MANIPAL INSTITUTE OF TECHNOLOGY
MANIPAL
(A constituent unit of MAHE, Manipal)

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

CERTIFICATE

This is to certify that Ms./Mr……..…….. Reg. No.: …..…… Section: ……..…… Roll No.: ……... has satisfactorily completed the lab exercises prescribed for OPEN SOURCE TECHNOLOGIES LAB [CSE 2164] of Second Year B. Tech. Degree at MIT, Manipal, in the academic year 2019-2020.

Date: …….....

Signature

Faculty in Charge

INDEX

Course Objectives

- Illustrate and explore the basic commands, shell scripting and system calls related to Linux operating system.
- To learn about Open Source programming and debugging tools
- To Learn about Open Source Documentation Tools

Course Outcomes

At the end of this course, students will have the

- Ability to execute Linux commands, shell scripting using appropriate Linux system calls.
- Ability to use development tools and debugging tools effectively
- Ability to develop simple web applications using open Source Technologies

Evaluation Plan

- Internal Assessment Marks : 60%
  - ✓ Continuous evaluation component (for each experiment): 10 marks
  - ✓ The assessment will depend on punctuality, program execution, maintaining the observation note and answering the questions in viva voce
  - ✓ Total marks of the 12 experiments will sum up to 60
- End semester assessment of 2 hour duration: 40 Marks

INSTRUCTIONS TO THE STUDENTS

Pre-Lab Session Instructions

1. Students should carry the Class notes, Lab Manual and the required stationery to every lab session.
2. Be in time and follow the Instructions from Lab Instructors.
3. Must Sign in the log register provided.
4. Make sure to occupy the allotted seat and answer the attendance.
5. Adhere to the rules and maintain the decorum.

In-Lab Session Instructions

- Follow the instructions on the allotted exercises given in Lab Manual.
- Show the program and results to the instructors on completion of experiments.
- On receiving approval from the instructor, copy the program and results in the Lab record.
- Prescribed textbooks and class notes can be kept ready for reference if required.

General Instructions for the exercises in Lab

- The programs should meet the following criteria:
  - Programs should be interactive with appropriate prompt messages, error messages if any, and descriptive messages for outputs.
  - Programs are properly indented and comments should be given whenever it is required.
  - Use meaningful names for variables and procedures.
- Plagiarism (copying from others) is strictly prohibited and would invite severe penalty during evaluation.
- The exercises for each week are divided under three sets:
  - Solved exercise
  - Lab exercises - to be completed during lab hours
  - Additional Exercises - to be completed outside the lab or in the lab to enhance the skill.
- In case a student misses a lab class, he/she must ensure that the experiment is completed at students end or in a repetition class (if available) with the permission of the faculty concerned but credit will be given only to one day's experiment(s).
- Questions for lab tests and examination are not necessarily limited to the questions in the manual, but may involve some variations and/or combinations of the questions.
- A sample note preparation is given later in the manual as a model for observation.

THE STUDENTS SHOULD NOT

- Carry mobile phones while working with computer.
- Go out of the lab without permission.

LAB NO.: 1 **Date:**

LINUX BASIC COMMANDS, SHELL CONCEPTS AND FILE FILTERS

Objectives:

In this lab, student will be able to:

1. **Learn Linux basic commands**
2. **Understand the working of commands and important shell concepts, file filters.**
3. **Write and execute basic commands in a Shell.**

shell

a utility program that enables the user to interact with the Linux operating system. Commands entered by the user are passed by the shell to the operating for execution. The results are then passed back by the shell and displayed on the user's display. There are several shells available like Bourne shell, C shell, Korn shell, etc. Each shell differs from the other in Command interpretation. The most popular shell is bash.

shell prompt

a character at the start of the command line which indicates that the shell is ready to receive the commands. The character is usually a '%' (percentage sign) or a '$' (dollar sign).

For. e.g.
Last login : Thu April 11 06:45:23
$ _ (This is the shell prompt, the cursor shown by the _ character).

Linux commands are executable binary files located in directories with the name bin (for binary). Many of the commands that are generally used are located in the directory /usr/bin.

echo is a command for displaying any string in the command prompt.
For e.g. $ echo "Welcome to MIT Manipal"
Environment variables: Shell has built in variables which are called environment variables. For e.g. the user who has logged in can be known by typing
$**echo** $USER
The above will display the current user's name.
When the command name is entered, the shell checks for the location of the command in each directory in the PATH environment variable. If the command is found in any of the directories mentioned in PATH, then it will execute. If not found, will give a message "Command not found".

COMMONLY USED LINUX COMMANDS

who: Unix is a system that can be concurrently used by multiple users and to know the users who are using the system can be known by a **who** command. For e.g. Current users are kumar, vipul and raghav. These are the user ids of the current users.

$ **who** [Enter]
kumar pts/10 May 1 09.32
vipul pts/4 May 1 09.32
raghav pts/5 May 1 09.32

The first columns indicates the user name of the user, second column indicates the terminal name and the third column indicates the login time. To know the user who has invoked the command can be known by the following command. For e.g. if kumar is the user who has typed the who command above then,

$ **who am i** [Enter]
kumar pts/10 May 1 09.32
ls: UNIX system has a large number of files that control its functioning and users also create files on their own. These files are stored in separate folders called directories. We can list the names of the files available in this directory with **ls** command. The list is displayed in the order of creation of files.
$ **ls** [Enter]
README
chap01
chap02
chap03
helpdir
progs

In the above output, **ls** displays a list of six files. We can also list specific files or directories by specifying the file name or directory names. In this we can use regular expressions.

For e.g. to list all files beginning with chap we can use the following command.
$ ls chap* [Enter]
chap01
chap02
chap03
To list further detailed information we can use **ls -l** command, where **-l** is an option between the command and filenames. The details include, file type, file or directory access permissions, number of links, owner name, group name, file or directory size, modification time and name of file or directory.
$ ls -l chap* [Enter]
-rw-r- - r- - 1 kumar users 5670 Apr 3 09.30 chap01
-rw-r- - r- - 1 kumar users 5670 Feb 23 09.30 chap02
-rw-r- - r- - 1 kumar users 5670 Apr 30 09.34 chap03
The argument beginning with hyphen is known as option. The main feature of option is it starts with hyphen. The command **ls** prints the columnar list of files and directories. With the **–l** option it displays all the information as shown above.
General syntax of **ls** command:

ls –[options][file list][directory list]

In Linux, file names beginning with period are hidden files, are not normally displayed in **ls** command. To display all files, including the hidden ones, use option **–a** in **ls** command as shown below:

$ ls -a

$ ls / will display the name of the files and sub-directories under the root directory.

pwd: This command gives the present working directory where the user is currently located.

$ **pwd**

/home/kumar/pis

cd: To move around in the file system use cd (change directory) command. When used with argument, it changes the current directory to the directory specified as argument, for instance:

$ **pwd**

/home/kumar

$ **cd** progs

$ **pwd**

$ /home/kumar/progs

cd .. : To change the working directory to the parent of the current directory we need to use

$ cd ..

.. (double dot) indicates parent directory. A single dot indicates current directory.

cat: **cat** is a multipurpose command. Using this we can display a file, create a file as well as concatenate files vertically.

$ **cat >** filename[Enter]

cat > os.txt

Welcome to Manipal. (This the content which will be placed in file with filename)

[Ctrl D] End of input

$_ (comes to the shell prompt)

The above command will create a file named os.txt in the current directory. To see the contents of the file.

$ **cat** os.txt[Enter]

Welcome to Manipal.

To display a file we can use **cat** command as shown above.

We can use **cat** for displaying more than one file, one after the other by listing the files after **cat**. For e.g.

$ cat os.txt lab.txt

will display os.txt followed with lab.txt

cp: To copy the contents of one file to another.

Syntax: **cp** sourcefilename targetfilename [Enter]

This command is also used to copy one or more files to a directory. The syntax of this form of **cp** command is

Syntax : **cp** filename(s) directoryname

If the file **os.txt** in current directory i.e. /home/kumar/pis needs to be copied into /home directory then it will be done as follows.

$ **cp** os.txt /home/ **OR** **$ cp** os.txt ../../

mv: This command renames or moves files. It has two distinct function: It renames a file or a directory and it moves a group of files to a different directory.

Syntax: **mv** oldfilename newfilename

Syntax of another form of this command is

mv file(s) directory

mv doesn't create a copy of the file, it merely renames it. No additional space is consumed on disk for the file after renaming. To rename the file chap01 to man01,

$ mv chap01 man01.

If the destination file doesn't exist, it will be created. For the above example, **mv** simply replaces the filename in the existing directory with the new name. By default **mv** doesn't prompt for overwriting the destination file if it exist.
The following command moves three files to the progs directory:
$ **mv** chap01 chap02 chap03 progs
mv can also be used to rename a directory for instance pis to pos:
$ **mv** pis pos
rm: This command deletes one or more files.
Syntax: **rm** filename

The following command deletes three files
$ **rm** chap01 chap02 chap03[Enter]

A file once deleted can be recovered subject to conditions by using additional software. **rm** won't normally remove a directory but it can remove files from one or more directories. It can remove two chapters from the progrs directory by using:
$ **rm** progrs/chap01 progrs/chap02

mkdir: Directories are created by **mkdir** command. The command is followed by the name of the directories to be created.
Syntax: **mkdir** directoryname
$ **mkdir** data [Enter]
This creates a directory named data under the current directory.
$ **mkdir** data dbs doc
The above command creates three directories with names data, dbs and doc.
rmdir : Directories are removed by **rmdir** command. The command is followed by the name of the directory to be removed. If a directory is not empty, then the directory will not be removed.
Syntax: **rmdir** directoryname
$ **rmdir** patch [Enter]
The command removes the directory by the name patch.

In Linux every file and directory has access permissions. Access permissions define which users have permission to access a file or directory. Permissions are three types, read, write and execute. Access permissions are defined for user, group and others.
For e.g. If access permission is only read for user, group and others, then it will be

r- -r--r- -

Access permissions can also be represented as a number. This number is in octal system. An access permission represented in numerical octal format is called absolute permission. The absolute permission for the above is

444

If the access permission is read, write for user, read, execute for group and only execute for others then it will be,

rw-r-x- -x

The absolute permission for the above is

651

chmod: changes the permission specified in the argument and leaves the other permissions unaltered. In this mode the following is the syntax.

Sytax: **chmod** category operation permission filename(s)

chmod takes as its argument an expression comprising some letters and symbols that completely describe the user category and the type of permission being assigned or removed. The expression contains three components:

User category (user, group, others)

The operation to be performed (assign or remove a permission). The type of permission (read, write and execute)

The abbreviations used for these three components are shown in Table 1.1.
E.g. to assign execute permission to the user of the file xstart;
$ **chmod u+x xstart**
$ ls –l xstart
- rwxr- - r- - 1 kumar metal 1980 May 01 20:30 xstart.

The command assigns (+) execute (x) permission to the user (u), but other permissions remain unchanged. Now the owner of the file can execute the file but the other categories i.e. group and others still can't. To enable all of them to execute this file:

$ **chmod ugo+x xstart**

$ **ls –l** xstart

- rwxr-x r- x 1 kumar metal 1980 May 01 20:30 xstart.

The string **ugo** combines all the three categories user, group and others. This command accepts multiple filenames in the command line:

$ **chmod u+x note note1 note3**
$ **chmod a-x, go+r xstart; ls –l xstart** (Two commands can be run simultaneously with ;)
- rw-r--rwx 1 kumar metal 1980 May 01 20:30 xstart.

Table 1.1: Abbreviations Used by chmod

Category	**Operation**	**Permission**
u- User	+ Assigns permission	r- Read permission
g- Group	- Removes permission	w- Write permission
o- Others	= Assigns absolute permission	x- Execute permission
a- All(ugo)		

Absolute Permissions:

Sometimes without needing to know what a file's current permissions the need to set all nine permission bits explicitly using **chmod** is done.

Read permission – 4 (Octal 100)

Write permission – 2 (Ocal 010)

Execute permission – 1 (Octal 001)

For instance, 6 represents read and write permissions, and 7 represents all permissions as can easily be understood from Table 1.2.

Table 1.2: Absolute Permissions

Binary	**Octal**	**Permissions**	**Significance**
000	0	---	No permissions
001	1	--x	Executable only
010	2	-w-	Writable only
011	3	-wx	Writable and executable

100	4	r--	Readable only
101	5	r-x	Readable and executable
110	6	rw-	Readable and writable
111	7	rwx	Readable, writable and executable

$ **chmod 666 xstart; ls –l xstart**
- rw-rw- rw - 1 kumar metal 1980 May 01 20:30 xstart.

The 6 indicates read and write permissions (4 + 2).

date: This displays the current date as maintained in the internal clock run perpetually.
$ **date** [Enter]

clear: The screen clears and the prompt and cursor are positioned at the top-left corner.
$ **clear [Enter]**

man: is used to display help file related to a command or system call.

Syntax: **man {command name/system call name}**

e.g. man date

man open

wc: displays a count of lines, words and characters in a file.

e.g. wc os.txt

1 3 19 os.txt

Syntax: **wc [-c | -m | -C] [-l] [-w] [file….]**

Options: The following options are supported:

-c Count bytes.

-m Count characters.

-C Same as – m,

-l Count lines

-w Count words delimited by white space characters or new line characters. If no option is specified the default is –lwc (count lines, words, and bytes).

Redirection Operators
For any program whether it is developed using C, C++ or Java, by default three streams are available known as input stream, output stream and error stream. In programming languages, to refer to them some symbolic names are used (i.e. they are system defined variables).

The following operators are the redirection operators

1. > standard output operator
> is the standard output operator which sends the output of any command into a file.

Syntax: command > file1

e.g. **ls > file1**

Output of the **ls** command is sent to a file1. First, file file1 is created if not exists otherwise, its content is erased and then output of the command is written.
E.g.: **cat file1 > file2**

Here, file2 get the content of file1.

E.g.: **cat file1 file2 file3 > file4**

This creates the file file4 which gets the content of all the files file1, file2 and file3 in order.

2. < standard input operator

< operator (standard input operator) allows a command to take necessary input from a file.

Syntax: **$ command < file**

E.g.: **cat<file1**

This displays output of file file1 on the screen.

E.g.: **cat <file1 >file2**

This makes cat command to take input from the file file1 and write its output to the file file2. That is, it works like a **cp** command.

3. >> appending operator

Similarly, >> operator can be used to append standard output of a command to a file.
E.g.: **command>>file1**
This makes, output of the given command to be appended to the file1. If the file1 doesn't exist, it will be created and then standard output is written.

4. << document operator

There are occasions when the data of your program reads is fixed and fairly limited. The shell uses the << symbols to read data from the same file containing the script. This is referred to as **here document**, signifying that the data is here rather than in a separate file. Any command using standard input can also take input from a here document.

Example.:

```
#!/bin/bash
cat <<DELIMITER
hello
this is a here
document
DELIMITER
```

This gives the output:
hello
this is a here
document

Shell Concepts

This section will describe some of the features that are common in all of the shells.

1. Wild-card: The metacharacters that are used to construct the generalized pattern for matching filenames belong to a category called wild-cards.

List of shell's wild-cards:

Wild-card	**Matches**
*	Any number of characters including none
?	A single or zero character
[ijk]	A single character - either an i, j or k
[x –z]	A single character between x and z
[!ijk]	A single character that is not an i, j or k.

[!x–z]	A single character not between x and z.
{pat1, pat2, ….}	pat1, pat2, etc.

Example: Consider a directory structure /home/kumar which have the following files:

README

chap01

chap02

chap03

helpdir

progs

Then with the below command the following output would be displayed.

$ **ls chap***
chap chap01 chap02 chap03
$ **ls .***
.bash_profile .exrc .netscape .profile

2. Pipes: Standard input and standard output constitute two separate streams that can be individually manipulated by the shell. If so then one command can take input from the other. This is possible with the help of pipes.

Assume if the **ls** command which produces the list of files, one file per line, use redirection to save this output to a file:

$ **ls > user.txt**

$ **cat user.txt**

The file shows the list of files.

Now to count the number of files:

$ **ls | wc – l**

The above command gives the number of files. This is how | (pipe) is used. There's no restriction on the number of commands to be used in pipe.

3. Command substitution: The shell enables the connecting of two commands in yet another way. While a pipe enables a command to obtain its standard input from the standard output of another command, the shell enables one or more command arguments to be obtained from the standard output of another command. This feature is called command substitution.

$ **echo The date today is `date`**
The date today is Sat May 6 19:01:56 IST 2019
$ **echo "There are total `ls | wc –l ` files and sub-directory in the current directory**
There are 15 files in the current directory.

4. Sequences: Two separate commands can be written in one line using ";" in sequences.
$ **chmod 666 xstart; ls –l xstart**

5. Conditional Sequences: The shell provides two operators that allow conditional execution - the && and ||, which typically have this syntax:

cmd1 && cmd2

cmd1 || cmd2

The && delimits two commands; the command cmd2 is executed only when cmd1 succeeds.

The || operator plays inverse role; the second command cmd2 is executed only when the first command cmd1 fails.

Note: All built-in shell commands returns non-zero if they fail. They return zero on success. e.g: if there is a program hello.c which displays 'Hello World' on compilation and execution. Then the following command in conditional sequences could be used to display the same:

$ **cc hello.c && ./a.out**

This command displays the output 'Hello World' if the compilation of the program succeeds. Similarly in case the compilation fails for the program the following output 'Error' could be displayed with the following command:

$ **cc hello.c || echo 'Error'**

File Filters commands in Linux:

1. head: To see the top 10 lines of a file - $ **head <file name>**
To see the top 5 lines of a file - $ **head -5 <file name>**
2. tail: To see last 10 lines of a file - $ **tail < file name>**
To see last 20 lines of a file - $ **tail -20 <file name>**
3. more: To see the contents of a file in the form of page views - $ more <file name>
$ **more f1.txt**

4. grep: To search a pattern of word in a file, **grep** command is used.

Syntax: **$ grep < word name> < file name>**
$ **grep hi file_1**
To search multiple words in a file
$ **grep -E 'word1|word2|word3' <file name>**
$ **grep -E 'hi|beyond|good' file_1**

5. sort: This command is used to sort the file.
$ **sort <file name>**
$ **sort file_1**
To sort the files in reverse order
$ **sort -r <file name>**

To display only files
$ ls –l | grep "^-"
To display only directories
$ ls –l | grep "^d"

Lab Exercises:

1. Write shell commands for the following.
   i) To create a directory in your home directory having 2 subdirectories.
   ii) In the first subdirectory, create 3 different files with different content in each of them.
   iii) Copy the first file from the first subdirectory to the second subdirectory.
   iv) Create one more file in the second subdirectory, which has the output of the number of users and number of files.
   v) To list all the files which starts with either a or A.

vi) To count the number of files in the current directory.
vii) Display the output if the compilation of a program succeeds.
viii) Count the number of lines in an input file.

2. Execute the following commands in sequence: i) date ii) ls iii) pwd

Additional Exercises:

1. Write shell commands for the following.
   i. To Display an error message if the compilation of a program fails.
   ii. To write a text block into a new file.
   iii. List all the files.
   iv. To count the number of users logged on to the system.

LAB NO.: 2 **Date:**

SHELL SCRIPTING – 1

Objectives:
In this lab, student will be able to:

1. Understand the importance of scripts.
2. Write and execute shell scripts.

The Linux shell is a program that handles interaction between the user and the system. Many of the commands that are typically thought of as making up the Linux system are provided by the shell. Commands can be saved as files called scripts, which can be executed like a program.

SHELL PROGRAMS: SCRIPTS
SYNTAX: **scriptname**
NOTE: A file that contains shell commands is called a script. Before a script can be run, it must be given execute permission by using **chmod** utility (chmod +x script). To run the script, only type its name. They are useful for storing commonly used sequences of commands to full-blown programs.
VARIABLES

Table 2.1 :Parameter Variables

$@	an individually quoted list of all the positional parameters
$#	the number of positional parameters
$!	the process ID of the last background command
$0	The name of the shell script.
$$	The process ID of the shell script, often used inside a script for generating unique temporary filenames; for example /tmp/tmpfile_$$.
$1, $2, …	The parameters given to the script.
$*	A list of all the parameters, in a single variable, separated by the first character in the environment variable IFS.

Lab Exercises:

1. Try the following shell commands
   $ echo $HOME, $PATH
   $ echo $MAIL
   $ echo $USER, $SHELL, $TERM
2. Try the following snippet, which illustrates the difference between local and environment variable:
   $ firstname=Rakesh ……local variables
   $ lastname=Sharma
   $ echo $firstname $lastname
   $ export lastname …..make “lastname” an envi var
   $ sh …..start a child shell

```
$ echo $firstname $lastname
$ ^D                              …..terminate child shell
$ echo $firstname $lastname
```

3. Try the following snippet, which illustrates the meaning of special local variables:

```
$ cat >script.sh
echo the name of this script is $0
echo the first argument is $1
echo a list of all the arguments is $*
echo this script places the date into a temporary file
echo called $1.$$
date > $1.$$          # redirect the output of date
ls $1.$$              # list the file
rm $1.$$              # remove the file
^D
$ chmod +x script.sh
$ ./script.sh Rahul Sachin Kumble
```

NOTE: A shell supports two kinds of variables: local and environment variables. Both hold data in a string format. The main difference between them is that when a shell invokes a subshell, the child shell gets a copy of its parent shell's environment variables, but not its local variables. Environment variables are therefore used for transmitting useful information between parent shells and their children.**Few predefined environment variables:**

$HOME	pathname of our home directory
$PATH	list of directories to search for commands
$MAIL	pathname of our mailbox
$USER	our username
$SHELL	pathname of our login shell
$TERM	type of the terminal

Creating a local variable:

variableName=value

Operations:

- Simple assignment and access
- Testing of a variable for existence
- Reading a variable from standard input
- Making a variable read only
- Exporting a local variable to the environment

Creating / Assigning a variable

Syntax: {name=value}

Example: $ firstName=Anand lastname=Sharma age=35

$ echo $firstname $lastname $age

$ name = Anand Sharma

$ echo $name

$ name = "Anand Sharma"

$ echo $name

Accessing variable:

Syntax: $name / ${name}

Example: $ verb=sing

$ echo I like $verbing

Reading a variable from standard input:

Syntax: read {variable}+

Example: $ cat > script.sh

echo “Please enter your name:”

read name

echo your name is $name

^D

Read-only variables:

Syntax: readonly {variable}+

Example: $ password=manipal

$ echo $password

$ readonly password

$ readonly .....list

$ password=mangalore

Running jobs in Background

A multitasking system lets a user do more than one job at a time. Since there can be only one job in foreground, the rest of the jobs have to run in the background. There are two ways of doing this: with the shell’s **& operator** and **nohup** command. The latter permits to log out while the jobs are running, but the former doesn’t allow that.

$ **sort –o emp.lst &**

550

The shell immediately returns a number the PID of the invoked command (550). The prompt is returned and the shell is ready to accept another command even though the previous command has not been terminated yet. The shell however remains the parent of the background process. Using an & many jobs can be run in background as the system load permits.

In the above case, if the shell which has started the background job is terminated, the background job will also be terminated. **nohup** is a command for running a job in background in which case the background job will not be terminated if the shell is close. nohup stands for no hang up.

e.g.

$ **nohup sort-o emp.lst &**

586

The shell returns the PID too. When the **nohup** command is run it sends the standard output of the command to the file **nohup.out**. Now the user can log out of the system without aborting the command.

JOB CONTROL

1. ps: ps is a command for listing processes. Every process in a system will have unique id called process id or PID. This command when used displays the process attributes.
$ **ps**

```
PID    TTY    TIME  CMD
291    console      0:00    bash
```

This command shows the PID, the terminal TTY with which the process is associated, the cumulative processor time that has been consumed since the process has started and the process name (CMD).

2. kill: This command sends a signal usually with the intention of killing one or more process. This command is an internal command in most shells. The command uses one or more PIDs as its arguments and by default sends the SIGTERM(15) signal. Thus:
$ **kill 105** terminates the job having PID 105. The command can take many PIDs at a time to be terminated.

3. sleep: This command makes the calling process sleep until the specified number of seconds or a signal arrives which is not ignored.

$ **sleep** 2

Lab Exercises:

1. Try the following, which illustrates the usage of **ps**:
   $ **(sleep 10; echo done) &**
   $ **ps**
2. Try the following, which illustrates the usage of **kill**:
   $ **(sleep 10; echo done) &**
   $ **kill pid** …..pid is the process id of background process
3. Try the following, which illustrates the usage of **wait**:
   $ **(sleep 10; echo done 1) &**
   $ **(sleep 10; echo done 2) &**
   $ **echo done 3; wait ; echo done 4** ….wait for children

NOTE: The following two utilities and one built-in command allow the listing controlling the current processes.

ps: generates a list of processes and their attributes, including their names, process ID numbers, controlling terminals, and owners

kill: allows to terminate a process on the basis of its ID number

wait: allows a shell to wait for one or all of its child processes to terminate

Sample Program:
$ cat>script.sh

echo there are $# command line arguments: $@

^D

$ script.sh arg1 arg2

Example:

```
#!/bin/sh
salutation="Hello"
echo $salutation
echo "The program $0 is now running"
echo "The second parameter was $2"
echo "The first parameter was $1"
echo "The parameter list was $*"
echo "The user's home directory is $HOME"
echo "Please enter a new greeting"
read salutation
echo $salutation
echo "The script is now complete"

exit 0
```

If we save the above shell script as try.sh, we get the following output:

```
$ ./try.sh foo bar baz
Hello
The program ./try.sh is now running
The second parameter was bar
The first parameter was foo
The parameter list was foo bar baz
The user's home directory is /home/rick
Please enter a new greeting
Sire
Sire
The script is now complete
$
```

Lab Exercises:

Write Shell Scripts to do the following:

1. List all the files under the given input directory, whose extension has only one character
2. Write a shell script that accepts two command line parameters. First parameter indicates the directory and the second parameter indicates a regular expression. The script should display all the files and directories in the directory specified in the first argument matching the format specified in the second argument.

3. Count the number of users logged on to the system. Display the output as Number of users logged into the system.
4. Count only the number of files in the current directory.
5. Write a shell script that takes two sorted numeric files as input and produces a single sorted numeric file without any duplicate contents.
6. Write a shell script that accepts two command line arguments. First argument indicates format of file and the second argument indicates the destination directory. The script should copy all the files as specified in the first argument to the location indicated by the second argument. Also, try the script where the destination directory name has space in it.

Additional Exercises:

1. Write Shell Scripts to do the following
   i) To list all the .c files in any given input subdirectory.
   ii) Write a script to include n different commands.

LAB NO.: 3 Date:

SHELL SCRIPTING – 2

Objectives:

In this lab, student will be able to

1. **Grasp the utility of the various variables in the Linux operating system.**
2. **Understand the different arithmetic and relational operators.**
3. **Understand the syntax and working of the various looping and decision statements.**
   The shell is not just a collection of commands but a really good programming language. A lot of tasks could be automated with it, along with this the shell is very good for system administration tasks. Many of the ideas could be easily tried with it thus making it as a very useful tool for simple prototyping and it is very useful for small utilities that perform some relatively simple tasks where efficiency is less important as compared to the ease of configuration, maintenance and portability.

COMMENTS

Comments in shell programming start with # and go until the end of the line.

List variables

Syntax: declare [-ax] [listname]

Example: $ declare –a teamnames

$ teamnames[0] = “India” …..assignment

$ teamnames[1] = “England”

$ teamnames[2] = “Nepal”

$ echo “There are ${#teamnames[*]} teams ….accessing

$ echo “They are: ${teamnames [*]}”

$ unset teamnames[1] …delete

Aliases

Allows to define your own commands

Syntax: alias [word[=string]]

Unalias [-a] {word}+

Example: $ alias dir=“ls –aF”

$ dir

ARITHMETIC

expr utility is s used for arithmetic operations. All of the components of expression must be separated by blanks, and all of the shell metacharacters must be escaped by a \.

Syntax: expr expression

Example: $ x=1

$ x=`expr $x +1`

$ echo $x

$ x=`expr 2 + 3 \* 5`

$echo $x

$echo `expr \( 4 \> 5 \)`

$echo `expr length "cat"`

$echo `expr substr "donkey" 4 3`

TEST EXPRESSION

Syntax: test expression

Table 3.1 :Forms of Test Expressions

Test	**Meaning**
!=	not equal
=	equal
-eq	equal
-gt	greater than
-ge	greater than or equal
-lt	less than
-le	less than or equal
!	logic negation
-a	logical and
-o	logical or
-r file	true if the file exists and is readable
-w file	true if the file exists and is writable
-x file	true if the file exists and is executable
-s file	true if the file exists and its size > 0
-d file	true if the file is a directory
-f file	true if the file is an ordinary file
-t filed	true if the file descriptor is associated with a terminal
-n str	true if the length of str is > 0
-z str	true if the length of str is zero

CONTROL STRUCTURES

(i) **The if** conditional

Syntax: **if** command1
then command2
fi

Example:

```
echo "enter a number:"
read number
if [ $number -lt 0 ]
then
echo "negative"
elif [ $number -eq 0 ]
then
echo "zero"
else
echo "positive"
fi
```

(ii) The **case** conditional

Syntax: **case** string **in**
pattern1) commands1 ;;
pattern2) commands2 ;;
……..
esac

case selectively executes statements if string matches a pattern. You can have any number of patterns and statements. Patterns can be literal text or wildcards. You can have multiple patterns separated by the "|" character.

Example:

```
case $1 in
*.c)
cc $1
;;
*.h | *.sh)
# do nothing
;;
```

The above example performs a compile if the filename ends in .c, does nothing for files ending in .h or .sh. else it writes to stdout that the file is an unknown type. Note that the: character is a NULL command to the shell (similar to a comment field).

```
case $1 in
[AaBbCc])
option=0
;;
*)
option=1
;;
esac
echo $option
```

In the above example, if the parameter $1 matches A, B or C (uppercase or lowercase), the shell variable *option* is assigned the value 0, else is assigned the value 1.

(iii) **while**: looping

Syntax: **while** *condition is true*
do
commands

done

Example 1:

```
# menu program
echo “menu test program”
stop=0
while test $stop  -eq 0
do
cat << ENDOFMENU
1: print the date
2,3     : print the current working directory
4: exit
ENDOFMENU
echo
echo “your choice ?”
read reply
echo
case $reply in
   “1”)
         date
           ;;
   “2” | “3”)
         pwd
           ;;
    “4”)
        stop =1
           ;;
     *)
        echo “illegal choice”
           ;;
  esac
done
```

Example 2:

```
#!/bin/bash
X=0
while [ $X -le 20 ]
do
         echo $X
         X=$(X+1))
done
# echo all the command line arguments
while test $# != 0
do
        echo $1
        #The shift command shifts arguments to the left
        shift
done
```

(iv) **until:** Looping

Syntax: **until** *command-list1*
do

command-list2
done

Example:

```
x=1
until [ $x –gt 3  ]
do
echo x = $x
x=`expr $x + 1`
done
```

(v) **for:** Looping

Syntax: **for** *variable* in *list*
do
command-list
done

Sample Program

```
homedir=`pwd`
for files in /*
do
echo $files
done
cd $homedir
```

The above example lists the names of all files under / (the root directory)

Lab Exercises:

Write shell scripts to perform the following

1. Find whether the given number is even or odd.
2. Print the first 'n' odd numbers.
3. Find all the possible quadratic equation roots using case.
4. Find the factorial of a given number.

Additional Exercises

1. Write Shell scripts to do the following
   i) Find whether the given string is palindrome.
   ii) Find out the sum of the numbers given by user.

LAB NO.: 4 **Date:**

PROGRAMMING TOOLS IN LINUX-I

Objectives:

- **To Learn about using gcc C Compiler, Header files, Static and shared Library files**
- **The make command and makefiles**

The C Compiler

On POSIX-compliant systems, the C compiler is called c89. Historically, the C compiler was simply called cc. Over the years, different vendors have sold UNIX-like systems with C compilers with different facilities and options, but often still called cc.

When the POSIX standard was prepared, it was impossible to define a standard cc command with which all these vendors would be compatible. Instead, the committee decided to create a new standard command for the C compiler, c89. When this command is present, it will always take the same options, independent of the machine.

On Linux systems that do try to implement the standards, you might find that any or all of the commands c89, cc, and gcc refer to the system C compiler, usually the GNU C compiler, or gcc. On UNIX systems, the C compiler is almost always called cc.

We use gcc because it's provided with Linux distributions and because it supports the ANSI standard syntax for C.

Try it out: Your First Linux C Program

In this example you start developing for Linux using C by writing, compiling, and running your first Linux program. It might as well be that most famous of all starting points, Hello World.

1. Here's the source code for the file hello.c:

```
#include <stdio.h>
#include <stdlib.h>

int main()
  {
        printf("Hello World\n");
        exit(0);
  }
```

2. Now compile, link, and run your program.

```
$ gcc -o hello hello.c
$ ./hello
  Hello World
$
```

How It Works?

You invoked the GNU C compiler (on Linux this will most likely be available as cc too) that translated the C source code into an executable file called hello. You ran the program and it printed a greeting. If this did not work for you, make sure that the C compiler is installed on your system. For example, many Linux distributions have an install option called Software Development (or something similar) that you should select to make sure the necessary packages are installed.

Because this is the first program you've run, it's a good time to point out some basics. The hello program will probably be in your home directory. If *PATH* doesn't include a reference to your home directory, the shell won't be able to find hello. Furthermore, if one of the directories in *PATH* contains another program called hello, that program will be executed instead. This would also happen if such a directory is mentioned in *PATH* before your home directory. To get around this potential problem, you can prefix program names with *./* (for example, ***./hello***). This specifically instructs the shell to execute the program in the current directory with the given name. (The dot is an alias for the current directory.)

If you forget the ***-o*** name option that tells the compiler where to place the executable, the compiler will place the program in a file called ***a.out*** (meaning assembler output).

Header Files

For programming in C and other languages, you need header files to provide definitions of constants and declarations for system and library function calls. For C, these are almost always located in ***/usr/include*** and subdirectories thereof. You can normally find header files that depend on the particular incarnation of Linux that you are running in ***/usr/include/sys*** and ***/usr/include/linux.***

Other programming systems will also have header files that are stored in directories that get searched automatically by the appropriate compiler. Examples include ***/usr/include/X11*** for the X Window System and ***/usr/include/c++*** for GNU C++.

You can use header files in subdirectories or nonstandard places by specifying the -I flag (for include) to the C compiler. For example,

$ ***gcc -I/usr/openwin/include fred.c***

will direct the compiler to look in the directory ***/usr/openwin/include*** , as well as the standard places, for header files included in the ***fred.c*** program. Refer to the manual page for the C compiler (man gcc) for more details.

Library Files

Libraries are collections of precompiled functions that have been written to be reusable. Typically, they consist of sets of related functions to perform a common task. Examples include libraries of screen-handling functions (the curses and ncurses libraries) and database access routines (the dbm library).

Standard system libraries are usually stored in ***/lib*** and ***/rus/lib***. The C compiler (or more exactly, the linker) needs to be told which libraries to search, because by default it searches only the standard C library. This is a remnant of the days when computers were slow and CPU cycles were expensive. It's not enough to put a library in the standard directory and hope that the compiler will find it; libraries need to follow a very specific naming convention and need to be mentioned on the command line.

A library filename always starts with lib. Then follows the part indicating what library this is (like c for the C library, or m for the mathematical library). The last part of the name starts with a dot (.), and specifies the type of the library:

- ***.a*** for traditional, static libraries
- ***.so*** for shared libraries (see the following)

The libraries usually exist in both static and shared formats, as a quick ***ls /usr/lib*** will show. You can instruct the compiler to search a library either by giving it the full path name or by using the ***-l*** flag. For example,

$ ***gcc -o fred fred.c /usr/lib/libm.a***

tells the compiler to compile file ***fred.c***, call the resulting program file ***fred***, and search the mathematical library in addition to the standard C library to resolve references to functions. A similar result is achieved with the following command:

$ ***gcc -o fred fred.c -lm***

The ***-lm*** (no space betwen the l and the m) is shorthand (shorthand is much valued in UNIX circles) for the library called ***libm.a*** in one of the standard library directories (in this case ***/usr/lib***). An additional advantage of the ***-lm*** notation is that the compiler will automatically choose the shared library when it exists.

Although libraries are usually found in standard places in the same way as header files, you can add to the search directories by using the ***-L*** (uppercase letter) flag to the compiler. For example,
$ ***gcc -o x11fred -L/usr/openwin/lib x11fred.c -lX11***
will compile and link a program called ***x11fred*** using the version of the library ***libX11*** found in the ***/usr/openwin/lib*** directory.

Static Libraries

The simplest form of library is just a collection of object files kept together in a ready-to-use form. When a program needs to use a function stored in the library, it includes a header file that declares the function. The compiler and linker take care of combining the program code and the library into a single executable program. You must use the ***–l*** option to indicate which libraries other than the standard C runtime library are required.

Static libraries, also known as archives, conventionally have names that end with .a. Examples are ***/usr/lib/libc.a*** and ***/usr/lib/libX11.a*** for the standard C library and the X11 library, respectively.
You can create and maintain your own static libraries very easily by using the ar (for archive) program and compiling functions separately with ***gcc -c***. Try to keep functions in separate source files as much as possible. If functions need access to common data, you can place them in the same source file and use *static variables* declared in that file.

Try It Out: Static Libraries

In this example, you create your own small library containing two functions and then use one of them in an example program. The functions are called fred and bill and just print greetings.

1. First, create separate source files (imaginatively called fred.c and bill.c) for each function.
Here's the first:

```
#include <stdio.h>
void fred(int arg)
{
  printf("fred: we passed %d\n", arg);
}
```

And here's the second:

```
#include <stdio.h>

void bill(char *arg)
{
  printf("bill: we passed %s\n", arg);
}
```

2. You can compile these functions individually to produce object files ready for inclusion into a library. Do this by invoking the C compiler with the ***-c*** option, which prevents the compiler from trying to create a complete program. Trying to create a complete program would fail because you haven't defined a function called ***main***.

```
$ gcc -c bill.c fred.c
$ ls *.o
bill.o  fred.o
```

3. Now write a program that calls the function bill. First, it's a good idea to create a header file for your library. This will declare the functions in your library and should be included by all applications that want to use your library. It's a good idea to include the header file in the files ***fred.c*** and ***bill.c*** too. This will help the compiler pick up any errors.

```
/* This is lib.h. It declares the functions fred and bill for users */
void bill(char *);
void fred(int);
```

4. The calling program (***program.c***) can be very simple. It includes the library header file and calls one of the functions from the library.

```
#include <stdlib.h>
#include "lib.h"

int main()
{
bill("Hello World");
exit(0);
}
```

5. You can now compile the program and test it. For now, specify the object files explicitly to the compiler, asking it to compile your file and link it with the previously compiled object module ***bill.o***

```
$ gcc -c program.c
$ gcc -o program program.o bill.o
$ ./program
bill: we passed Hello World
$
```

6. Now you'll create and use a library. Use the ar program to create the archive and add your object files to it. The program is called ar because it creates archives, or collections, of individual files placed together in one large file. Note that you can also use ar to create archives of files of any type. (Like many UNIX utilities, ar is a generic tool.)

```
$ ar crv libfoo.a bill.o fred.o
a - bill.o
a - fred.o
```

7. The library is created and the two object files added. To use the library successfully, some systems, notably those derived from Berkeley UNIX, require that a table of contents be created for the library. Do this with the ranlib command. In Linux, this step isn't necessary (but it is harm-

less) when you're using the GNU software development tools.
$ ***ranlib libfoo.a***
Your library is now ready to use. You can add to the list of files to be used by the compiler to create your program like this:
$ ***gcc -o program program.o libfoo.a***
$ ***./program***
bill: we passed Hello World
$
You could also use the –l option to access the library, but because it is not in any of the standard places, you have to tell the compiler where to find it by using the **–L** option like this:
$ ***gcc –o program program.o –L. –lfoo***
The ***–L.*** option tells the compiler to look in the current directory **(.)** for libraries. The ***–lfoo*** option tells the compiler to use a library called ***libfoo.a*** (or a shared library, l***ibfoo.so***, if one is present). To see which functions are included in an object file, library, or executable program, you can use the ***nm*** command. If you take a look at program and ***lib.a***, you see that the library contains both ***fred*** and ***bill***, but that program contains only ***bill***. When the program is created, it includes only functions from the library that it actually needs. Including the header file, which contains declarations for all of the functions in the library, doesn't cause the entire library to be included in the final program.

Shared Libraries

One disadvantage of static libraries is that when you run many applications at the same time and they all use functions from the same library, you may end up with many copies of the same functions in memory and indeed many copies in the program files themselves. This can consume a large amount of valuable memory and disk space.

Many UNIX systems and Linux-support shared libraries can overcome this disadvantage. A complete discussion of shared libraries and their implementation on different systems is beyond the scope of this book, so we'll restrict ourselves to the visible implementation under Linux.

Shared libraries are stored in the same places as static libraries, but shared libraries have a different filename suffix. On a typical Linux system, the shared version of the standard math library is /lib/libm.so.

When a program uses a shared library, it is linked in such a way that it doesn't contain function code itself, but references to shared code that will be made available at run time. When the resulting program is loaded into memory to be executed, the function references are resolved and calls are made to the shared library, which will be loaded into memory if needed.

In this way, the system can arrange for a single copy of a shared library to be used by many applications at once and stored just once on the disk. An additional benefit is that the shared library can be updated independently of the applications that rely on it. Symbolic links from the /lib/libm.so file to the actual library revision (/lib/libm.so.N where N represents a major version number — 6 at the time of writing) are used. When Linux starts an application, it can take into account the version of a library required by the application to prevent major new versions of a library from breaking older applications.

For Linux systems, the program (the dynamic loader) that takes care of loading shared libraries and resolving client program function references is called ld.so and may be made available as ld-linux.so.2 orld-lsb.so.2 or ld-lsb.so.3. The additional locations searched for shared libraries are configured in the file /etc/ld.so.conf, which needs to be processed by ldconfig if changed (for example, if X11 shared libraries are added when the X Window System is installed).
$ ***ldd program***
linux-gate.so.1 => (0xffffe000)

libc.so.6 => /lib/libc.so.6 (0xb7db4000)
/lib/ld-linux.so.2 (0xb7efc000)

In this case, you see that the standard C library (***libc***) is shared (.so). The program requires major Version 6. Other UNIX systems will make similar arrangements for access to shared libraries.

Experiment:

1. In C, write a program to implement a stack with push, pop operations using suitable functions. Create static libraries for various operations on stack. Create a header file for function declarations .
(Use vim to edit code. Note down various vim commands useful in editing source codes.).

The make command and makefiles

Problems of Multiple Source F iles

When they're writing small programs, many people simply rebuild their application after edits by recompiling all the files. However, with larger programs, some problems with this simple approach become apparent. The time for the edit-compile-test cycle will grow. Even the most patient programmer will want to avoid recompiling all the files when only one file has been changed.

A potentially much more difficult problem arises when multiple header files are created and included in different source files. Suppose you have header files a.h, b.h, and c.h, and C source files main.c, 2.c, and 3.c. (We hope that you choose better names than these for real projects!)
You could have the following situation:

```
/* main.c */
#include "a.h"
...
/* 2.c */
#include "a.h"
#include "b.h"
...
/* 3.c */
#include "b.h"
#include "c.h"
...
```

If the programmer changes c.h, the files main.c and 2.c don't need to be recompiled, because they don't depend on this header file. The file 3.c does depend on c.h and should therefore be recompiled if c.h is changed. However, if b.h was changed and the programmer forgot to recompile 2.c, the resulting program might no longer function correctly.
The make utility can solve both of these problems by ensuring that all the files affected by changes are recompiled when necessary.

The make Command and Makefiles

Although, as you will see, the make command has a lot of built-in knowledge, it can't know how to build your application all by itself. You must provide a file that tells make how your application is constructed. This file is called the makefile.

The makefile most often resides in the same directory as the other source files for the project. You can have many different makefiles on your machine at any one time. Indeed, if you have a large project, you may choose to manage it using separate makefiles for different parts of the

project. The combination of the make command and a makefile provides a very powerful tool for managing projects. It's often used not only to control the compilation of source code, but also to prepare manual pages and to install the application into a target directory.

The Syntax of Makefiles

A makefile consists of a set of dependencies and rules. A dependency has a target (a file to be created) and a set of source files upon which it is dependent. The rules describe how to create the target from the dependent files. Typically, the target is a single executable file.

The makefile is read by the make command, which determines the target file or files that are to be made and then compares the dates and times of the source files to decide which rules need to be invoked to construct the target. Often, other intermediate targets have to be created before the final target can be made. The make command uses the makefile to determine the order in which the targets have to be made and the correct sequence of rules to invoke.

Options and Parameters to make

The make program itself has several options. The three most commonly used are

- -k, which tells make to keep going when an error is found, rather than stopping as soon as the first problem is detected. You can use this, for example, to find out in one go which source files fail to compile.
- -n, which tells make to print out what it would have done without actually doing it.
- -f <filename>, which allows you to tell make which file to use as its makefile. If you don't use this option, the standard version of make looks first for a file called makefile in the current directory. If that doesn't exist, it looks for a file called Makefile. However if you are using GNU Make, which you probably are on Linux, that version of make looks for GNUmakefile first, before searching for makefile and then Makefile. By convention, many Linux programmers use Makefile; this allows the makefile to appear first in a directory listing of a directory filled with lowercase-named files. *We suggest you don't use the name GNUmakefile, because it is specific to the GNU implementation of make.*

To tell make to build a particular target, which is usually an executable file, you can pass the target name to make as a parameter. If you don't, make will try to make the first target listed in the makefile. Many programmers specify all as the first target in their makefile and then list the other targets as being dependencies for all. This convention makes it clear which target the makefile should attempt to build by default when no target is specified. We suggest you stick to this convention.

Dependencies

The dependencies specify how each file in the final application relates to the source files. In the programming example shown earlier in the chapter, you might specify dependencies that say your final application requires (depends on) main.o, 2.o, and 3.o; and likewise for main.o (main.c and a.h); 2.o (2.c,a.h, and b.h); and 3.o (3.c, b.h, and c.h). Thus main.o is affected by changes to main.c and a.h, and it needs to be re-created by recompiling main.c if either of these two files changes.

In a makefile, you write these rules by writing the name of the target, a colon, spaces or tabs, and then a space- or tab-separated list of files that are used to create the target file. The dependency list for the earlier example is

myapp: main.o 2.o 3.o
main.o: main.c a.h
2.o: 2.c a.h b.h
3.o: 3.c b.h c.h

This says that myapp depends on main.o, 2.o, and 3.o, main.o depends on main.c and a.h, and so on. This set of dependencies gives a hierarchy showing how the source files relate to one

other. You can see quite easily that, if b.h changes, you need to revise both 2.o and 3.o, and because 2.o and 3.o will have changed, you also need to rebuild myapp.

If you want to make several files, you can use the phony target all. Suppose your application consisted of both the binary file myapp and a manual page, myapp.1. You could specify this with the line

all: myapp myapp.1

Again, if you don't specify an all target, make simply creates the first target it finds in the makefile.

Rules

The second part of the makefile specifies the rules that describe how to create a target. In the example in the previous section, what command should be used after the make command has determined that 2.o needs rebuilding? It may be that simply using gcc -c 2.c is sufficient (and, as you'll see later, make does indeed know many default rules), but what if you needed to specify an include directory, or set the symbolic information option for later debugging? You can do this by specifying explicit rules in the makefile.

At this point, we have to clue you in to a very strange and unfortunate syntax of makefiles: the difference between a space and a tab. All rules must be on lines that start with a tab; a space won't do. Because several spaces and a tab look much the same and because almost everywhere else in Linux programming there's little distinction between spaces and tabs, this can cause problems. Also, a space at the end of a line in the makefile may cause a make command to fail. However, it's an accident of history and there are far too many makefiles in existence to contemplate changing it now, so take care! Fortunately, it's usually reasonably obvious when the make command isn't working because a tab is missing.

Try It Out : A Simple Makefile

Most rules consist of a simple command that could have been typed on the command line. For the example, you create your first makefile, ***Makefile1****:*

```
myapp: main.o 2.o 3.o
gcc -o myapp main.o 2.o 3.o

main.o: main.c a.h
        gcc -c main.c

2.o: 2.c a.h b.h
    gcc -c 2.c

3.o: 3.c b.h c.h
     gcc -c 3.c
```

You invoke the make command with the -f option because your makefile doesn't have either of the usual default names of ***makefile*** or ***Makefile***. If you invoke this code in a directory containing no source code, you get this message:

```
$ make -f Makefile1
make: *** No rule to make target 'main.c', needed by 'main.o'.
Stop.
$
```

The make command has assumed that the first target in the makefile, myapp, is the file that you want to create. It has then looked at the other dependencies and, in particular, has determined that a file called ***main.c*** is needed. Because you haven't created this file yet and the makefile does not say how it might be created, make has reported an error. So now create the source files and try again. Because you're not interested in the result, these files can be very simple. The header files are actually empty, so you can create them with touch:

```
$ touch a.h
```

$ ***touch b.h***
$ touch c.h

main.c contains main, which calls *function_two* and *function_three*. The other two files define *function_two* and *function_three*. The source files have *#include* lines for the appropriate headers, so they appear to be dependent on the contents of the included headers. It's not much of an application, but here are the listings:

```
/* main.c */
#include <stdlib.h>
#include "a.h"
extern void function_two();
extern void function_three();
int{
}
main()
function_two();
function_three();
exit (EXIT_SUCCESS);

/* 2.c */
#include "a.h"
#include "b.h"
void function_two() {
}

/* 3.c */
#include "b.h"
#include "c.h"
void function_three() {
}
```

Now try make again:

```
$ make  -f Makefile1
        gcc -c  main.c
        gcc -c  2.c
        gcc -c  3.c
        gcc -o  myapp main.o 2.o 3.o
$
```

This is a successful make.

How It Works

The make command has processed the dependencies section of the makefile and determined the files that need to be created and in which order. Even though you listed how to create myapp first, make has determined the correct order for creating the files. It has then invoked the appropriate commands you gave it in the rules section for creating those files. The make command displays the commands as it executes them. You can now test your makefile to see whether it handles changes to the file b.h correctly:

```
$ touch b.h
$ make -f Makefile1
        gcc -c 2.c
        gcc -c 3.c
        gcc -o myapp main.o 2.o 3.o
$
```

The make command has read your makefile, determined the minimum number of commands required to rebuild myapp, and carried them out in the correct order. Now see what happens if you delete an object file:

$ ***rm 2.o***
$ ***make -f Makefile1***
gcc -c 2.c
gcc -o myapp main.o 2.o 3.o
$
Again, make correctly determines the actions required

Comments in a Makefile

A comment in a makefile starts with # and continues to the end of the line. As in C source files, comments in a makefile can help both the author and others to understand what was intended when the file was written.

Macros in a Makefile

Even if this was all there was to make and makefiles, they would be powerful tools for managing multiple source file projects. However, they would also tend to be large and inflexible for projects consisting of a very large number of files. Makefiles therefore allow you to use macros so that you can write them in a more generalized form.

You define a macro in a makefile by writing MACRONAME=value, then accessing the value of MACRONAME by writing either $(MACRONAME) or ${MACRONAME}. Some versions of make may also accept $MACRONAME. You can set the value of a macro to blank (which expands to nothing) by leaving the rest of the line after the = blank.

Macros are often used in makefiles for options to the compiler. Often, while an application is being developed, it will be compiled with no optimization, but with debugging information included. For a release version the opposite is usually needed: a small binary with no debugging information that runs as fast as possible.

Another problem with Makefile1 is that it assumes the compiler is called gcc. On other UNIX systems, you might be using cc or c89. If you ever wanted to take your makefile to a different version of UNIX, or even if you obtained a different compiler to use on your existing system, you would have to change several lines of your makefile to make it work. Macros are a good way of collecting all these system dependent parts, making it easy to change them.

Macros are normally defined inside the makefile itself, but they can be specified by calling make with the macro definition, for example, make CC=c89. Command-line definitions like this override defines in the makefile. When used outside makefiles, macro definitions must be passed as a single argument, so either avoid spaces or use quotes like this: make “CC = c89 “ .

Try It Out : A Makefile with Macros

Here’s a revised version of the makefile, Makefile2, using some macros:

```
all: myapp
# Which compiler
CC = gcc

# Where are include files kept
INCLUDE = .
```

```
# Options for development
CFLAGS = -g -Wall –ansi

# Options for release
# CFLAGS = -O -Wall –ansi

myapp: main.o 2.o 3.o
$(CC) -o myapp main.o 2.o 3.o
        main.o: main.c a.h
$(CC) -I$(INCLUDE) $(CFLAGS) -c main.c
        2.o: 2.c a.h b.h
$(CC) -I$(INCLUDE) $(CFLAGS) -c 2.c
        3.o: 3.c b.h c.h
$(CC) -I$(INCLUDE) $(CFLAGS) -c 3.c
```

If you delete your old installation and create a new one with this new makefile, you get

```
$ rm *.o myapp
$ make -f Makefile2
gcc -I. -g -Wall -ansi -c main.c
gcc -I. -g -Wall -ansi -c 2.c
gcc -I. -g -Wall -ansi -c 3.c
gcc -o myapp main.o 2.o 3.o
$
```

How It Works

The make program replaces macro references $(CC), $(CFLAGS), and $(INCLUDE) with the appropriate definitions, rather like the C compiler does with #define. Now if you want to change the compile command, you need to change only a single line of the makefile.

In fact, make has several special internal macros that you can use to make makefiles even more succinct.

We list the more common ones in the following table; you will see them in use in later examples. Each of these macros is only expanded just before it's used, so the meaning of the macro may vary as the makefile progresses. In fact, these macros would be of very little use if they didn't work this way.

Macro	Definition
$?	List of prerequisites (files the target depends on) changed more recently than the current target
$@	Name of the current target
$<	Name of the current prerequisite
$*	Name of the current prerequisite, without any suffix

There are two other useful special characters you may see in a makefile, preceding a command:

- - tells make to ignore any errors. For example, if you wanted to make a directory but wished to ignore any errors, perhaps because the directory might already exist, you just precede mkdir with a minus sign. You will see - in use a bit later in this chapter.
- @ tells make not to print the command to standard output before executing it. This character is handy if you want to use echo to display some instructions.

LAB NO.: 5 **Date:**

PROGRAMMING TOOLS IN LINUX-II

Objectives:

- **To Learn about Debugging tools like gdb,DDD**
- ***T*o Learn about Version Control like git, svn.**

Debugging tools

Every significant piece of software will contain defects, typically two to five per 100 lines of code. These mistakes lead to programs and libraries that don't perform as required, often causing a program to behave differently than it's supposed to. Bug tracking, identification, and removal can consume a large amount of a programmer's time during software development.

Types of Errors

A bug usually arises from one of a small number of causes, each of which suggests a method of detection and removal:

Specification Errors: If a program is incorrectly specified, it will inevitably fail to perform as required. Even the best programmer in the world can sometimes write the wrong program. Before you start programming (or designing), make sure that you know and understand clearly what your program needs to do. You can detect and remove many (if not all) specification errors by reviewing the requirements and agreeing that they are correct with those who will use the program.

Design Errors: Programs of any size need to be designed before they're created. It's not usually enough to sit down at a computer keyboard, type source code directly, and expect the program to work the first time. Take time to think about how you will construct the program, what data structures you'll need, and how they will be used. Try to work out the details in advance, because it can save many rewrites later on.

Coding Errors: Of course, everyone makes typing errors. Creating the source code from your design is an imperfect process. This is where many bugs will creep in. When you're faced with a bug in a program, don't overlook the possibility of simply rereading the source code or asking someone else to. It's surprising just how many bugs you can detect and remove by talking through the implementation with someone else.

The purpose of a debugger is to allow you to see what is going on inside your C program while it runs. In addition, you can use gdb to see what your program was doing at the moment it crashed.

GDB:

Gdb is a debugger for C. It is the default debugger with most of the unix systems. It allows you to do things like run the program up to a certain point then stop and print out the values of certain variables at that point, or step through the program one line at a time and print out the values of

each variable after executing each line. It uses a command line interface. To prepare your program for debugging with gdb, you must compile it with the -g flag.

$ gcc -g -o helloworld helloworld.c
(helloworld is executable and helloworld.c is source code file)

Start gdb:
$ gdb *helloworld*

Gdb will give you a prompt that looks like this: (gdb). From that prompt you can run your program, look at variables, etc., using the commands listed below (and others not listed).

To exit from the gdb prompt type quit or simply q.

GDB COMMANDS :

help
Gdb provides online documentation. Just typing help will give you a list of topics. Then you can type help *topic* to get information about that topic (or it will give you more specific terms that you can ask for help about). Or you can just type help *command* and get information about any other command.

file
file *executable* specifies which program you want to debug.

run
run will start the program running under gdb. (The program that starts will be the one that you have previously selected with the file command, or on the unix command line when you started gdb. You can give command line arguments to your program on the gdb command line the same way you would on the unix command line, except that you are saying run instead of the program name:
run 2048 24 4
You can even do input/output redirection: run > outfile.txt.

break
A ``breakpoint'' is a spot in your program where you would like to temporarily stop execution in order to check the values of variables, or to try to find out where the program is crashing, etc. To set a breakpoint you use the break command.

break *function* sets the breakpoint at the beginning of *function*. If your code is in multiple files, you might need to specify *filename:function*.

break *linenumber* or break *filename:linenumber* sets the breakpoint to the given line number in the source file. Execution will stop before that line has been executed.

delete
delete will delete all breakpoints that you have set.

delete *number* will delete breakpoint numbered *number*. You can find out what number each breakpoint is by doing info breakpoints. (The command info can also be used to find out a lot of other stuff. Do help info for more information.)

clear

clear *function* will delete the breakpoint set at that function. Similarly for *linenumber*, *filename:function*, and *filename:linenumber*.

continue
continue will set the program running again, after you have stopped it at a breakpoint.

step
step will go ahead and execute the current source line, and then stop execution again before the next source line.

next
next will continue until the next source line in the current function (actually, the current innermost stack frame, to be precise). This is similar to step, except that if the line about to be executed is a function call, then that function call will be completely executed before execution stops again, whereas with step execution will stop at the first line of the function that is called.

until
until is like next, except that if you are at the end of a loop, until will continue execution until the loop is exited, whereas next will just take you back up to the beginning of the loop. This is convenient if you want to see what happens after the loop, but don't want to step through every iteration.

list
list *linenumber* will print out some lines from the source code around *linenumber*. If you give it the argument *function* it will print out lines from the beginning of that function. Just list without any arguments will print out the lines just after the lines that you printed out with the previous list command.

print
print *expression* will print out the value of the expression, which could be just a variable name. To print out the first 25 (for example) values in an array called list, do
print list[0]@25

Try it out: Exercise

Following is the buggy code (Located on https://github.com/rohitmit/GDBExercise/blob/master/FirstExample.c)

```
#include<stdio.h>
int main(){
        int num;
        do{
                printf("Enter a positive integer");
                scanf("%d",&num);
        } while(num<0);

        int factorial, i;
        for (i = 1; i<=num; i++){
                factorial = factorial *i;
```

```
    }
    printf("%d! = %d\n", num, factorial);
    return 0;
}
```

The output of this program is wrong. So, we will use gdb to debug it. As gdb operates on executable files, therefore:

1. Run in gdb environment:

$ gcc -g -o FirstExample FirstExample.c
$ gdb FirstExample
GNU gdb (GDB) Red Hat Enterprise Linux (7.2-92.el6)
Copyright (C) 2010 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. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/cseuser10/testgit/debugging/FirstExample...done.
(gdb) run
Starting program: /home/cseuser10/testgit/debugging/FirstExample
Enter a positive integer4
4! = 0

Program exited normally.
Missing separate debuginfos, use: debuginfo-install glibc-2.12-1.209.el6_9.2.x86_64

2. As from the output, it is clearly visible that the program is giving wrong output. Therefore, we will be analysing the program by introducing "Breakpoint". As we are not sure where to put breakpoint, we will put it at the main

(gdb) break main
Breakpoint 1 at 0x40053c: file FirstExample.c, line 6.
(gdb) run
Starting program: /home/cseuser10/testgit/debugging/FirstExample

Breakpoint 1, main () at FirstExample.c:6
6 printf("Enter a positive integer");
(gdb) next
7 scanf("%d",&num);

The breakpoint will halt the program at the main, then to continue we give run command and for line by line execution next command will be used.

(gdb) next
Enter a positive integer4
8 } while(num<0);

This suggests that we are at the end of do-while loop.

3. List command will show the code in gdb prompt

```
(gdb) list
3       int main(){
4               int num;
5               do{
6                       printf("Enter a positive integer");
7                       scanf("%d",&num);
8               } while(num<0);
9
10              int factorial, i;
11              for (i = 1; i<=num; i++){
12                      factorial = factorial *i;
(gdb) list
13              }
14              printf("%d! = %d\n", num, factorial);
15              return 0;
16      }
```

4. We will check the value of variable whose input we have given using print command

```
(gdb) next
11              for (i = 1; i<=num; i++){
(gdb) print num
$1 = 4
(gdb) next
12                      factorial = factorial *i;
(gdb) next
11              for (i = 1; i<=num; i++){
(gdb) next
12                      factorial = factorial *i;
(gdb) next
11              for (i = 1; i<=num; i++){
(gdb) next
12                      factorial = factorial *i;
(gdb) next
11              for (i = 1; i<=num; i++){
(gdb) next
12                      factorial = factorial *i;
(gdb) next
11              for (i = 1; i<=num; i++){
(gdb) next
14              printf("%d! = %d\n", num, factorial);
(gdb) next
4! = 0
15              return 0;
(gdb) print factorial
$2 = 0
```

5. As the values of num is correct but factorial is having wrong value. Hence, we will rerun the code and set a new breakpoint after the do-while loop. **When working with multiple files give the file name along with line number to set the breakpoint.**

(gdb) break FirstExample.c:10
Breakpoint 2 at 0x40058d: file FirstExample.c, line 10.

6. We again run the program in gdb and till second break point. Here, we can see the values of variables

(gdb) next
Breakpoint 2, main () at FirstExample.c:11
11 for (i = 1; i<=num; i++){
(gdb) print i
$1 = 0
(gdb) next
12 factorial = factorial *i;
(gdb) print i
$2 = 1
(gdb) print factorial
$3 = 0
The value of factorial is coming out as 0. That is giving the wrong answer. (In your case, it may be any garbage value). Info locals is another command to see the values of variables.

(gdb) info locals
num = 4
factorial = 0
i = 1

7. We can try the debugging with various values of a variable using print command. Here, as factorial variable was not initialized and giving incorrect output. Therefore, we can quickfix the rest of the execution as:

(gdb) print factorial = 1
$4 = 1

8. Check for the rest of the program to check if the final output is correct.

(gdb) n
12 factorial = factorial *i;
(gdb) n
11 for (i = 1; i<=num; i++){
(gdb) n
12 factorial = factorial *i;
(gdb) n
11 for (i = 1; i<=num; i++){
(gdb) n
12 factorial = factorial *i;
(gdb) n
11 for (i = 1; i<=num; i++){
(gdb) n
14 printf("%d! = %d\n", num, factorial);
(gdb) n
4! = 24
15 return 0;
(gdb) info locals
num = 4
factorial = 24

i = 5
(gdb) next
16 }
(gdb) next
0x000000397001ed1d in __libc_start_main () from /lib64/libc.so.6
(gdb) next
Single stepping until exit from function __libc_start_main,
which has no line number information.

Program exited normally.

The presented is one of the initial basic way to debug your program using gdb. Try and explore various other gdb commands for debugging.

We can list the process as:
Step 1. Compile the C program with debugging option -g
Step 2. Launch gdb
Step 3. Set up a **break** point inside C program
Step 4. Execute (**run**) the C program in gdb debugger
Step 5. **Print**ing the variable values inside gdb debugger
Step 6. Continue, stepping over and in – gdb commands (using continue, next, step)

Version Control:

Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later. For the example you will use software source code as the files being version controlled, though in reality you can do this with nearly any type of file on a computer. In other words, versioning is the process of assigning either unique version names or unique version numbers to unique states of computer software.

If you are a graphic or web designer and want to keep every version of an image or layout (which you would most certainly want to), a Version Control System (VCS) is a very wise thing to use. It allows you to revert selected files back to a previous state, revert the entire project back to a previous state, compare changes over time, see who last modified something that might be causing a problem, who introduced an issue and when, and more. Using a VCS also generally means that if you screw things up or lose files, you can easily recover. In addition, you get all this for very little overhead.

Local Version Control Systems

Many people's version-control method of choice is to copy files into another directory (perhaps a time-stamped directory, if they're clever). This approach is very common because it is so simple, but it is also incredibly error prone. It is easy to forget which directory you're in and accidentally write to the wrong file or copy over files you don't mean to.

To deal with this issue, programmers long ago developed local VCSs that had a simple database that kept all the changes to files under revision control.

Centralized Version Control Systems

The next major issue that people encounter is that they need to collaborate with developers on other systems. To deal with this problem, Centralized Version Control Systems (CVCSs) were developed. These systems (such as CVS, Subversion, and Perforce) have a single server that contains all the versioned files, and a number of clients that check out files from that central place. For many years, this has been the standard for version control.

This setup offers many advantages, especially over local VCSs. For example, everyone knows to a certain degree what everyone else on the project is doing. Administrators have fine-grained control over who can do what, and it's far easier to administer a CVCS than it is to deal with local databases on every client.

Distributed Version Control Systems

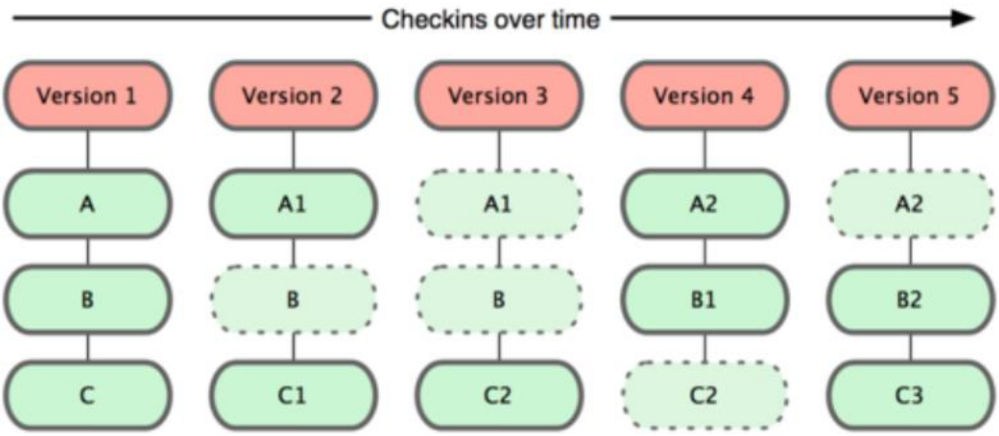
This is where Distributed Version Control Systems (DVCSs) step in. In a DVCS (such as Git, Mercurial, Bazaar or Darcs), clients don't just check out the latest snapshot of the files; rather, they fully mirror the repository, including its full history. Thus, if any server dies, and these systems were collaborating via that server, any of the client repositories can be copied back up to the server to restore it. Every clone is really a full backup of all the data.

GIT

Git is a distributed version-control system for tracking changes in source code during software development. It is designed for coordinating work among programmers, but it can be used to track changes in any set of files. Its goals include speed, data integrity, and support for distributed, non-linear workflows. Git was created by Linus Torvalds in 2005 for development of the Linux kernel, with other kernel developers contributing to its initial development. Its current maintainer since 2005 is Junio Hamano. As with most other distributed version-control systems, and unlike most client–server systems, every Git directory on every computer is a full-fledged repository with complete history and full version-tracking abilities, independent of network access or a central server. Git is free and open-source software distributed under the terms of the GNU General Public License version 2.

Git keeps "snapshots" of the entire state of the project. With Git, every time you commit, or save the state of your project, Git basically takes a picture of what all your files look like at that moment and stores a reference to that snapshot. To be efficient, if files have not changed, Git doesn't store the file again, just a link to the previous identical file it has already stored. Git thinks about its data more like a stream of snapshots.

–Each checkin version of the overall code has a copy of each file in it.
–Some files change on a given checkin, some do not.
–More redundancy, but faster.

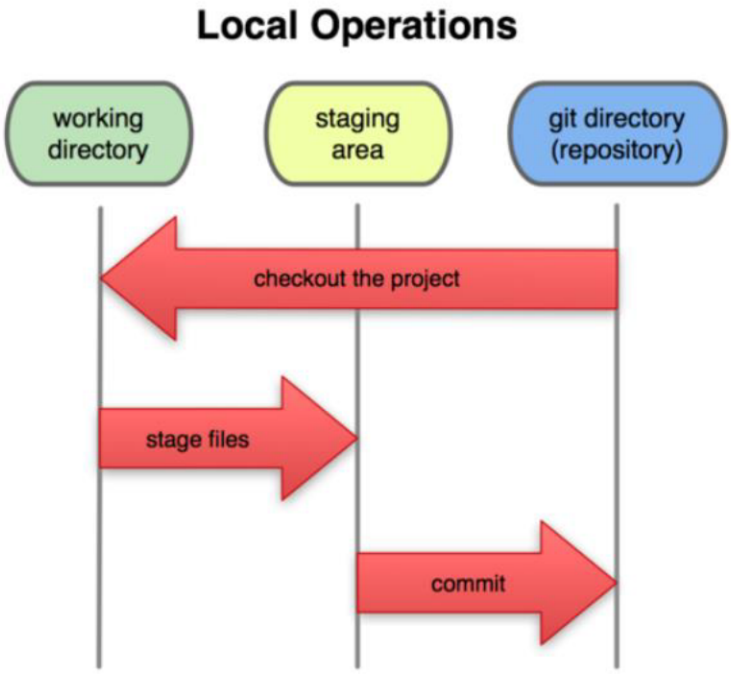


Three States of Git:

Git has three main states that your files can reside in: committed, modified, and staged:

- Committed means that the data is safely stored in your local database.
- Modified means that you have changed the file but have not committed it to your database yet.
- Staged means that you have marked a modified file in its current version to go into your next commit snapshot.

This leads us to the three main sections of a Git project: the Git directory, the working tree, and the staging area.



The Git directory is where Git stores the metadata and object database for your project. This is the most important part of Git, and it is what is copied when you *clone* a repository from another computer.

The working tree is a single checkout of one version of the project. These files are pulled out of the compressed database in the Git directory and placed on disk for you to use or modify.

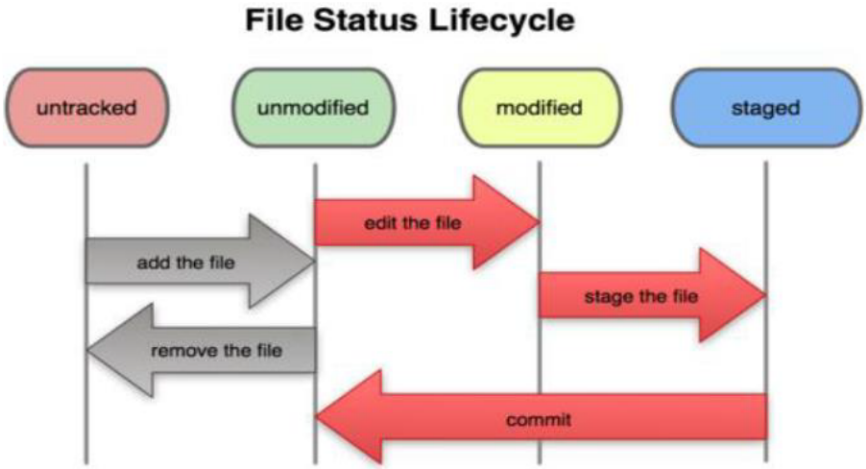
The staging area is a file, generally contained in your Git directory, that stores information about what will go into your next commit. Its technical name in Git parlance is the "index", but the phrase "staging area" works just as well.

The basic Git workflow goes something like this:

1. You modify files in your working tree.

2. You selectively stage just those changes you want to be part of your next commit, which adds *only* those changes to the staging area.

3. You do a commit, which takes the files as they are in the staging area and stores that snapshot permanently to your Git directory.



Initializing a Git Repository in an Existing Directory:

```
$ cd /home/user/my_project
$ git init
```

This creates a new subdirectory named .git that contains all of your necessary repository files — a Git repository skeleton.

```
$ git --version
```

To know git version.

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

The main tool you use to determine which files are in which state is the git status command. Default branch is "master". This means you have a clean working directory; in other words, none of your tracked files are modified. Git also doesn't see any untracked files, or they would be listed here. Short status can be viewed by $ git status -s

Start version-controlling existing files:

```
$ git add helloworld.c
```

$ git add README
$ git commit -m 'initial project version'

git add commands that specify the files you want to track (here some c program helloworld.c and a README file), followed by a git commit ('initial project version' is a message or summary of commit).

Exercises:

1. Write a simple C program (HelloWorld.c) that takes a number as an input and print the same number as an output. Now create a new git repository (check the contents of .git directory).
2. Check the status of the repository. Add the file and commit the changes.
   (Note: You can add all the changes file for commit using $ git add -A)
3. Modify your C program (HelloWorld.c) such that it takes a number as input and prints a new number after adding 1 to the input. Check the changes that you have made using $ git diff
4. Add this file to your git repository and commit. Check the logs of git commit using
   $ git log
5. Unmodifying a Modified File:
   a. Again, modify your C program (HelloWorld.c) such that now it adds 2 to the input before printing it. Check the git status.
   b. Later you can unmodify the program by $ git checkout -- HelloWorld.c
   (CAREFUL this deletes the local changes in the tracked file)
   c. Check git status again.
6. Checking old commit:
   a. Each commit operation generates a commit ID (you can see using git log)
   commit 7973e54895bdfb7c226952bd612fec81055305e6
   Author: cseuser10 <cseuser10@param-cse.(none)>
   Date: Mon May 6 17:08:24 2019 +0530
   b. Run $ git show 7973e54895bdfb7c226952bd612fec81055305e6 (Your commit ID).

Cloning an Existing Remote Repository

You clone a repository with git clone <url>. For example, if you want to clone the Git linkable library called libgit2, you can do so like this:

$*git clone https://github.com/rohitmit/OSTLab*

If you want to clone the repository into a directory named something other than libgit2, you can specify the new directory name as an additional argument:

$*git clone https://github.com/rohitmit/OSTLab myOSTLab*

Try it out :Adding Remote Repositories

1. First create an account on https://github.com/ and login with new login credentials.
2. Create a new public repository named "MyFirstRepo" by clicking on new and leaving all other details to default.
3. Go to terminal of your linux system. Go to your project directory and put "HelloWorld.c" here and its "a.out".
4. Create the local git repository using $ git init add the code files to the repository and commit in local repository.
5. Now remote add and push local code to remote repository using:
   $ git remote add origin https://<USER_NAME>@github.com/<USER NAME>/MyFirstRepo.git
   $ git push -u origin master

6. You can add multiple collaborator to your github repository and work simultaneously. To update your local code-base you can pull from remote repository using:
   $ git pull

Show the existing remotes

To see the details of the remotes, e.g., the URL use the following command.

```
$ git remote
origin
$ git remote -v
github  https://github.com/rohitmit/OSTLab.git (fetch)
github  https://github.com/rohitmit/OSTLab.git (push)
origin  https://github.com/rohitmit/OSTLab (fetch)
origin  https://github.com/rohitmit/OSTLab (push)
```

Pushing to Your Remotes

When you have your project at a point that you want to share, you have to push it upstream. The command for this is simple: git push <remote> <branch>. If you want to push your master branch to your origin server (again, cloning generally sets up both of those names for you automatically), then you can run this to push any commits you've done back up to the server:

```
$ git push origin master
```

LAB NO.: 6 Date:

INTRODUCTION TO LATEX

Objectives:

In this lab, student will be able to:

1. Use Latex to create documents
2. Draw Tables and insert figures in a document using Latex
3. Reference tables and figures in the text

- **INTRODUCTION**

LATEX (pronounced lay-tek) is a document preparation system for producing professional-looking documents without having to worry about formatting, typesetting, alignments, references, etc. It is not a word processor. LATEX is a scripting language, this means that you will write in a script and then execute it to generate a .pdf file. It is available as free software for most operating systems.

A LATEX document is a plain text file with a .tex extension. It can be typed in a simple text editor such as Notepad, but most people find it easier to use a dedicated LATEX editor. As you type you mark the document structure (title, chapters, subheadings, lists etc.) with tags. When the document is finished, you compile it and convert it into another format. Several different output formats are available, but probably the most useful is Portable Document Format (PDF).

- **INSTALLING A LATEX EDITOR**

Before writing in LATEX one has to install the required libraries for the computer to understand the scripts. This is accomplished by installing MiKTEX. After installing MiKTEX you will now need to install a suitable LATEX editor (a shell program) to write the LATEX scripts. The editor used here is TeXStudio, this is an open-source cross-platform editor. Other TEX editors like TeXmacs and TeXshop, for macintosh, exist. TeXStudio can be freely downloaded and installed for any operating system.

- **STRUCTURE OF A DOCUMENT**

The documents in LaTeX are structured slightly different from Word. A document consists of two parts, the preamble and the main document.

Preamble: The purpose of the preamble is to tell LaTeX the kind of document you will set up and the packages you are going to need. A package is a set of additional functions such as *amsmath* for additional math formatting. For example, the preamble looks like this:

You can set the class of the *documentclass* with the *documentclass* command and add packages with the *usepackage* command. Only the *documentclass* command is mandatory, you can compile a document also without packages. The *\documentclass* command must appear at the start of every LATEX document. The text in the curly brackets specifies the document class. The *article* document class is suitable for shorter documents such as journal articles and short reports. Other document classes include *report* (for longer documents with chapters, e.g. PhD theses), *proc* (conference proceedings), *book* and *slides*. The text in the square brackets specifies options -- in this case it sets the paper size to A4 and the main font size to 12pt.

Main document: The main document is contained within the document environment like this:

- **STARTING A NEW DOCUMENT**

1. Start TeXstudio.
   A new document will automatically open.
2. Type the following:

```
\documentclass[a4paper,12pt]{article}
    \begin{document}
        A sentence of text.
    \end{document}
```

3. Click on the Save button.

4. Create a new folder called LaTeX course in Libraries>Documents.

5. Name your document Doc1 and save it as a TeX document in this folder.

It is a good idea to keep the LATEX documents in a separate folder as the compiling process creates multiple files.

6. Make sure the typeset menu is set to pdfLaTeX.

7. Click on the Typeset button.

There will be a pause while the document is being converted to a PDF file. When the compiling is complete TeXStudio PDF viewer will open and display the document. The PDF file is automatically saved in the same folder as the .tex

- **CREATING A TITLE**

The *\maketitle* command creates a title. The title of the document should be specified after the *\title* command. If the date is not specified today's date is used. Author is optional.

1. Type the following directly after the *\begin{document}* command:

```
\title{My First Document}
\author{My Name}
\date{\today}
\maketitle
```

2. Click on the Typeset button and check the PDF.

Points to note:

1. *\today* is a command that inserts today's date. You can also type in a different date, for example *\date{November 2013}*.
2. Article documents start the text immediately below the title on the same page. Reports put the title on a separate page (like this workbook).

- **SECTIONS**

To divide the document into chapters (if needed), the following sectioning commands are available for the article class:

- *\section{...}*
- *\subsection{...}*
- *\subsubsection{...}*
- *\paragraph{...}*
- *\subparagraph{...}*

The title of the section replaces the dots between the curly brackets. With the report and book classes we also have \chapter{...}.

1. Replace "A sentence of text." with the following:

```
\section{Introduction}
This is the introduction.

\section{Methods}
\subsection{Stage 1}
The first part of the methods.
\subsection{Stage 2}
The second part of the methods.
```

\section{Results}
Here are my results.

2. Click on the Typeset button and check the PDF.

- **TABLE OF CONTENTS**

It is very easy to generate a table of contents, when sectioning commands are used.

Type *\tableofcontents* where you want the table of contents to appear in your document often directly after the title page.

You may also want to change the page numbering so that roman numerals (i, ii, iii) are used for pages before the main document starts. This will also ensure that the main document starts on page 1. Page numbering can be switched between arabic and roman using *\pagenumbering{...}*.

1. Type the following on a new line below *\maketitle*:

\pagenumbering{roman}
\tableofcontents
\newpage
\pagenumbering{arabic}

The *\newpage* command inserts a page break so that we can see the effect of the page numbering commands.

2. Click on the Typeset button and check the PDF file.

- **LABELLING AND REFERENCING**

Any of the sectioning commands can be labelled so that they can be referred to in other parts of the document. Label the section with *\label{labelname}*.

Then type *\ref{labelname}* or *\pageref{labelname}*, when you want to refer to the section or page number of the label.

1. Type *\label{sec1}* on a new line directly below *\subsection{Stage 1}*.
2. Type Referring to section *\ref{sec1}* on page *\pageref{sec1}* in the Results section.
3. Click on the Typeset button and check the PDF. You may need to typeset the document twice before the references appear in the PDF.

- **LISTS**

Ordered and unordered lists can be created by using the *enumerate* and *itemize* commands, respectively.

For example:

Unordered list:
\begin{itemize}
\item Apple
\item Orange
\end{itemize}

Output

Ordered list:
\begin{enumerate}
\item Apple
\item Orange
\end{enumerate}

- **FIGURES**

To include figures in a document the package graphicx is required, so the command \usepackage{graphicx} must be written in the preamble. To add a figure, one can write the following commands.

```
\begin{figure}[h] % between [] you put options e.g. ht = here top
\centering
\includegraphics[width=0.5\textwidth]{directory/to/file.png}
\caption{This figure shows a plot of x vs. y}
\label{Cross-reference_key}
\end{figure}
```

1. [h] is the placement specifier. h means keep the figure approximately here (if it will fit). Other options are t (at the top of the page), b (at the bottom of the page) and p (on a separate page for figures). You can also add !, which overrides the rule LATEX uses for choosing where to put the figure, and makes it more likely it will put it where you want.
2. \centering centres the image on the page, if not used images are left-aligned by default.
3. \includegraphics[]{} is the command that resizes, transforms and positions the image. There are several ways of resizing an image (width, height or scale), the particular command width=0.5\textwidth resizes the image to 50 % of the text width.
4. The command \label{Cross-reference_key} labels the figure using the key written between {}. This can be used to refer to the figure number by simply writing \ref{Cross-reference_key} anywhere in the document.

- **TABLES**

The tabular command is used to typeset tables. By default, LATEX tables are drawn without horizontal and vertical lines | you need to specify if you want lines drawn. LATEX determines the width of the columns automatically.

This code starts a table:

```
\begin{tabular}{...}
```

Where the dots between the curly brackets are replaced by code defining the columns:

- l for a column of left-aligned text.
- r for a column of right-aligned text.
- c for a column of centre-aligned text.
- | for a vertical line.

The table data follows the \begin command:

- & is placed between columns.
- \\ is placed at the end of a row (to start a new one).
- \hline inserts a horizontal line.
- \cline{2-3} inserts a partial horizontal line between column 2 and column 3.

The command \end{tabular} finishes the table.

For example:

```
\begin{tabular}{r|cl}
    1st Column & 2nd Column & 3rd Column\\
    \hline
    a & b & c\\
    \cline{2-3}
    d & e & f\\
\end{tabular}
```

Output:

1st Column	2nd Column	3rd Column
a	b	c
d	e	f

11.1 MultiColumn

The command *\multicolumn* allows user to combine multiple columns.

Format: *\multicolumn{num}{col}{text}*

The value *num* in the above command specifies the number of columns to be combined. The argument *col* must contain a position symbol *l, r o c.*

An example is provided below:

```
\ begin { tabular }{ llc }
\ hline %
\ multicolumn {2}{ c }{ Sample } & Roughness \\
        & & ( nm )\\
                \ hline %
        A & ring & 385\\
                \ cline {2 -3}
        & plate & 397\\
                \ hline
        B & ring & 376\\
                \ cline {2 -3}
        & plate & 390\\
                \ hline
\ end { tabular }
```

Output:

Sample		Roughness (nm)
A	Ring	385
	plate	397
B	Ring	376
	plate	390

- **Multirow**

As *\multicolumn* allows to have cells on more than one column, the *\multirow* command allows to have cells on more than one row. This command requires the package *multirow*.

\multirow can be used in two different ways:

\multirow{n}{Sample}* creates a cell that contains the text *Sample* and extends on *n* rows and has an undefined width;

\multirow{n}{larg}{ Sample }* creates a cell that contains the text *Sample* and extends on *n* rows and has an width equal to *larg*;

The following code is an example to create a table with multirow cells.

```
\begin {tabular}{ llcccc }
        \toprule %
```

```
\multirow {2}{*}{Name} & \multirow {2}{*}{Course} & \multicolumn
{3}{c}{Marks} & Total \\
\cmidrule (l) {3 -5}
& & Test1 & Test2 & Test3 & \\
\toprule %
\multirow {2}*{Alice} & C1 & 10 & 9 & 8 & 27\\
\cmidrule (l ){2 -6}
& C2 & 9 & 9 & 7 & 25\\
\midrule
\multirow {2}*{Bob} & C1 & 7 & 5 & 9 & 21\\
\cmidrule (l ){2 -6}
& C2 & 8 & 8 & 8 & 24\\
\bottomrule
\end {tabular}
```

Output:

Name	Course	Marks			Total
		Test1	Test2	Test3	
Alice	C1	10	9	8	27
	C2	9	9	7	25
Bob	C1	7	5	9	21
	C2	8	8	8	24

Note: Include package *booktabs* to use the commands \bottomrule, \toprule, \midrule and \cmidrule.

- **Font sizes and styles**

LATEX normally chooses the appropriate font and font size based on the logical structure of the document (e.g. sections). In some cases, you may want to set fonts and sizes by hand.

To scale text relative to the default body text size, use the following commands: ***\tiny,*** ***\scriptsize,*** ***\footnotesize,*** ***\small,*** ***\normalsize,*** ***\large,*** ***\Large,*** ***\LARGE,*** ***\huge,*** ***\Huge.*** These commands change the size within a given scope. For instance, {\Large some words} will change the size of the text "**some words**" only, and does not affect the font in the rest of the document.

The most common font styles in LATEX are bold, italics and underlined, but there are a few more as listed below:

Style	**Command**	**Switch command**	**Output**
medium	\textmd{Sample Text 0123}	\mdseries	Sample Text 0123
bold	\textbf{Sample Text 0123}	\bfseries	**Sample Text 0123**

upright	\textup{Sample Text 0123}	\upshape	Sample Text 0123
italic	\textit{Sample Text 0123}	\itshape	*Sample Text 0123*
slanted	\textsl{Sample Text 0123}	\slshape	*Sample Text 0123*
small caps	\textsc{Sample Text 0123}	\scshape	SAMPLE TEXT 0123

Lab Exercises:

1) Create a document in latex using report class and perform the following.
   a. Include sections and subsections in the document
   b. Generate table of contents for the report.
   c. Include a title page for the report.

2) Create a document in latex using the article class and perform the following.
   a. Include tables in the document and add referencing to the table in the text.
   b. Include figures in the document and add referencing to the figures in the text.
   c. Create ordered and unordered lists in the document.

3) Create a class timetable as shown below using tabular environment.

(Note: *\rotatebox* command can be used to rotate the text.
Command Format: *\rotatebox{angleofrotation}{text}*. Include the package *rotating* to use the command.)

Day	9 – 9.50	9.55 – 10.50	10.50 – 11. 05	11.05 – 12.00	12.00 – 1.00	1.00 – 1.50	1.55 – 2.45	2.45 – 3.00	3.00 – 4.00
MON			Break		Lunch			Break	
TUE									
WED									
THU									
FRI									

LAB NO.: 7 Date:

LATEX EQUATIONS, BIBLIOGRAPHY AND USAGE OF TEMPLATES

Objectives:

In this lab, student will be able to:

1. **Write equations in latex**
2. **Generate bibliography for a document**
3. **Use templates for preparing a research article**

1. EQUATIONS

Math expressions are separate from text in LATEX. To enter a math environment in the middle of text, use the dollar sign $, for example $F = ma$ produces $F = ma$. Everything between the two $ signs will be considered math formula.

To type a math expression that is on its own line and centered, use $$:

Source: Output:

The following is an important equation:

```
$$E = mc^2$$
```

To give an equation a number and have it referable, use the equation environment and use a \label command:

Source:

```
The following is an important equation:
\begin{equation}
    \label{emc}
    E = mc^2
\end{equation}
Please memorize Equation \ref{emc}.
```

Output:

- Superscript and subscript are done using ^ and _ characters. If multiple characters are to be used in the super/subscript, then surround them with curly braces.
- ≠, ≥, and ≤are \neq, \geq, and \leq.
- ∀and ∃are written as \forall and \exists.
- ∪and ∩are \cup and \cap.
- {} are done with \{ and \}.

There are many many other symbols available. You can search for "latex symbols" online and come up with the references.

1. INSERTING REFERENCES

LATEX includes features that allow you to easily cite references and create bibliographies in your document. This document will explain how to do this using a separate BibTeX file to store the details of your references.

2.1 The BibTeX file

Your BibTeX file contains all the references you want to cite in your document. It has the file extension .bib. It should be given the same name as and kept in the same folder as your .tex file. The .bib file is plain text - it can be edited using Notepad or your LATEX editor (e.g. TeXworks). You should enter each of your references in the BibTeX file in the following format:

```
@article{
    Birdetal2001,
    Author = {Bird, R. B. and Smith, E. A. and Bird, D. W.},
```

```
    Title = {The hunting handicap: costly signaling in human foraging strategies},
    Journal = {Behavioral Ecology and Sociobiology},
    Volume = {50},
    Pages = {9-19},
    Year = {2001}
}
```

Each reference starts with the reference type (@article in the example above). Other reference types include @book, @incollection for a chapter in an edited book and @inproceedings for papers presented at conferences1.

The reference type declaration is followed by a curly bracket, then the citation key. Each reference's citation key must be unique - you can use anything you want, but a system based on the first author's name and year (as in the example above) is probably easiest to keep track of.

The remaining lines contain the reference information in the format

Field name = {field contents},.

You need to include LaTeX commands in your BibTeX file for any special text formatting - e.g. italics (\emph{Rattus norvegicus}), quotation marks (``..."), ampersand (\&).

Surround any letters in a journal article title that need to be capitalised with curly brackets {...}. BibTeX automatically uncapitalises any capital letters within the journal article title. For example, \Dispersal in the contemporary United States" will be printed as \Dispersal in the contemporary united states", but \Dispersal in the contemporary {U}nited {S}tates" will be printed as \Dispersal in the contemporary United States".

You can type the BibTeX file yourself, or you can use reference management software such as EndNote to create it.

2.2 Inserting the bibliography

Type the following where you want the bibliography to appear in your document (usually at the end):

```
\bibliographystyle{plain}
\bibliography{Doc1}
```

where *Doc1* is the name of your .bib file.

2.3 Citing references

Type \cite{citationkey} where you want to cite a reference in your .tex document. If you don't want an in text citation, but still want the reference to appear in the bibliography, use \nocite{citationkey}.

To include a page number in your in-text citation put it in square brackets before the citation key: \cite[p. 215]{citationkey}.

To cite multiple references include all the citation keys within the curly brackets separated by commas: \cite{citation01,citation02,citation03}.

2.4 Styles

2.4.1 Numerical citations

LATEX comes with several styles with numerical in-text citations, these include:

- **Plain** The citation is a number in square brackets (e.g. [1]). The bibliography is ordered alphabetically by first author surname. All of the authors' names are written in full.

- **Abbrv** The same as plain except the authors' first names are abbreviated to an initial.

- **Unsrt** The same as plain except the references in the bibliography appear in the order that the citations appear in the document.

- **Alpha** The same as plain except the citation is an alphanumeric abbreviation based on the author(s) surname(s) and year of publication, surrounded by square brackets (e.g. [Kop10]).

2.4.2 Author-date citations

Use the natbib package if you want to include author-date citations. Natbib uses the command \citep{...} for a citation in brackets (e.g. [Koppe, 2010]) and \citet{...} for a citation where only the year is in brackets (e.g. Koppe [2010]). There are lots of other ways that you can modify citations when using the natbib package - see the package's reference sheet for full details.

Natbib comes with three bibliography styles: plainnat, abbrvnat and unsrtnat. These format the bibliography in the same way as the plain, abbrv and unsrt styles, respectively.

2.4.3 Other bibliography styles

If you want to use a different style (e.g. one provided by the journal you are submitting an article to) you should save the style file (.bst file) in the same folder as your .tex and .bib files. Include the name of the .bst file in the \bibliographystyle{...} command.

2.5 Try it out:

- Create a new file in your LATEX editor (File menu > New).
- Type your references in the correct format
- Click the Save button, the Save File window will open.
- Give the file the same name as your .tex document (for example, Doc1) and save it as a BibTeX database in the same folder as your .tex file.
- Switch to your .tex document and insert \cite, \bibliographystyle and \bibliography commands in the relevant places.
- Typeset your .tex file.
- Switch to your .bib file, choose BibTeX from the typeset menu and click the Typeset button.
- Switch to your .tex file and typeset it twice. The in-text citations and reference list should be inserted.

Lab Exercises:

1) Create an article class for the provided sample text, and then insert equations into the created article.

2) Generate bibliography for the article and refer to them in the text in author-year format and then numbering format.

3) Create your own CV.

4) For the provided sample text, create an article class in latex using IEEE template. (Note: You have to set the class of the documentclass to IEEETrans class)

LAB NO.: 8 **Date:**

HTML5

Objectives:
In this lab, student will be able to:

- **Develop HTML5 web pages**

DESCRIPTION

HTML5 – Hyper Text Markup Language Version 5

HTML5 is the 5th and newest version of HTML standard, providing new features like rich media support, interactive web applications etc.

The most interesting HTML5 elements are:

- Semantic elements like <header>, <footer>, <article> and <section>
- Attributes of form elements like number, date, time, calendar and range.
- Graphic elements like <svg> and <canvas>
- Multimedia elements like <audio> and <video>

There are several Application Programming Interfaces too in HTML5 like HTML Geolocation, HTML Drag and Drop, HTML Web Workers etc.

Several elements of HTML4 have been removed in HTML5 like <big>, , <font>, <frame>, <frameset>, <strike> etc.

To indicate that your HTML content uses HTML5, simply add <!DOCTYPE html> on top of the html code.

Procedure to create and HTML document:

In notepad type the necessary code & save with the file name mentioned with .html or .htm extension.

Example:

```
<html>
<head>
<title> My First Page </title>
</head>
<body>
<h1> Hello </h1>
<h2> Welcome to Internet Technologies Lab </h2>
</body>
</html>
```

HTML5 Elements

HTML5 offers new elements for better document structure. The below given table gives a brief description on few HTML5 elements.

TAG	DESCRIPTION
<article>	Defines an article in a document
<dialog>	Defines a dialog box or window
<header>	Defines a header for a document or a section
<footer>	Defines a footer for a document or a section
<nav>	Defines navigation links
<time>	Defines a date/time
<output>	Defines the result of a calculation
<canvas>	Draw graphics, on the fly, via scripting(JavaScript)
<audio>	Defines sound content
<source>	Defines multiple media resources for media elements

<video>	Defines video or movie

Figure 1.1 Few HTML5 elements

HTML5 Input types and Attributes

New Input Types	New Input Attributes
• color • date • datetime • datetime-local • email • month • number • range • search • tel • time • url • week	• autocomplete • autofocus • form • formaction • formenctype • formmethod • formnovalidate • formtarget • height and width • list • min and max • multiple • pattern (regexp) • placeholder • required • step

Figure 1.2 HTML5 input types and attributes

The above figure lists the new input types and attributes of HTML5.

HTML5 Structural Tags

Tag	Description
<a>	Defines a hyperlink
 	Produces a single line break
	Specifies a division or a section in a document
<h1> to <h6>	Defines a HTML headings
<hr>	Produces a horizontal lines
	Defines a inline styleless section in a document
<nav>	Defines a navigation links

HTML5 Events

On visiting a website the user perform actions like clicking on links or image, hover over things etc. These are considered to be examples for Events.

Event handlers are developed to handle these events and this can be done using a scripting language like JavaScript, VBScript etc wherein event handlers are specified as a value of event tag attribute.

The following attributes (very few) can be used to trigger any **javascript** or **vbscript** code given as value, when there is any event occurs for any HTM5 element.

Attribute	Value	Description
offline	script	Triggers when the document goes offline
onchange	script	Triggers when an element changes
onclick	script	Triggers on a mouse click
oncontextmenu	script	Triggers when a context menu is triggered
ondrag	script	Triggers when an element is dragged
onerror	script	Triggers when an error occur
onfocus	script	Triggers when the window gets focus

onformchange	script	Triggers when a form changes
onload	script	Triggers when the document loads
onmousedown	script	Triggers when a mouse button is pressed
onpause	script	Triggers when a media data is paused
onselect	script	Triggers when an element is selected
onsubmit	script	Triggers when a form is submitted

Figure 1.3 HTML5 event attributes

HTML5 Canvas

The HTML <canvas> element is used to draw graphics, on the fly, via JavaScript. The <canvas> element is only a container for graphics. The user must use JavaScript to actually draw the graphics. Canvas has several methods for drawing paths, boxes, circles, text, and adding images.

A canvas is a rectangular area on an HTML page. By default, a canvas has no border and no content. The markup looks like this:

```
<canvas id="myCanvas" width="200" height="100"></canvas>
```

Note: Always specify an Id attribute (to be referred to in a script), and a width and height attribute to define the size of the canvas. To add a border, use the style attribute.

Example: To draw a circle

```
<!DOCTYPE html>
<html>
<body>
<canvas id="myCanvas" width="200" height="100" style="border:1px solid #d3d3d3;">
Your browser does not support the HTML5 canvas tag.</canvas>
<script>
var c = document.getElementById("myCanvas");
var ctx = c.getContext("2d");
ctx.beginPath();
ctx.arc(95,50,40,0,2*Math.PI);
ctx.stroke();
</script>
 </body>
</html>
```

Output

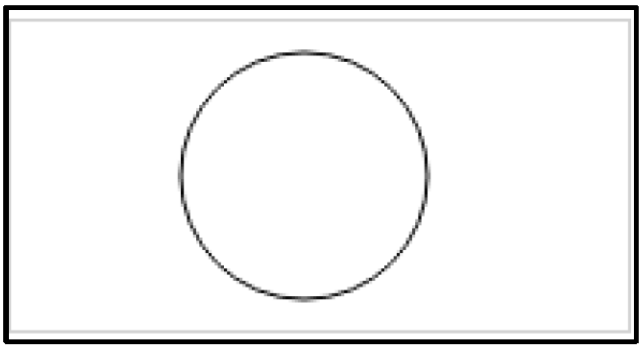


Figure 1.4

HTML5 Web Forms

The HTML <form> element defines a form that is used to collect user input. Form elements consist of various input elements like text field, check boxes, radio buttons, submit buttons etc.

<form>
</form>

Input Element

It is the most important form element and can be displayed in several ways, depending on the type attribute.

<input type = "text"> Defines a one line text input field
<input type = "radio"> Defines a radio button
<input type = "submit"> Defines a submit button

Action attribute

The action attribute defines the action to be performed when the form is submitted. Usually the form data is sent to a web page on the server when the user clicks on the submit button.

<form action = "/action_page.aspx">
// action_page.aspx contains a server side script that handles the form data.

If the action attribute is omitted, the action is set to the current page.

Method attribute

The method attribute specifies the HTTP method (GET or POST) to be used when submitting the form data.

<form action = "/action_page.aspx" method="get">

The default method when submitting form data is GET. When GET is used the submitted form data will be visible in the page address field. Therefore it must not be used when sending sensitive information.

Use POST method if the form data contains sensitive or personal information. It does not display the submitted form data in the page address field. It has no size limitations and can be used to send large amounts of data.

II. LAB EXERCISE:

- Write an HTML 5 program to show the usage of all the tags, attributes and elements used in HTML 5.

1. Specifying headings of the content and inserting paragraphs of text content :
    - Headings – h1 to h6
    - Paragraphs and line breaks
    - Text formatting
    - Preformatted text
    - Marking deleted and inserted text
    - Setting text direction of an element
    - Usage of div and span tags
1. Creating links to other HTML documents or Web resources :
    - Create hyperlinks
    - Using image as a Link
    - Open link in a new browser window
    - Navigate inside page
1. Inserting images into the HTML document :
    - Setting width and height of the images
    - Aligning images
    - Make a hyperlink of an image
    - Creating an image – map (An image with clickable regions)
1. Creating a simple table :
    - Setting the dimension of a table
    - Specify the table caption

- Tables with borders
- Using cell padding, cell spacing, row span, column span attributes

1. Making your own lists :
    - Unordered, Ordered and definition list
1. Creating a simple form :
    - Using Text input field
    - Creating password input field
    - Checkboxes and radio buttons
    - Drop down list
    - Select box with preselected value
    - Text area
    - Creating buttons (Show the usage of submit and reset buttons)
    - Grouping similar form fields
1. Making the HTML5 page more stylish using CSS
    - Using inline style - style attribute
    - Using embedded style
    - Link to an external style sheet – Link element
1. HTML5 Canvas
    - Embedding canvas into an HTML document
    - Drawing a line, circle, rectangle, arc onto the canvas
    - Filling the colour into shapes on canvas
    - Filling linear and radial gradient inside canvas shapes
    - Setting the stroke colour and width on canvas.

ADDITIONAL EXERCISE:

- Design a basic webpage of your interest using all the HTML5 elements.

LAB NO.: 09 **Date:**

CSS AND JAVASCRIPT

Objectives:

In this lab, student will be able to:

- **Familiarize with Cascading Style Sheets**
- **Embed the JavaScript code in HTML5 pages**

I. DESCRIPTION

CSS syntax

A CSS rule set consists of a selector and a declaration block. The selector points to the HTML element to be styled. The declaration block contains one or more declarations separated by semicolons.

H1 { color : blue ; font-size:12px}

Selector Property Value Declaration

CSS Selectors are used to "find" or select HTML elements based on their element name, id, class, attribute etc. The element selector selects the elements based on the element name. The id selector uses the id attribute of an HTML element to select a specific element. The id of an element should be unique within a page. To select an element with a specific id, write a # character followed by the id of the element.

```
#para1{
 text-align: center;
color:red; }
```

The class selector selects the elements with a specific class attribute. To select elements with a specific class, write a period (.) character, followed by the name of the class.

```
.center {
text-align: center;
color:red; }
```

JavaScript

JavaScript is a light weight and interpreted programming language. It is a scripting language that is commonly used for the client side web development. It makes the HTML pages more dynamic and interactive.

It can be used to put dynamic text in to HTML page (Eg: document.write("<h1>" + name + "</h1>"); // write variable text in to HTML page), react to events, validate data, create cookies etc.

Syntax

It can be implemented using JavaScript statements that are placed within the <script>….</script> HTML tags in a web page. The <script> tags, containing the JavaScript code can be placed anywhere within the web page, but normally recommended to place within the <head> tags.

The <script> tag alerts the browser program to start interpreting all the text between these tags as script. The script tag have mainly two attributes language and type, specifying the scripting language used.

To select an HTML element, JavaScript very often use the document.getElementById(id) method. The word document.write is a standard JavaScript command for writing output to a page.

```
<script language=”javascript” type=”text/javascript”>
document.getElementById("demo").innerHTML = "Hello JavaScript!";
</script>
```

II. SOLVED EXERCISE

Develop an HTML5 program to validate the credentials with appropriate internal styling with the help of CSS and JavaScript.

Program:

Login.html

```
<!DOCTYPE html>
<html>
<head>
<title>  Login page  </title>
<style>
body {
   background-color: lightblue;
}

h1 {
   color: white;
   text-align: center;
}

p {
   font-family: verdana;
   font-size: 20px;
}
</style>
<script language="javascript">
function check(form)       /*function to check userid & password*/
{
  /*the following code checkes whether the entered userid and password are matching*/
 if(form.userid.value == "myuserid" && form.pswrd.value == "mypswrd")
  {
   window.open("https://www.google.com", "_blank");
/*opens the target page while Id & password matches*/
  }
 else
 {
  alert("Error Password or Username")    /*displays error message*/
  }
}


</script>
</head>
<body>
```

```
<h1> Validate Credentials </h1>
<form name="login">
Username<input type="text" name="userid"/>
Password<input type="password" name="pswrd"/>
<input type="button" onclick="check(this.form)" value="Login"/>
<input type="reset" value="Cancel"/>
</form>
</body>
</html>
```

Output

III. LAB EXERCISE:

1) Create an HTML5 document to get an HTML5 element's position on the web page with the help of CSS and JavaScript function.
2) Write a JavaScript program to "Wish a User" at different hours of a day. Use appropriate dialog boxes for wishing the user. Display the dynamic clock on the web page. Make use of CSS and HTML5 elements for creative and attractive designs
3) Create an animation of rain using HTML5 canvas element. Apply appropriate usage of CSS and Javascript function to develop the animation.
4) Create an HTML 5 document that displays a bouncing ball. Use HTML5 elements, CSS and JavaScript functions.
5) Develop a color picker using HTML5 elements, CSS and JavaScript functions.

ADDITIONAL EXERCISE:

- Make the webpage dynamic by applying JavaScript functionality for the webpage which you have designed in Lab 8.

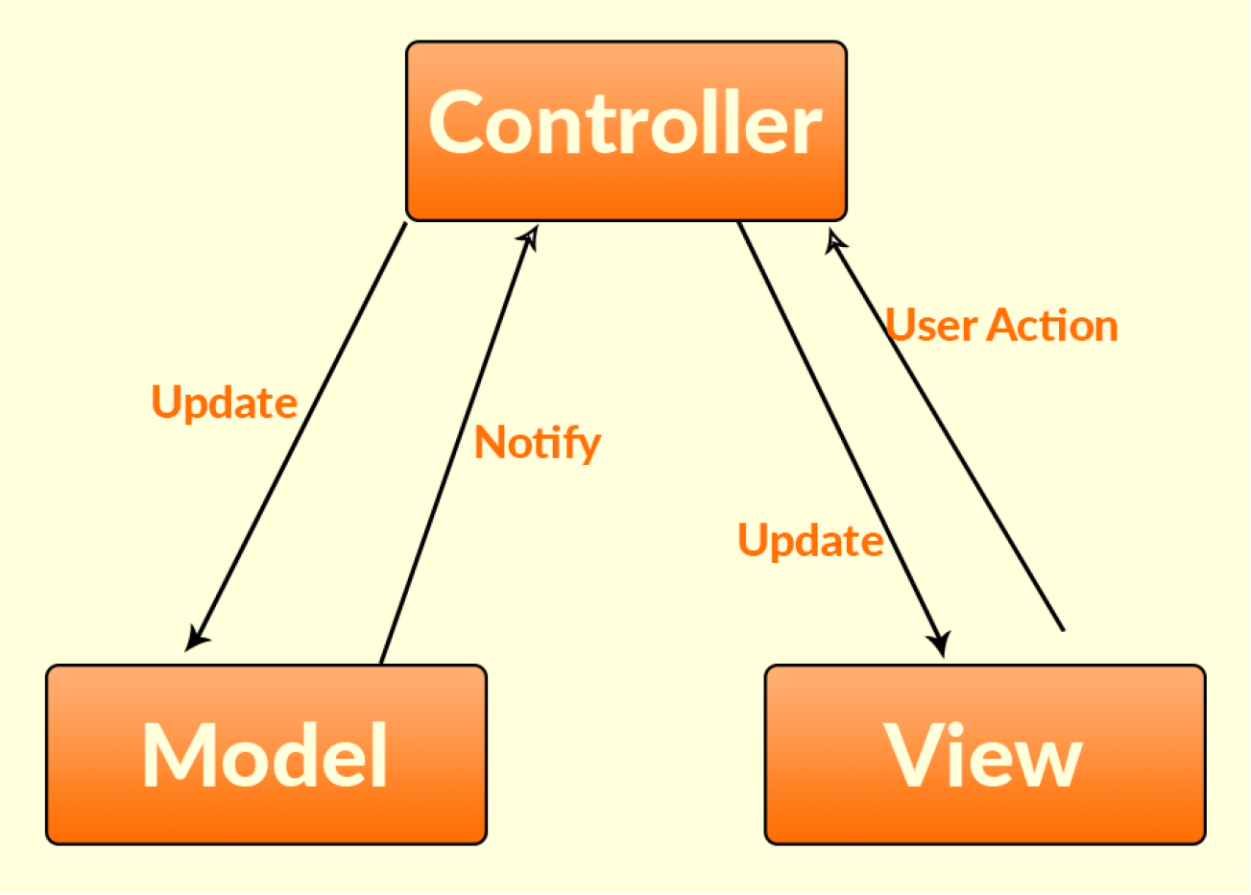
LAB NO.: 10 **Date:**

ANGULARJS

Objectives:
In this lab, student will be able to:

- **Develop web pages using Angularjs**

AngularJS:
AngularJS is an open source Javascript framework used for building Model-View-Controller based applications. This framework has been developed by a group of developers from Google.



The Angularjs framework is built on Model-View-Controller pattern. This pattern is based on splitting the business logic layer, the data layer and the presentation layer into separate sections. The advantage of Angularjs is that you need not have to write lot of code to bind data to HTML controls. By writing small snippets of code it is possible to bind data. The Controller represents the layer that has the business logic. User events trigger the functions which are stored inside your controller. The user events are part of the controller. Views are used to represent the presentation layer which is provided to the end users.

Models are used to represent your data. The data in your model can be as simple as just having primitive declarations. For example, if you are maintaining a student application, your data model could just have a student id and a name.

Hello World app

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="utf 8">
   <title>Hello World!</title>
</head>
<body ng-app="app">
<h1 ng-controller="HelloWorldCtrl">{{message}}</h1>
<script src="https://code.angularjs.org/1.6.9/angular.js"></script>
<script>
   angular.module("app", []).controller("HelloWorldCtrl", function($scope) {
   $scope.message="Hello World"
   } )
</script>

</body>
</html>
```

Save the above code as HelloWorld.htm or HelloWorld.html file. Open the file in a browser. You should see a message Hello World with title Hello World!.

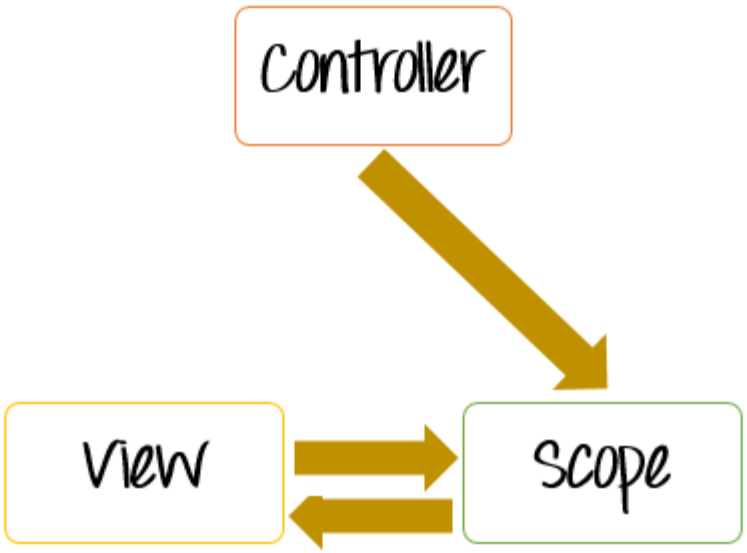
Explanation :

1. The "ng-app" keyword is used to denote that this application should be considered as an angular js application. Any name can be given to this application.
2. The controller is what is used to hold the business logic. In the h1 tag, we want to access the controller, which will have the logic to display "HelloWorld", so we can say, in this tag we want to access the controller named "HelloWorldCtrl".
3. We are using a member variable called "message" which is nothing but a placeholder to display the "Hello World" message.
4. The "script tag" is used to reference the angular.js script which has all the necessary functionality for angular js. Without this reference, if we try to use any AngularJS functions, they will not work.
5. "Controller" is the place where we are actually creating our business logic, which is our controller. The specifics of each keyword will be explained in the subsequent sections. What is important to note that we are defining a controller method called 'HelloWorldCtrl' which is being referenced in Step2.
6. We are creating a "function" which will be called when our code calls this controller. The $scope object is a special object in AngularJS which is a global object used within Angular.js. The $scope object is used to manage the data between the controller and the view.
7. We are creating a member variable called "message", assigning it the value of "HelloWorld" and attaching the member variable to the scope object.

Controller in Angularjs:

A Controller in AngularJs takes the data from the View, processes the data, and then sends that data across to the view which is displayed to the end user. The Controller will have your core business logic.

What Controller does from Angular's perspective



The controller's primary responsibility is to control the data which gets passed to the view. The scope and the view have two-way communication.

The properties of the view can call "functions" on the scope. Moreover events on the view can call "methods" on the scope. The below code snippet gives a simple example of a function.

```
app.controller('Ctrl', function($scope) {
            $scope.firstname="Ashok";
      $scope.lastname="G";
```

The function($scope) which is defined when defining the controller and an internal function which is used to return the concatenation of the $scope.firstName and $scope.lastName.

In AngularJS when you define a function as a variable, it is known as a Method.

How to build a basic controller

```
<!DOCTYPE html>
<html>
<head>
      <meta chrset="UTF 8">
</head>
<body>
<h1> Global Event</h1>

<div ng-app="DemoApp" ng-controller="DemoController">

      Name : <input type="text" ng-model="tutorialName"><br>
```

```
        Name is {{name}}
</div>
<script>
        var app = angular.module('DemoApp',[]);

        app.controller('DemoController', function($scope){
        $scope.name = "Angular JS";
        });
</script>

</body>
</html>
```

1. The ng-app keyword is used to denote that this application should be considered as an angular application. Anything that starts with the prefix 'ng' is known as a directive. "DemoApp" is the name given to this AngularJS application.
2. We have created a div tag and in this tag we have added an ng-controller directive along with the name of our Controller "DemoController". This basically makes our div tag the ability to access the contents of the Demo Controller. You need to mention the name of the controller under the directive to ensure that you are able to access the functionality defined within the controller.
3. We are creating a model binding using the ng-model directive. What this does is that it binds the text box for Tutorial Name to be bound to the member variable "name".
4. We are creating a member variable called "name" which will be used to display the information which the user types in the text box for Name.
5. We are creating a module which will attach to our DemoApp application. So this module now becomes part of our application.
6. In the module, we define a function which assigns a default value of "AngularJS" to our name variable.

$scope in Angularjs

The scope is a JavaScript object which basically binds the "controller" and the "view". One can define member variables in the scope within the controller which can then be accessed by the view.

```
angular.module('app',[]).controller(
        function($scope)
        {
                $scope.message = "Hello World"
        });



<h1>Global Event</h1>
<script src="https://code.angularjs.org/1.6.9/angular.js"></script>
<div ng-controller="DemoController">
        {{fullName("MIT","99")}}
</div>
<script type="text/javascript">
        var app = angular.module("DemoApp", []);
        app.controller("DemoController", function($scope) {
```

```
    $scope.fullName=function(firstName,lastname){
                return firstName + lastname;
                }
        } );
</script>
```

1. We are creating a behavior called "fullName". This behavior is a function which accepts 2 parameters (firstName,lastname).
2. The behavior then returns the concatenation of these 2 parameters.
3. In the view we are calling the behavior and passing in 2 values of "MIT" and "99" which gets passed as parameters to the behavior.

ng-repeat Directive

Sometimes we may be required to display a list of items in the view, so the question is that how can we display a list of items defined in our controller onto our view page.

Angular provides a directive called "ng-repeat" which can be used to display repeating values defined in our controller.

```
<h1>Global Event</h1>
<script src="https://code.angularjs.org/1.6.9/angular.js"></script>

<div ng-app="DemoApp" ng-controller="DemoController">
<h1>Topics</h1>
<ul><li ng-repeat="tpname in TopicNames">
                {{tpname.name}}
                </li></ul>
</div>

<script>
        var app = angular.module('DemoApp',[]);
        app.controller('DemoController', function($scope){

        $scope.TopicNames =[
                {name: "What controller do from Angular's perspective"},
                {name: "Controller Methods"},
                {name: "Building a basic controller"}];
                });
</script>
```

1. In the controller, we first define our array of list items which we want to define in the view. Over here we have defined an array called "TopicNames" which contains three items. Each item consists of a name-value pair.
2. The array of TopicsNames is then added to a member variable called "topics" and attached to our scope object.

3. We are using the HTML tags of (Unordered List) and (List Item) to display the list of items in our array. We then use the ng-repeat directive for going through each and every item in our array. The word "tpname" is a variable which is used to point to each item in the array topics.TopicNames.
4. In this, we will display the value of each array item.

ng-model Directive:

ng-model is a directive in Angular.JS that represents models and its primary purpose is to bind the "view" to the "model".

Usage of ng-model in textarea

```
<h1>Global Event</h1>
<script src="https://code.angularjs.org/1.6.9/angular.js"></script>

<div ng-app="DemoApp" ng-controller="DemoCtrl">
        <form>
                   Topic Description:<br> <br>
                   
        <textarea rows="4" cols="50" ng-model="pDescription"></textarea><br><br>
        </form>
</div>

<script>
        var app = angular.module('DemoApp',[]);
        app.controller('DemoCtrl', function($scope){
                $scope.pDescription="This topic looks at how Angular JS works \nModels in
Angular JS"});
</script>
```

The ng-model directive is used to attach the member variable called "pDescription" to the "textarea" control.

1. The "pDescription" variable will actually contain the text, which will be passed on to the text area control. We have also mentioned 2 attributes for the textarea control which is rows=4 and cols=50. These attributes have been mentioned so that we can show multiple lines of text. By defining these attributes the textarea will now have 4 rows and 50 columns so that it can show multiple lines of text.
2. Here we are attaching the member variable to the scope object called "pDescription" and putting a string value to the variable.
3. Note that we are putting the \n literal in the string so that the text can be of multiple lines when it is displayed in the text area. The \n literal splits the text into multiple lines so that it can render in the textarea control as multiple lines of text.

Usage of ng-model in input element:

The ng-model directive can also be applied to the input elements such as the text box, checkboxes, radio buttons, etc.

```
<h1> Global Event</h1>
<div ng-app="DemoApp" ng-controller="DemoCtrl">
	<form>
		   Topic Description:<br> <br>
		   

		Name : <input type="text" ng-model="pname"><br>
		   
		Topic : <br>   
		<input type="checkbox" ng-
model="Topic.Controller">Controller<br>   
		<input type="checkbox" ng-model="Topic.Models">Models
	</form>
	</div>

<script>
	var app = angular.module('DemoApp',[]);
	app.controller('DemoCtrl', function($scope){
		$scope.pname="MIT";

		$scope.Topic =
		{
			Controller:true,
			Models:false
		};	});
</script>
```

Usage of ng-model in select element:

```
<h1> Global Event</h1>

<div ng-app="DemoApp" ng-controller="DemoCtrl">
	<form>
		   Topic Description:<br> <br>
		   

		Name : <input type="text" ng-model="pName" value="MIT"><br>
		   
		Topic : <br>   

		<select ng-model="Topics">
		<option>{{Topics.option1}}</option>
		<option>{{Topics.option2}}</option>
		</select>
	</form>
	</div>

<script>
	var app = angular.module('DemoApp',[]);
	app.controller('DemoCtrl', function($scope){
	$scope.pName="MIT";
```

```
        $scope.Topics =
        {
                option1 : "Controller",
                option2 : "Module"
        };    });
</script>
</body>
</html>
```

Lab Exercises :

- Develop an AngularJS Application to display the text CSE branch, MIT Manipal with H3 tag.

- Develop an AngularJS Application which has two textboxes for First Name and Last Name. A hint should be displayed on the two textboxes containing "Enter First Name" and "Enter Last Name". As soon as some content is entered in the two textboxes it should be displayed below the two textboxes.

- Develop an AngularJS Application which has two textboxes. The first textbox is for price and the second one for quantity. Write an AngularJS function in the scope object which takes two numbers and returns the product of the two numbers. As soon as the textboxes are altered, display the product of the two numbers below the second textbox.

- Develop an AngularJS Application which has an object named Months. Store all the months in the Months object. Display the months in a listbox.

Additional Exercises :

1. Create an Array named leapYears in an AngularJS application and store all the leap years starting from 2000 to 2030. Display the leap years in a list box.

2. Develop an AngularJS Application which has a variable in $scope named count. Initialize the count to 5. Display count number of textboxes in the web page.

LAB NO.: 11 **Date:**

ANGULARJS-II

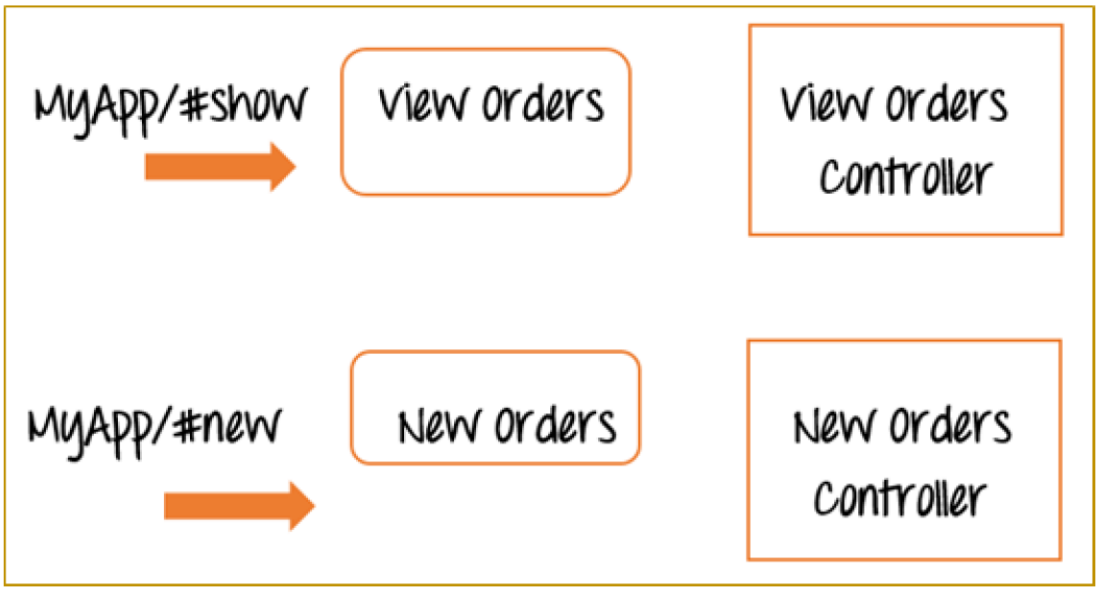
ng-view:

Nowadays, everyone would have heard about the "Single Page Applications". Many of the well-known websites such as Gmail use the concept of Single Page Applications (SPA's).

SPA's is the concept wherein when a user requests for a different page, the application will not navigate to that page but instead display the view of the new page within the existing page itself.

A view is the content which is shown to the user. Basically what the user wants to see, accordingly that view of the application will be shown to the user.

The below diagram and subsequent explanation demonstrate how to make this application as a single page application.



Now, instead of having two different web pages, one for "View orders" and another for "New Orders", in AngularJS, you would instead create two different views called "View Orders" and "New Orders" in the same page.

So when the application goes to MyApp/#show, it will show the view of the View Orders, at the same time it will not leave the page. It will just refresh the section of the existing page with the information of "View Orders". The same goes for the "New Orders" view.

```
<h1> Global Event</h1>

<div class="container">
   <ul><li><a href="#!NewEvent"> Add New Event</a></li>
```

```
      <li><a href="#!DisplayEvent"> Display Event</a></li>
   </ul>
   <div ng-view></div>
</div>

<script>
   var app = angular.module('sampleApp',["ngRoute"]);
   app.config(function($routeProvider){
      $routeProvider.
      when("/NewEvent",{
         templateUrl : "add_event.html",
         controller: "AddEventController"
      }).
      when("/DisplayEvent", {
         templateUrl: "show_event.html",
         controller: "ShowDisplayController"
      }).
      otherwise ({
         redirectTo: '/DisplayEvent'
      });
   });
   app.controller("AddEventController", function($scope) {

      $scope.message = "This is to Add a new Event";

   });
   app.controller("ShowDisplayController",function($scope){

      $scope.message = "This is display an Event";

   });
</script>
</body>
</html>
```

Create pages called add_event.html and show_event.html. Keep the pages simple as shown below.

add_event.html

```
<h2>Add New Event</h2>

{{message}}
```

show_event.html

```
<h2>Show Event</h2>

{{message}}
```

AngularJS Expressions

Expressions are variables which were defined in the double braces {{ }}.A simple example of an expression is {{5 + 6}}.

```
<script src="https://code.angularjs.org/1.6.9/angular-route.js"></script>
<script src="https://code.angularjs.org/1.6.9/angular.min.js"></script>

<h1> Global Event</h1>

<div ng-app="">
      Addition : {{6+9}}
</div>
```

Expression

```
<div ng-app="" ng-init="margin=2;profit=200">
      Total profit margin

      {{margin*profit}}
   </div>
```

Angularjs Strings

```
<div ng-app="" ng-init="firstName='MIT';lastName='Manipal'">

         
      First Name : {{firstName}}<br>   
      last Name : {{lastName}}

   </div>
```

Angular.js Objects

```
<div ng-app="" ng-init="person={firstName:'MIT',lastName:'Manipal'}">

      
   First Name : {{person.firstName}}<br>   
   Last Name : {{person.lastName}}

</div>
```

Angularjs Arrays

```
<div ng-app="" ng-init="marks=[1,15,19]">

   Student Marks<br>   
   Subject1 : {{marks[0] }}<br>   
   Subject2 : {{marks[1] }}<br>   
   Subject3 : {{marks[2] }}<br>   
</div>
```

$eval function

The $eval function allows one to evaluate expressions from within the controller itself. So while expressions are used for evaluation in the view, the $eval is used in the controller function.

Let's look at a simple example on this.

```
<script>
   var sampleApp = angular.module('sampleApp',[]);
   sampleApp.controller('AngularController',function($scope){
      $scope.a=1;
      $scope.b=1;

      $scope.value=$scope.$eval('a+b');
   });
</script>
```

AngularJS Filter Example: Currency, JSON, Number, Lowercase

A filter formats the value of an expression to display to the user.

For example, if you want to have your strings in either in lowercase or all in uppercase, you can do this by using filters in Angular.

There are built-in filters such as 'lowercase', 'uppercase' which can retrieve the output in lowercase and uppercase accordingly. Similarly, for numbers, you can use other filters.

Lowercase

```
<div ng-app="DemoApp" ng-controller="DemoController">
   Tutorial Name : <input type="text" ng-model="tutorialName"><br>
   <br>
   This tutorial is {{tutorialName | lowercase}}

</div>
<script type="text/javascript">
   var app = angular.module('DemoApp',[]);
   app.controller('DemoController',function($scope){

      $scope.tutorialName ="Angular JS";
   });
</script>
```

Uppercase

```
   This tutorial is {{tutorialName | uppercase}}
```

Number

```
<div ng-app="DemoApp" ng-controller="DemoController">

   This tutorialID is {{tutorialID | number:2}}

</div>
<script type="text/javascript">
   var app = angular.module('DemoApp',[]);
   app.controller('DemoController',function($scope){

      $scope.tutorialID =3.565656;
   });
</script>
```

Currency:

This filter formats a currency filter to a number.

Suppose, if you wanted to display a number with a currency such as $, then this filter can be used.

```
<div ng-app="DemoApp" ng-controller="DemoController">

   This tutorial Price is {{tutorialprice | currency}}

</div>
<script type="text/javascript">
   var app = angular.module('DemoApp',[]);
   app.controller('DemoController',function($scope){

      $scope.tutorialprice =20.56;
   });
</script>
```

JSON:

This filter formats a JSON like input and applies the JSON filter to give the output in JSON.

In the below example we will use a controller to send a JSON type object to a view via the scope object. We will then use a filter in the view to apply the JSON filter.

```
<div ng-app="DemoApp" ng-controller="DemoController">

   This tutorial is {{tutorial | json}}

</div>
<script type="text/javascript">
   var app = angular.module('DemoApp',[]);
   app.controller('DemoController',function($scope){

      $scope.tutorial ={TutorialID:12,tutorialName:"Angular"};
   });
</script>
```

AngularJS Custom Filter:

Sometimes the built-in filters in Angular cannot meet the needs or requirements for filtering output. In such a case a custom filter can be created which can pass the output in the required manner.

In the below example we are going to pass a string to the view from the controller via the scope object, but we don't want the string to be displayed as it is.

We want to ensure that whenever we display the string, we pass a custom filter which will append another string and displayed the completed string to the user.

```
<div ng-app="DemoApp" ng-controller="DemoController">

   This tutorial is {{tutorial | Demofilter}}

</div>
<script type="text/javascript">
   var app = angular.module('DemoApp',[]);
   app.filter('Demofilter',function(){
      return function(input)
      {
         return input + " Tutorial"
      }
   });

   app.controller('DemoController',function($scope){

      $scope.tutorial ="Angular";
    });

</script>
```

Angularjs Custom Directive:

A custom directive in Angular Js is a user-defined directive with your desired functionality. Even though AngularJS has a lot of powerful directives out of the box, sometime custom directives are required.

```
<div ng-app="DemoApp">
   <div ng-mit=""></div>

</div>

<script type="text/javascript">
   var app = angular.module('DemoApp',[]);
   app.directive('ngMit',function(){

   return {
      template: '<div>Angular JS Tutorial</div>'
   }
   });
```

```
</script>
```

AngularJS Events:

ng-click

When creating web-based applications, application need to handle DOM events like mouse clicks, moves, keyboard presses, change events, etc. For example, if there is a button on the page and you want to process something when the button is clicked, we can use the ng-click event directive.

The "ng-click directive" is used to apply custom behavior to when an element in HTML clicked. This is normally used for buttons because that is the most common place for adding events which respond to clicks performed by the user.

```
<h1> Global Event</h1>

<button ng-click="count = count + 1" ng-init="count=0">
   Increment
</button>

<div>The Current Count is {{count}}</div>
```

ng-show

```
<h1> Global Event</h1>
<div ng-app="DemoApp" ng-controller="DemoController">
   <input type="button" ng-value="buttonText" ng-click="ShowHide()"/>

   <br><br><div ng-show = "IsVisible">Angular</div>
</div>

<script type="text/javascript">

   var app = angular.module('DemoApp',[]);

   app.controller('DemoController',function($scope){
      $scope.IsVisible = false;
         $scope.buttonText="Show Angular"

      $scope.ShowHide = function(){
         $scope.IsVisible = !$scope.IsVisible;
                  $scope.buttonText=($scope.IsVisible?"Hide Angular":"Show Angular");
      }
      });
</script>
```

ng-hide

```
<h1> Global Event</h1>
<div ng-app="DemoApp" ng-controller="DemoController">
   <input type="button" ng-value="buttonText" ng-click="ShowHide()"/>
```

```
    <br><br><div ng-hide = "IsHidden">Angular</div>
</div>

<script type="text/javascript">

    var app = angular.module('DemoApp',[]);

    app.controller('DemoController',function($scope){
        $scope.IsHidden = false;
            $scope.buttonText="Hide Angular"

        $scope.ShowHide = function(){
            $scope.IsHidden = !$scope.IsHidden;
                    $scope.buttonText=($scope.IsHidden?"Show Angular":"Hide Angular");
        }
        });
</script>
```

AngularJS Event Listener Directives:

You can add AngularJS event listeners to your HTML elements by using one or more of these directives:

- ng-blur
- ng-change
- ng-click
- ng-copy
- ng-cut
- ng-dblclick
- ng-focus
- ng-keydown
- ng-keypress
- ng-keyup
- ng-mousedown
- ng-mouseenter
- ng-mouseleave
- ng-mousemove
- ng-mouseover
- ng-mouseup
- ng-paste

Routing

Single Page Applications

Single page applications or (SPAs) are web applications that load a single HTML page and dynamically update the page based on the user interaction with the web application.

In AngularJS, routing is what allows to create Single Page Applications.

Adding Angular Route ($routeProvider)

Routes in AngularJS are used to route the user to a different view of your application. And this routing is done on the same HTML page so that the user has the experience that he has not left the page.

In order to implement routing the following main steps have to be implemented in your application in any specific order.

1. Reference to angular-route.js. This is a JavaScript file developed by Google that has all the functionality of routing. This needs to be placed in your application so that it can reference all of the main modules which are required for routing.

2. The next important step is to add a dependency to the ngRoute module from within your application. This dependency is required so that routing functionality can be used within the application. If this dependency is not added, then one will not be able to use routing within the angular.JS application.

Below is the general syntax of this statement. This is just a normal declaration of a module with the inclusion of the ngRoute keyword.

```
var module = angular.module("sampleApp", ['ngRoute']);
```

3. The next step would be to configure your $routeProvider. This is required for providing the various routes in your application.

Below is the general syntax of this statement which is very self-explanatory. It just states that when the relevant path is chosen, use the route to display the given view to the user.

```
when(path, route)
```

Links to your route from within your HTML page. In your HTML page, you will add reference links to the various available routes in your application.

```
<a href="#/route1">Route 1</a><br/>
```

Finally would be the inclusion of the ng-view directive, which would normally be in a div tag. This would be used to inject the content of the view when the relevant route is chosen.

```
<script src="https://code.angularjs.org/1.6.9/angular-route.js"></script>
<script src="https://code.angularjs.org/1.6.9/angular.min.js"></script>
<script src="https://code.angularjs.org/1.6.9/angular.js"></script>

<h1> Global Event</h1>

<div class="container">
   <ul>
      <li><a href="#Angular">Angular JS Topics</a></li>
      <li><a href="#Node.html">Node JS Topics</a></li>
   </ul>
   <div ng-view></div>
</div>
```

```
<script>
  var sampleApp = angular.module('sampleApp',['ngRoute']);
  sampleApp.config(['$routeProvider',

    function($routeProvider){
    $routeProvider.
    when('/Angular',{
      templateUrl : '/Angular.html',
      controller: 'AngularController'
    }).
    when("/Node", {
      templateUrl: '/Node.html',
      controller: 'NodeController'
    });
  }]);
  sampleApp.controller('AngularController',function($scope) {

    $scope.tutorial = [
      {Name:"Controllers",Description :"Controllers in action"},
      {Name:"Models",Description :"Models and binding data"},
      {Name:"Directives",Description :"Flexibility of Directives"}
    ]
  });

  sampleApp.controller('NodeController',function($scope){

    $scope.tutorial = [
      {Name:"Promises",Description :"Power of Promises"},
      {Name:"Event",Description :"Event of Node.js"},
      {Name:"Modules",Description :"Modules in Node.js"}
      ]
  });

</script>
```

Create pages called Angular.html and Node.html.

Lab Exercises:

1. Develop an AngularJS application which has three checkboxes. Define a boolean array with three values. Set the state of the checkboxes to the value of each array element. Use ng-bind.

2. Dseevelop an AngularJS application which has a textbox and button. User will enter a number in the textbox and click button. Then, the multiplication table for the number should be shown. Use AngularJS expressions.

3. Display a string "QwErTy" in an AngularJS application in all lowercase and uppercase using filters.

4. An automobile company sells cars, jeeps and SUVs. An AngularJS application has to be developed which has hyperlinks for car,jeep and SUV. On clicking the link a description about the corresponding vehicle has to be shown on the web page. Use routing.

Additional Exercises:

1. Display the value of PI(assume 22/7) to 2 and 4 decimal places. Use filters.
2. Display a JSON expression in JSON format. Use filters.

REFERENCES

1. Beginning Linux Programming, Neil Matthew & Richard Stones,(4E), Wiley India Pvt. Ltd.
2. Unix Concepts and Applications,Sumitabha Das,(4E), McGraw Hill Education(India) Private Limited.
3. Leslie Lamport, Latex - A document preparation system, Leslie Lamport, (2e), Addison-Wesley 1994.
4. A simple guide to LaTeX - Step by Step, URL: https://www.latex-tutorial.com/tutorials/
5. https://www.guru99.com/angularjs-tutorial.html last accessed on 14th may 2019