings, echo returns a blank line on the display screen followed by the command prompt on the subsequent line. This is because pressing the ENTER key is a signal to the system to start a new line, and thus echo repeats this signal.

When one or more strings are provided as arguments, echo by default repeats those strings on the screen. Thus, for example, typing in the following and pressing the ENTER key would cause echo to repeat the phrase *This is a pen.* on the screen:

**echo This is a pen.**

It is not necessary to surround the strings with quotes, as it does not affect what is written on the screen. If quotes (either single or double) are used, they are not repeated on the screen.

### Echo Options

These options may be specified before the string, and affect the behavior of **echo**.

<b>-n</b>	Do not output a trailing <u>newline</u> .
<b>-e</b>	Enable interpretation of backslash <u>escape sequences</u> (see below for a list of these).
<b>-E</b>	Disable interpretation of backslash escape sequences. This is the default.

### Options

If a long option is specified, you may not specify a string to be echoed. These options are for getting information about the program only.

<b>--help</b>	Display a help message and exit.
<b>--version</b>	Output version information and exit.



- **Display a line of text on standard output.**

```
echo Hello, World!
Hello, World!
```

- **Display a line of text containing a double quote.**

To print a double quote, enclose it within single quotes or escape it with the backslash character.

```
echo 'Hello "Linuxize"'
echo "Hello \"Linuxize\""
```

Hello "Linuxize"

- **Display a line of text containing a single quote.**

To print a single quote, enclose it within double quotes or use the [ANSI-C Quoting](#).

```
echo "I'm a Linux user."
echo $'I\'m a Linux user.'
```

I'm a Linux user.

- **Display a message containing special characters.**

Use the `-e` option to enable the interpretation of the escape characters.

```
echo -e "You know nothing, Jon Snow.\n\t- Ygritte"
You know nothing, Jon Snow.
    - Ygritte
```

- **Pattern matching characters.**

The `echo` command can be used with pattern matching characters, such as the wildcard characters. For example, the command below will return the names of all the `.php` files in the current directory.

```
echo The PHP files are: *.php
The PHP files are: index.php contact.php functions.php
```

- **Redirect to a file**

Instead of displaying the output on the screen, you can redirect it to a file using the `>`, `>>` operators.

```
echo -e 'The only true wisdom is in knowing you know nothing.\nSocrates' >>
/tmp/file.txt
```

If the `file.txt` doesn't exist, the command will create it. When using `>` the file will be overwritten, while the `>>` will [append the output to the file](#).

Use the `cat` command to view the content of the file:

```
cat /tmp/file.txt
```

```
The only true wisdom is in knowing you know nothing.  
Socrates
```

- **Displaying variables**

`echo` can also display variables. In the following example, we'll print the name of the currently logged in user:

```
echo $USER  
linuxize
```

`$USER` is a shell variable that holds your username.

- **Displaying output of a command**

Use the `$(command)` expression to include the command output in the `echo`'s argument. The following command will display the current date:

```
echo "The date is: $(date +%D)"  
The date is: 04/17/19
```

- **Displaying in color**

Use ANSI escape sequences to change the foreground and background colors or set text properties like underscore and bold.

```
echo -e "\033[1;37mWHITE"  
echo -e "\033[0;30mBLACK"  
echo -e "\033[0;34mBLUE"  
echo -e "\033[0;32mGREEN"  
echo -e "\033[0;36mCYAN"  
echo -e "\033[0;31mRED"  
echo -e "\033[0;35mPURPLE"  
echo -e "\033[0;33mYELLOW"  
echo -e "\033[1;30mGRAY"
```

```
WHITE  
BLACK  
BLUE  
GREEN  
CYAN  
RED  
PURPLE  
YELLOW  
GRAY
```

## Escape Sequences

**echo** recognizes a number of *escape sequences* which it expands internally. An escape command is a backslash-escaped character that signifies some other character. The ones recognized by **echo** are common throughout the shell syntax, as follows:

If the `-e` option is given, the following backslash-escaped characters will be interpreted:

<code>\\</code>	Displays a backslash character.
<code>\a</code>	Alert (BEL)
<code>\b</code>	Displays a backspace character.
<code>\c</code>	Suppress any further output
<code>\e</code>	Displays an escape character.
<code>\f</code>	Displays a form feed character.
<code>\n</code>	Displays a new line.
<code>\r</code>	Displays a carriage return.
<code>\t</code>	Displays a horizontal tab.
<code>\v</code>	Displays a vertical tab.

- The `-E` option disables the interpretation of the escape characters. This is the default.

## COMMAND SUBSTITUTION

Command substitution is the mechanism by which the shell performs a given set of commands and then substitutes their output in the place of the commands.

### Syntax

The command substitution is performed when a command is given as –

### **\$(command)**

When performing the command substitution make sure that you use the `$`, with parenthesis `()`.

### Example

Command substitution is generally used to assign the output of a command to a variable. Each of the following examples demonstrates the command substitution –

```
#!/bin/sh
DATE=$(date)
echo "Date is $DATE"
USERS=$(who | wc -l)
echo "Logged in user are $USERS"
```

Upon execution, you will receive the following result –

```
Date is Thu Jul 2 03:59:57 IST 2020
Logged in user are 1
Uptime is Thu Jul 2 03:59:57 IST 2020
03:59:57 up 20 days, 14:03, 1 user, load avg: 0.13, 0.07, 0.15
```

## SHELL SCRIPT ARGUMENTS AND POSITIONAL PARAMETERS

A bash shell script have parameters. These parameters start from \$1 to \$9.

When we pass arguments into the command line interface, a **positional parameter** is assigned to these arguments through the shell.

The first argument is assigned as \$1, second argument is assigned as \$2 and so on...

If there are more than 9 arguments, then **tenth** or onwards arguments can't be assigned as \$10 or \$11.

You have to either process or save the \$1 parameter, then with the help of **shift** command drop parameter 1 and move all other arguments down by one. It will make \$10 as \$9, \$9 as \$8 and so on.

### Shell Parameters

Parameters	Function
\$1-\$9	Represent positional parameters for arguments one to nine
\${10}-\${n}	Represent positional parameters for arguments after nine
\$0	Represent name of the script
\$*	Represent all the arguments as a single string
@	Same as \$*, but differ when enclosed in (")
\$#	Represent total number of arguments
\$\$	PID of the script
\$?	Represent last return code

### Example:

```

sssit@JavaTpoint: ~
#!/bin/bash
echo this script name is $0
#
echo The first argument is $1
echo The second argument is $2
echo And third argument is $3
#
echo \$ $$ PID of the script
echo \# $# Total number of arguments
"script.sh" 14 lines, 264 characters
    
```

Look at the above snapshot, this is the script we have written to show the different parameters.

```

sssit@JavaTpoint: ~
sssit@JavaTpoint:~$ chmod +x script.sh
sssit@JavaTpoint:~$ ./script.sh 1 5 90
this script name is ./script.sh
The first argument is 1
The second argument is 5
And third argument is 90
$ 2488 PID of the script
# 3 Total number of arguments
? 0 Last return code
* 1 5 90 All the arguments
sssit@JavaTpoint:~$
    
```

Look at the above snapshot, we have passed arguments **1, 5, 90**. All the parameters show their value when script is run.

## TEST Command

On Unix-like operating systems, **test** is a builtin command of the Bash shell that can test file attributes, and perform string and arithmetic comparisons.

**test** is used as part of the **conditional** execution of shell commands. **test** exits with the status determined by **EXPRESSION**. Placing the **EXPRESSION** between square brackets ([ and ]) is the same as testing the **EXPRESSION** with **test**. To see the exit status at the command prompt, echo the value "\$?". A value of 0 means the expression evaluated as true, and a value of 1 means the expression evaluated as false. **test** provides no output, but returns an exit status of **0** for "true" (test successful) and **1** for "false" (test failed).

The **test** command is frequently used as part of a conditional expression. For instance, the following statement says, "If 4 is greater than 5, output yes, otherwise output no."

```

num=4;
if (test $num -gt 5); then
echo "yes";
else
echo "no"; fi
    
```

**no**

The following statement says, "If 6 is greater than 5, output yes, otherwise output no."

```

num=6;
if (test $num -gt 5); then
echo "yes";
else
echo "no"; fi
    
```

**yes**

The **test** command may also be expressed with single brackets [ ... ], as long as they are separated from all other arguments with whitespace. For example, the following statement checks that the system file **/etc/passwd** exists, and if not, outputs "uh-oh."

```
file="/etc/passwd";
if [ -e $file ]; then
echo "whew";
else echo "uh-oh";
fi
```

whew

## Syntax

### File tests:

```
test [-a] [-b] [-c] [-d] [-e] [-f] [-g] [-h] [-L] [-k] [-p] [-r] [-s] [-S] [-u] [-w] [-x] [-O] [-G] [-N]
[file]
```

```
test -t fd
```

```
test file1 {-nt | -ot | -ef} file2
```

### String tests:

```
test [-n | -z] string
```

```
test string1 {= | != | < | >} string2
```

### Shell options and variables:

```
test -o option
```

```
test {-v | -R} var
```

### Simple logic (test if values are null):

```
test [!] expr
```

```
test expr1 {-a | -o} expr2
```

### Numerical comparison (for integer values only; bash doesn't do floating point math):

```
test arg1 {-eq | -ne | -lt | -le | -gt | -ge} arg2
```

**Test command Options**

The **test** builtin command takes the following options.

<b>-a file</b>	Returns true if <i>file</i> exists. Does the same thing as <b>-e</b> . Both are included for compatibility reasons with legacy versions of <u>Unix</u> .
<b>-b file</b>	Returns true if <i>file</i> is "block-special". Block-special files are similar to regular files, but are stored on block devices – special areas on the <u>storage device</u> that are written or read one <u>block</u> ( <u>sector</u> ) at a time.
<b>-c file</b>	Returns true if <i>file</i> is "character-special." Character-special files are written or read <u>byte-by-byte</u> (one <u>character</u> at a time), immediately, to a special device. For example, <b>/dev/urandom</b> is a character-special file.
<b>-d file</b>	Returns true if <i>file</i> is a <u>directory</u> .
<b>-e file</b>	Returns true if <i>file</i> exists. Does the same thing as <b>-a</b> . Both are included for compatibility reasons with legacy versions of Unix.
<b>-f file</b>	Returns true if <i>file</i> exists, and is a regular file.
<b>-r file</b>	Returns true if <i>file</i> is readable by the user running <b>test</b> .
<b>-s file</b>	Returns true if <i>file</i> exists, and is not empty.
<b>-w file</b>	Returns true if the user running <b>test</b> has <u>write permission</u> to <i>file</i> , i.e. make changes to it.
<b>-x file</b>	Returns true if <i>file</i> is <u>executable</u> by the user running <b>test</b> .
<b>-O file</b>	Returns true if <i>file</i> is <u>owned</u> by the user running <b>test</b> .
<b>-G file</b>	Returns true if <i>file</i> is owned by the group of the user running <b>test</b> .
<b>-N file</b>	Returns true if <i>file</i> was <u>modified</u> since the last time it was read.
<i>file1 -nt file2</i>	Returns true if <i>file1</i> is newer (has a newer <u>modification date/time</u> ) than <i>file2</i> .
<i>file1 -ot file2</i>	Returns true if <i>file1</i> is older (has an older modification date/time) than <i>file2</i> .
<i>file1 -ef file2</i>	Returns true if <i>file1</i> is a <u>hard link</u> to <i>file2</i> .
<b>test [-n] string</b>	Returns true if <i>string</i> is not empty. Operates the same with or without <b>-n</b> . For example, if <b>mystr=""</b> , then <b>test "\$mystr"</b> and <b>test -n "\$mystr"</b> would both be false. If <b>mystr="Not empty"</b> , then <b>test "\$mystr"</b> and <b>test -n "\$mystr"</b> would both be true.
<b>-z string</b>	Returns true if string <i>string</i> is empty, i.e. "".
<i>string1 = string2</i>	Returns true if <i>string1</i> and <i>string2</i> are equal, i.e. contain the same characters.

<code>string1 != string2</code>	Returns true if <i>string1</i> and <i>string2</i> are not equal.
<code>string1 &lt; string2</code>	Returns true if <i>string1</i> sorts before <i>string2</i> lexicographically, according to <u>ASCII</u> numbering, based on the first character of the string. For instance, <b>test "Apple" &lt; "Banana"</b> is true, but <b>test "Apple" &lt; "banana"</b> is false, because all <u>lowercase</u> letters have a lower ASCII number than their uppercase counterparts.  <b>Tip:</b> Enclose any variable names in double quotes to protect whitespace. Also, <u>escape</u> the less than symbol with a <u>backslash</u> to prevent bash from interpreting it as a <u>redirection</u> operator. For instance, use <b>test "\$str1" \ "\$str2"</b> instead of <b>test \$str1 &lt; \$str2</b> . The latter command will try to read from a file whose name is the value of variable <b>str2</b> . For more information, see <u>redirection in bash</u> .
<code>string1 &gt; string2</code>	Returns true if <i>string1</i> sorts after <i>string2</i> lexicographically, according to the ASCII numbering. As noted above, use <b>test "\$str1" \&gt; "\$str2"</b> instead of <b>test \$str1 &gt; \$str2</b> . The latter command creates or overwrites a file whose name is the value of variable <b>str2</b> .
<code>-v var</code>	Returns true if the <u>shell variable</u> <i>var</i> is <u>set</u> .
<code>-R var</code>	Returns true if the shell variable <i>var</i> is set, and is a name reference. (It's possible this refers to an <i>indirect reference</i> , as described in <u>Parameter expansion in bash</u> .)
<code>! expr</code>	Returns true if and only if the expression <i>expr</i> is null.
<code>expr1 -a expr2</code>	Returns true if expressions <i>expr1</i> and <i>expr2</i> are both not null.
<code>expr1 -o expr2</code>	Returns true if either of the expressions <i>expr1</i> or <i>expr2</i> are not null.
<code>arg1 -eq arg2</code>	true if <u>argument</u> <i>arg1</i> equals <i>arg2</i> .
<code>arg1 -ne arg2</code>	true if argument <i>arg1</i> is not equal to <i>arg2</i> .
<code>arg1 -lt arg2</code>	true if numeric value <i>arg1</i> is less than <i>arg2</i> .
<code>arg1 -le arg2</code>	true if numeric value <i>arg1</i> is less than or equal to <i>arg2</i> .
<code>arg1 -gt arg2</code>	true if numeric value <i>arg1</i> is greater than <i>arg2</i> .
<code>arg1 -ge arg2</code>	true if numeric value <i>arg1</i> is greater than or equal to <i>arg2</i>



---

## CONDITIONAL CONTROL STRUCTURES OR DECISION MAKING OR BRANCHING

Most of our programming languages today are able to make decisions based on conditions we set. A condition is an expression that evaluates to a Boolean value - true or false. Any programmer can make his program smart based on the decision and logic he puts into his program. The bash shell supports *if* and *switch (case)* decision statements.

### If statement

*If* is a statement that allows the programmer to make a decision in the program based on conditions he specified. If the condition is met, the program will execute certain lines of code otherwise, the program will execute other tasks the programmer specified. The following is the supported syntax of the *if* statement in the bash shell.

#### #1) The if statements

Simple *if* is used for decision making in shell script. if the given condition is true then it will execute the set of code that you have allocated to that block.

#### Syntax

```
if [ condition ]  
then  
Execute the statements  
fi
```

#### #1) Check if an input number is positive:

```
echo "Enter a number"  
read num  
if [ $num -gt 0 ]  
then  
echo "It is a positive number"  
fi
```

#### #2) The if...else statements

*if..else* is used for decision making in shell script where the given condition is true then it will execute the set of code that you have allocated to that block otherwise you can execute the rest of code for the false condition.

#### Syntax

```
if [ condition ]  
then  
Execute Statement if Condition is True  
elif  
Execute Statement if Condition is False  
fi
```

**#2) Check if an input number is positive or not:**

```
echo "Enter a number"
read num
if [ $num -gt 0 ]
then
echo "It is a positive number"
else
echo "It is not a positive integer"
fi
```

**#3) The if...elif...else...fi statement**

It is possible to create compound conditional statements by using one or more else if (elif) clause. if the 1st condition is false, then subsequent elif statements are checked. When an elif condition is found to be true, the statements following that associated parts are executed.

**Syntax**

```
if [ condition ]
then
Execute Statement if Condition 1
elif [ condition ]
Execute Statement if Condition 2
elif [ condition ]
Execute Statement if Condition 3
elif
Else Condition
fi
```

**#3) Check if an input number is positive, zero or negative:**

```
echo "Enter a number"
read num
if [ $num -gt 0 ]
then
echo "It is a positive number"
elif [ $num -eq 0 ]
then
echo "num is equal to zero"
else
echo "It is not a positive integer"
fi
```

**#4) Nested if**

if statement and else statement can be nested in bash shell programming. The keyword "fi" indicates the end of the inner if statement and all if statement should end with "fi".

**Syntax**

```
if [ condition ]
then
if [ condition ]
then
Execute Statement
elif
Execute Statement
fi
elif
Execute Statement
fi
```

**#4) Nested if Example**

```
echo "Enter Your Country:"
read cn
if [$cn -eq 'BHARATH']
then
echo "Enter Your State:"
read st
if [$st -gt 'Karnataka']
then
echo "Welcome to Karnataka"
elif
echo "You are Not Kannadiga"
fi
elif
echo "Other Country"
fi
```

**The Shell Switch Case**

The case statement is good alternative to multilevel if then else fi statement.it enables you to match several values against one variable. It's easier to read and write multiple conditions.

**Syntax**

```
case $[ variable_name ] in
value1)
Statement 1
;;
value2)
Statement 2
;;
value3)
Statement 3
;;
value4)
Statement 4
;;
```

```
valueN)
Statement N
;;
*)
Default Statement
;;
esac
```

Here, the value of the word expression is matched against each of the choice patterns. If a match is found then the corresponding statements are executed until the ';;' statement is encountered. If there is no match, the default statements under '\*' are executed.

**The following is an Example of a switch case program:**

```
echo "Enter a number"
read num
case $num in
[0-9])
echo "you have entered a single digit number"
;;
[1-9][1-9])
echo "you have entered a two-digit number"
;;
[1-9][1-9][1-9])
echo "you have entered a three-digit number"
;;
*)
echo "your entry does not match any of the conditions"
;;
esac
```

## **LOOPING OR ITERATIVE OR REPITITIVE STATEMENTS**

A loop is a powerful programming tool that enables you to execute a set of commands repeatedly. Loops are required whenever a set of statement must be executed repeatedly. The repeated execution also need decision making to terminate the loop.

- The three types of looping are
  - **until loop**
  - **while loop**
  - **for loop**

### **Until Loop**

The *until* statement executes every command inside the loop until the Boolean expression declared results to false. It is the complete opposite of the while statement. The until loop is useful when you need to execute a set of commands until a condition is true.

**Syntax**

```
until [ condition ]
do
  statement 1
  statement 2
done
```

**Example**

```
i=1
while [ !$i -lt 10 ]
do
  echo $i
  i=`expr $i + 1`
done
```

**While Loop**

The while loop enables you to execute set of commands repeatedly until some condition occurs. It is usually used when you need to manipulate the value of a variable repeatedly. The *while* repetitive control structure separates the initialization, Boolean test and the increment/decrement statement.

**Syntax**

```
while [ condition ]
do
  statement 1
  statement 2
done
```

**Example**

```
i = 1
while [ $i -le 10 ]
do
  echo $i
  i = `expr $i + 1`
done
```

## For Loop

A for loop is a bash programming language statement which allows code to be repeatedly executed. The *for* loop is a looping statement that uses the keyword *for* to declare a repetitive statement. The bash supports different syntaxes for the *for* loop statement:

### Syntax

```
for [ variable_name ] in ...
do
    statement 1
    statement 2
    statement n
done
```

This syntax starts with the keyword *for*, then followed by a variable name, the keyword *in* and the list of possible values for the variable. Each value in the list will be separated with a space and the start of the code lines that will be repeated is defined in the *do* and ends with a *done* keyword.

### Example

```
for no in {1..10}
do
    echo $no
done
```

## JUMPING CONTROLS

Most of the time your loops are going to through in a smooth and orderly manner. Sometimes however we may need to intervene and alter their running slightly. There are two statements we may issue to do this.

### Break

The **break** statement tells Bash to leave the loop straight away. It may be that there is a normal situation that should cause the loop to end but there are also exceptional situations in which it should end as well. For instance, maybe we are copying files but if the free disk space gets below a certain level, we should stop copying.

```
for value in $1/*
do
    used=$( df $1 | tail -1 | awk '{ print $5 }' | sed 's/%%//')
    if [ $used -gt 90 ]
    then
        echo Low disk space 1>&2
        break
    fi
```

```
cp $value $1/backup/  
done
```

---

### Continue

The **continue** statement tells Bash to stop running through this iteration of the loop and begin the next iteration. Sometimes there are circumstances that stop us from going any further. For instance, maybe we are using the loop to process a series of files but if we happen upon a file which we don't have the read permission for we should not try to process it.

```
#!/bin/bash  
# Make a backup set of files  
for value in $1/*  
do  
if [ ! -r $value ]  
then  
echo $value not readable 1>&2  
continue  
fi  
cp $value $1/backup/  
done
```

---

### Exit

The **exit** command terminates a script, just as in a C program. It can also return a value, which is available to the script's parent process.

Every command returns an *exit status* (sometimes referred to as a *return status* or *exit code*). A successful command returns a 0, while an unsuccessful one returns a non-zero value that usually can be interpreted as an *error code*. Well-behaved UNIX commands, programs, and utilities return a 0 exit code upon successful completion, though there are some exceptions.

Likewise, functions within a script and the script itself return an exit status. The last command executed in the function or script determines the exit status. Within a script, an **exit *nnn*** command may be used to deliver an *nnn* exit status to the shell (*nnn* must be an integer in the 0 - 255 range).

When a script ends with an **exit** that has no parameter, the exit status of the script is the exit status of the last command executed in the script (previous to the **exit**).

```
#!/bin/bash  
COMMAND_1  
...  
COMMAND_LAST  
# Will exit with status of last command.  
Exit
```

The equivalent of a bare **exit** is **exit \$?** or even just omitting the **exit**.

```
#!/bin/bash
COMMAND_1
...
COMMAND_LAST
# Will exit with status of last command.
exit $?

#!/bin/bash
COMMAND1
...
COMMAND_LAST
# Will exit with status of last command.
```

\$? reads the exit status of the last command executed. After a function returns, \$? gives the exit status of the last command executed in the function. This is Bash's way of giving functions a "return value."

Following the execution of a pipe, a \$? gives the exit status of the last command executed.

After a script terminates, a \$? from the command-line gives the exit status of the script, that is, the last command executed in the script, which is, by convention, 0 on success or an integer in the range 1 - 255 on error.

## UNIX SYSTEM COMMUNICATION

When you work in a distributed environment, you need to communicate with remote users and you need to access remote UNIX machines. A network lets two or more computers communicate and work together. Partly because of its open design, UNIX has been one of the operating systems where a lot of networking development is done. Just as there are different versions of UNIX, there are different ways and programs to use networks from UNIX.

There are several UNIX utilities that help users compute in a networked, distributed environment.

### The Internet

A worldwide network of computers. Internet users can transfer files, log into other computers, and use a wide range of programs and services.

### WWW

The World Wide Web is a fast-growing set of information servers on the Internet. The servers are linked into a hypertext web of documents, graphics, sound, and more. Point-and-click browser programs turn that hypertext into an easy-to-use Internet interface.

### Mail

A UNIX program that has been around for years, long before networking was common, is mail. It sends electronic memos, usually called email messages, between a user and one or more other users. When you send email, your message waits for the other



user(s) to start their own mail program. The people who get your message can file it, print it, reply to it, forward it to other people, and much more. System programs can send you mail to tell you about problems or give you information. You can send mail to programs, to ask them for information. Worldwide mailing lists connect users into discussion groups.

There is more, of course. There are zillions of mail programs for UNIX-some standard, some from vendors, and many freely available. The more common email programs include mailx , Pine , mush , elm , and MH (a package made up of many utilities including comp , inc , show , and so on). Find one that is right for you and use it!

### **ftp**

The ftp program is one way to transfer files between your computer and another computer with TCP/IP, often over the Internet network. ftp requires a username and password on the remote computer. Anonymous ftp ( 52.7 ) uses the ftp program and a special restricted account named anonymous on the remote computer. It's usually used for transferring freely available files and programs from central sites to users at many other computers.

### **UUCP**

UNIX-to-UNIX Copy is a family of programs ( uucp ( 52.7 ) , uux , uulog , and others) for transferring files and email between computers. UUCP is usually used with modems over telephone lines.

### **Usenet**

Usenet isn't exactly a network. It's a collection of thousands of computers worldwide that exchange files called news articles . This "net news" system has hundreds of interactive discussion groups, electronic bulletin boards, for discussing everything from technical topics to erotic art.

### **telnet**

This utility logs you into a remote computer over a network (such as the Internet) using TCP/IP. You can work on the remote computer as if it were your local computer. The telnet program is available on many operating systems; telnet can log you into other operating systems from your UNIX host and vice versa. A special version of telnet called tn3270 will log into IBM mainframes.

### **rlogin**

Similar to telnet but mostly used between UNIX systems. Special setups, including a file named .rhosts in your remote home directory, let you log into the remote computer without typing your password.

### **rcp**

A " remote cp " program for copying files between computers. It has the same command-line syntax as cp except that hostnames are added to the remote pathnames.

**rsh**

Starts a "remote shell" to run a command on a remote system without needing to log in interactively.

**NFS**

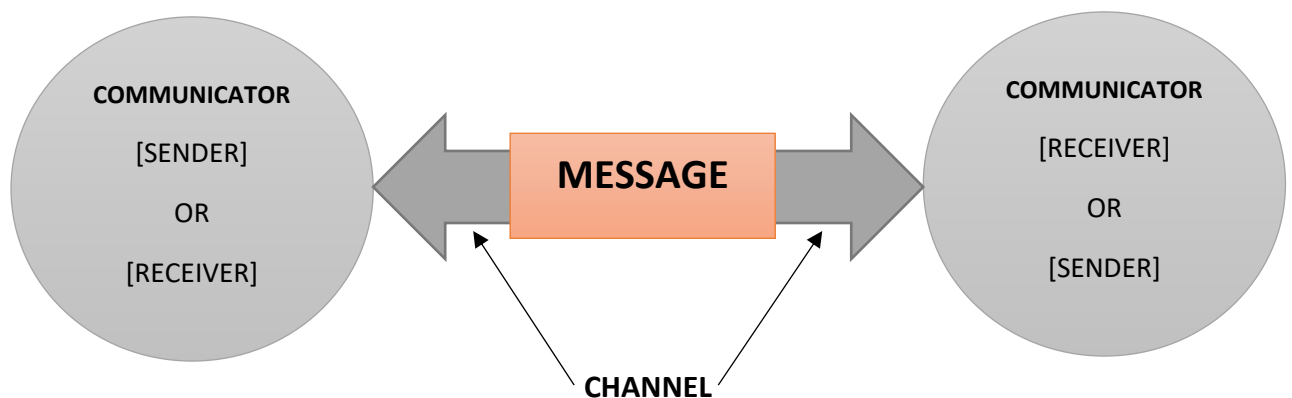
NFS is not a user utility. The Network File System and related packages like NIS (the Network Information Service) let your system administrator mount remote computers' file systems onto your local computer. You can use the remote file system as easily as if it were on your local computer.

**Write**

Sends messages to another user's screen. Two users can have a discussion with write.

**talk**

A more sophisticated program than write, talk splits the screen into two pieces and lets users type at the same time if they want to. Talk can be used over networks, though not all versions of talk can talk to one another.



Elements of a communication process

1. Sender
2. Receiver
3. Message
4. Channel

**The mesg command**

- UNIX facilitates users to other user's terminals who are logged in.
- This is possible only when the other terminal has given write permission.
- The mesg command is used to change the write permission of a user.

**Options:**

- y for yes
- n for no

**Example:**

- `$mesg y #grants write permission`
- `$mesg n #denies write permission`
- `$mesg is n #displays the current status of write permission of the terminal.`

**Write**

It is possible to send messages several ways in UNIX. Allows a two-way communication two users who are currently logged in and who have given write permission. To send a message directly to another user who is logged on, enter the command:

**write user\_name**

This starts a process that will **write** text to the other user's screen. Every line you enter will be sent to the other user, until you press `ctrl-d` to end the process. The other user can initiate a **write** command to reply to you, even while yours is still going on. This could be done immediately to notify the sender that you do not want to talk now.

If you don't want to be interrupted at all by messages, you can use the **mesg** command the letter **n** as an argument (**mesg n**) to turn off message reception. Using **y** as an argument will turn message reception back on.

**Example:****\$write MAAS**

Going on a tour? Happy journey.

-BCA

<ctrl d> #indicates end of message

Message from BCA@MAAS 14:43..... on pts/2 at

Going on a tour? Happy journey.

BCA

EOF

- Conversation continues until users decide to end it.
- Both users must be logged in else error message will appear.

**The finger command**

- Similar to `who` command.
- It lists the details of users who have logged in and given permission to accept messages.

**Example:**

`$finger`

Login Name Tty Idle Login time office office ph.

Rama rama pts/1 feb28 13:57

Uma uma.k pts/2 feb 28 14:21

### In this example

- Login shows login name of users
- Name shows full name of the users
- Tty shows device number of the terminals
- Idle shows idle time since user logged in
- Login time shows the time of logging in of the users.
- Office and office ph -> shows the address and phone number of the user.

### News command

The **news** command will read and display files that are placed in the /usr/news directory. This is another way for system operators to communicate information to you, and for users to communicate lengthy messages to all people who might want to read them. Options allow the user to specify what they want:

- -n lists new files in the news directory
- -a displays all files in the news directory
- -s counts the number of new files
- no option: displays only files that are new since you last ran the command

### wall command

The **wall** command means to *write all*, or to use the write command to all current users on the system. This command is very intrusive by nature and should not be used for trivial messages. It is immediate, and like write, it breaks into a user's current session. The command has some limitations. It will not be received by users who have message reception turned off. Also, it may not be accessible by anyone but the system administrator. If the idea of irritating all users does not bother you, you should know that the message is still tagged with your user name.

- This command can be used only by super user.
- Used to send message to all users on the system , known as broadcasting message to all users irrespective of whether user given write permission or not.
- Wall executable file is stored in /etc directory.

### Example:

```
$wall #message send by super user
```

```
There may be power failure.
```

```
Please save your files.
```

```
<ctrl d>
```

```
Broadcast message from root (pts/3) (sat09 14:37:28 2016): #DISPLAYED ON ALL LOGGED IN USERS.
```

```
There may be power failure.
```

```
Please save your files.
```

## Electronic mail

Sending and receiving messages using computer and communication tools is known as **Electronic Mail**. A more flexible and useful messaging process is **e-mail**, or electronic mail. E-mail can be sent to any number of users, it is not intrusive and the user need not be logged on at the time it is sent in order to see it later. While most environments now use third-party e-mail systems, you should be aware that there are e-mail functions built into UNIX.

### Sending mail

- The mail command is the basic e-mail program.
- Contains text editor to compose mail.
- Used to send and receive mails.

Depending on the system you use, either the **mail** or the **mailx** command may be available to you.

To use either e-mail command, you must have a mailbox on the system. This is a file that will hold your e-mail messages. This is your **system mailbox**. You can read mail in it with the **mailx** command, which will put a copy of that mail in a file called mbox, unless you save it elsewhere or exit **mailx** with the **x** command.

Like a lot of UNIX, you can save customized settings for your e-mail environment. **mailx** will look for these settings in a file called **.mailrc** in your home directory. This will override the general settings found in the **mail.rc** file.

Like using **write**, you can begin the **mailx** command by typing:

### Syntax:

```
$mail <options> addresses
message text
```

Example:

```
$mail user1 user2
Subject : seminar
```

```
.....
BCA<ctrl d>
```

- User1 and user2 are login names.
- If the receiver is not busy running a program, the following message will be displayed on his screen.
- You have new mail.
- If the user is not logged in when mail is sent to him/her, the message is displayed as you have mail.

This will begin a process that expects you to type a message to the named user. You end the message by entering ctrl-d, as the end of file marker. The message is then transmitted to the user's mail box.

Mail can be read by entering the **mailx** command with no argument. Your queue of messages will be displayed, and you can read a message by entering the message number displayed for it. When you wish to see the next message, enter its message number. Entering **q** will quit the **mailx** program and save messages you have read as noted above. Entering **x** will quit without saving opened mail to the mbox.

The **mailx** program has several commands available to the user in the input mode. Just to be different, these commands all start with a tilde. These commands are not mnemonic, and should be practiced to gain familiarity with them. We will do this in class.

## Receiving a mail

- The mail command without argument is used to receive mails.
- Example:

```
$mail
```

```
Mail version.....
```

```
"/var/spool/mail/rama":2messages 1 new 1
```

```
1. <uma> mon mar 03 10:40 labs>N
```

```
2.<std1> mon mar 04 12:32 projects &
```

- Received mails of a user are stored in a mailbox.
- Name of this will be his/her login name.
- Mailbox is found in /var/spool/mail directory.
- First line displays version of mail program
- Second line gives a summary of messages, with their status such as unread and new also indicates mail directory used and number of messages in it.
- Next list of mails are shown.
- First character on each line on gives status of each mail. Like new (N), unread (U), (>) character indicates that message as current message.
- The "&" character in the list line is the mail prompt
- Actions like reading, saving, deleting, forwarding and quitting the mail program.
- Personal mailbox called 'mbox' is located in user's directory.
- Any messages not deleted but read, will be saved in this file when user quits mail program.