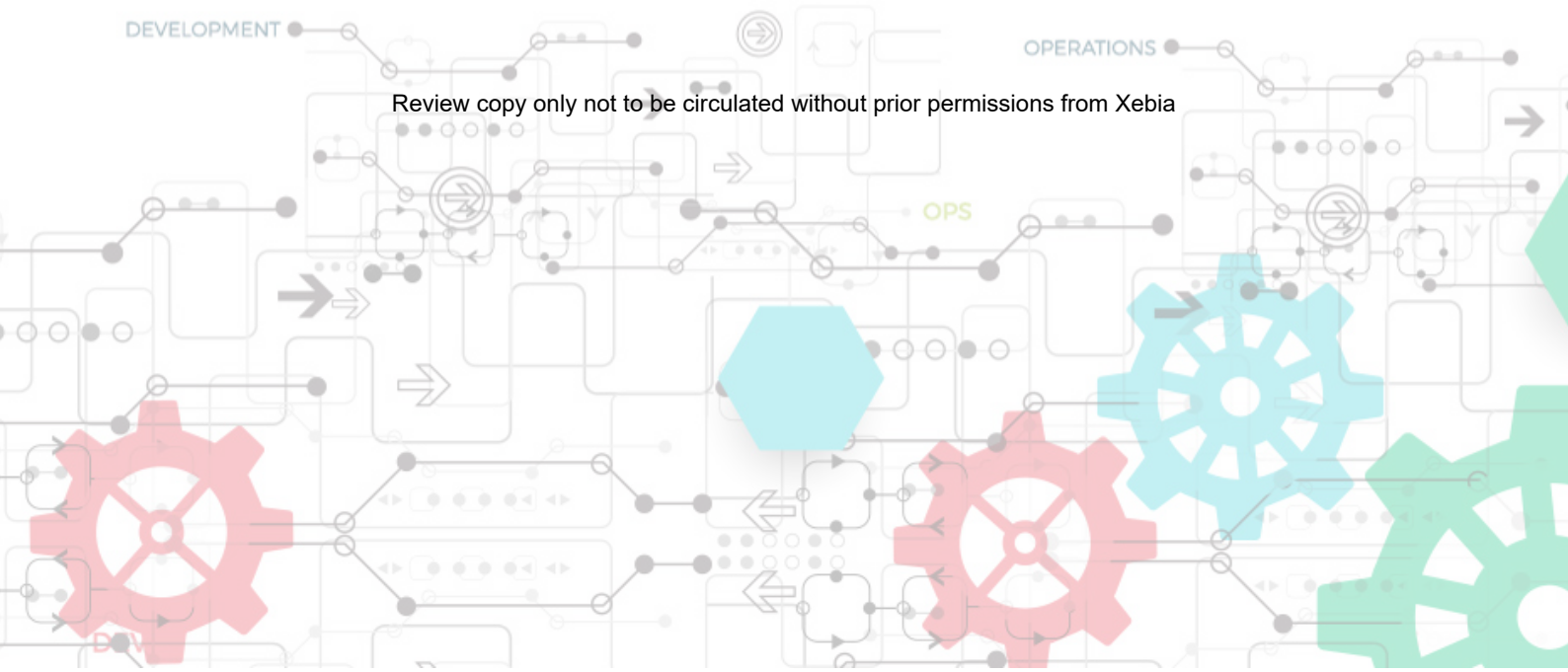


DevOps Automation: Lab

2. Development Automation Lab

(DevOps – Semester 3)

2.1 Introduction to Bash and Shell scripting

- 2.1.1 Input
- 2.1.2 Output
- 2.1.3 Streams
  - 2.1.3.1 Stdin
  - 2.1.3.2 Stdout
  - 2.1.3.3 Stderr
- 2.1.4 Variable
- 2.1.5 Substitution
  - 2.1.5.1 Variable Substitution
  - 2.1.5.2 Command Substitution
- 2.1.6 Function
- 2.1.7 Sub-Shell
- 2.1.8 Conditional statements
  - 2.1.8.1 IF statement
    - Nested IF
    - Ladder IF
  - 2.1.8.2 CASE statement
- 2.1.9 Loops
  - 2.1.9.1 FOR loop
  - 2.1.9.2 WHILE loop
- 2.1.10 Some Basic Shell Commands

2.2 Automation Scripts

- 2.2.1 Automation Scripts that save Time and Effort
  - 2.2.1.1 Archive logs and move them to the backup directory.
  - 2.2.1.2 Automatically delete archive files that are older than two days.
  - 2.2.1.3 Take MySQL Backups every 12 hours and move them to the backup directory.
  - 2.2.1.4 Email the summary of the web server requests every day.
  - 2.2.1.5 Continuously monitor and Restart the web server if it is not running.
  - 2.2.1.6. Block executing the forbidden commands.
  - 2.2.1.7 Monitor the disk usage and alert if it is beyond the given threshold.
  - 2.2.1.8 Moves the deleted files/folders to the recycle bin

2.2.1.9 Restores the deleted files/folders from the recycle bin
2.2.1.10 Log all the delete actions
2.2.1.11 Bulk file Downloader
2.2.1.12 Install LAMP stack.
2.2.2 Automation Scripts that prevent errors.
2.2.2.1 Find commands executed by all users that matches the given pattern.
2.2.2.2 Search and Replace a given text with a new text across multiple files
2.2.2.3 Check whether the given input is a valid AlphaNumeric Character
2.2.2.4 Truncate the given file to the specified size
2.2.2.5 Colourize the standard output by traditional diff tool.
2.2.2.6 Transform the given input text.

2.3 Working with Cron
2.3.1 Introduction
2.3.2 Structure of Cron Expression
2.3.3 Scheduling Cron Expressions
2.3.4 Examples
2.3.5 Best Practices

2.4 Working with Make and Makefiles
2.4.1 Introduction
2.4.2 Structure of Makefile
2.4.2.1 Target
2.4.2.2 Macro
2.4.2.3 Automatic Variables
2.4.2.4 Suffix & Pattern rules
2.4.3 The “Make” command
2.4.4 Best Practices
2.4.5 Examples
2.4.5.1 Basic Setup
2.4.5.2 Building binary from source code
2.4.5.3 Transforming the Makefile to adhere with Best Practices
2.4.5.4 Archiving logs with Make
2.4.5.5 Conditionals in Makefile

2.5 Error messages for users
2.5.1 Standard Output and Standard Error
2.5.2 Redirecting standard streams
2.5.3 Linux exit codes
2.5.4 Good error messages

2.6 Creating reusable library scripts for automation scripts

2.6.1 Examples

2.6.1.1 Formatting the Terminal

2.6.1.2 Utilities library to reduce logic duplication

2.1 Introduction to Bash and Shell scripting

Aim:

This lab tutorial aims to introduce the basics of Bash scripts.

Objective:

Bash - Bourne Again SHell is an sh-compatible shell for the Linux Operating System that incorporates useful features from the Korn shell (ksh) and C shell (csh). It offers functional improvements over SH for both programming and interactive use. In addition, most Shell scripts can be run by Bash without any modifications.

Methodology and Results:

Knowing the various components available in Bash and the basics to run scripts in a Linux Operating System.

2.1.1 Input

A script can make use of the Inputs given to the script to improve the user accessibility and also it enables the script to get the data dynamically during the runtime.

It receives the data from the stream "*stdin*" using the "read" command. This command gets the text typed in the "*stdin*" stream and assigns that data to the variable, which is passed in the argument of the read command.

Lab:

The following steps describe how to run the script in a Linux Operating System:

1. Create a script file called **"input.sh"** using the following code:

```
#!/bin/bash

# Reading the text from stdin and assigning it to the variable named "myName"
read myName

# Substitute the variable and print the text
echo "Hello ${myName}"
```

2. Assign execution permissions to the script.

```
$ chmod +x input.sh
```

3. Execute the script, type your name and press ENTER.

```
$ ./input.sh
```

2.1.2 Output

In Linux, the output is given to the terminal as two streams namely, "*stdout*" and "*stderr*". The former one is used to print any general information to the terminal while the latter one is used to display the error information in the terminal.

Lab:

1. Create a script file called **"output.sh"** with the following code:

```
#!/bin/bash

# Send the text to the stdout
echo "any standard text can be here"

# Send the text to the stderr
echo "any error text can be here" >&2
```

2. Assign execution permissions to the script.

```
$ chmod +x output.sh
```

3. Execute the script.

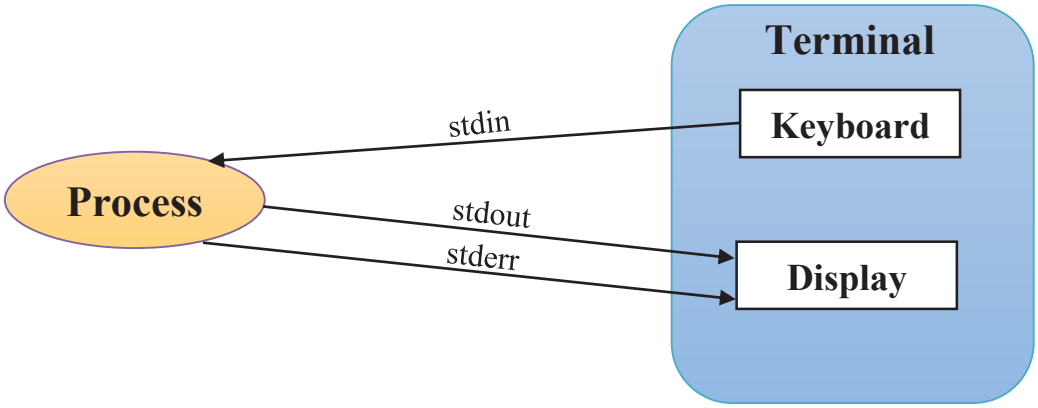
```
$ ./output.sh
```

2.1.3 Streams

Streams provide an abstraction of communication channels between a computer program and its execution environment. Primarily, there are three kinds of streams in most of the Linux Operating Systems, namely:

- Standard Input (*stdin*)
- Standard Output (*stdout*)

- Standard Error (*stderr*)



2.1.3.1 Stdin

Standard Input is a stream of data that enters the program from the terminal. Unless any output is redirected to the program, the standard input is expected from the keyboard.

This stream is generally used in programs to read the data dynamically and it has some additional benefits than getting the input from the program's argument. It allows to logically decide the number of inputs to get and prompt again in case of invalid input with exiting the program.

2.1.3.2 Stdout

Standard Output is the stream that is used to display an information on the terminal. By default, all the text printed in the terminal will be sent to the *stdout*. This can be a help message, a warning or an information that makes the program easier to use and know the process going on.

2.1.3.3 Stderr

Standard Error is a specific stream used to display the error information to the terminal. It is separated from *stdout* so that the error messages can be captured separately. With this separation, the general output of a program can be shown directly on the terminal while the error-related information can be separately written to a file.

2.1.4 Variable

A variable is a space labelled by a name in memory that has some data inside. In other words, a variable is a labelled container that contains some data. The data of a variable can be changed wherever required unless it is declared as read-only.

A variable can be created by having its name in the beginning, followed by an equal symbol and the value of the variable.

Example: VARIABLENAME = VARIABLEVALUE

Also, the variables can be annotated using the **local** and **readonly** keywords.

- The "local" annotation makes the variable to be available only within the scope in which it is defined.
- The "readonly" annotated variables don't allow to reassign their values after initialization.

Lab:

1. Create a script file called **"variable.sh"** with the following code:

```
#!/bin/bash

# Assigning the text "User!" to the variable
myName="User!"

# Substituting it by enclosing the variable name in ${}
echo "Hello ${myName}"
```

2. Assign execution permissions to the script.

```
$ chmod +x variable.sh
```

3. Execute the script.

```
$ ./variable.sh
```

2.1.5 Substitution

Substitution generally refers to placing a text inside some other text. At a higher level, any command or a script is just a text in a particular format. This enables to use the constructed text in place of any commands.

2.1.5.1 Variable Substitution

In the last lab, you have used a variable substitution operator to substitute the value of the variable *myName* to prefix the text *Hello* to it. In bash, this variable substitution can be done in the following two ways:

1. By prefixing the dollar symbol to the name of the variable.
   Example: $VARIABLE
2. By enclosing the variable in braces prefixed by the dollar symbol.
   Example: ${VARIABLE}

Both of them doesn't have any difference. But the second way of substitution is preferred to the first to minimize ambiguity. Besides, it also allows to have fallback values and search, replace operations.

2.1.5.2 Command Substitution

Command substitution operators can be used to substitute the text in the *stdout* of a process inside some other text. This operation will be very helpful in scenarios where one command depends on the output of another command especially when the latter command gets its input as arguments.

Bash allows to substitute the command's output in the following two ways:

1. Enclosing the command within back-tick symbols.
   Example: 'COMMAND ARGUMENTS'
2. Enclosing the command within parenthesis prefix by the dollar symbol.
   Example: $(COMMAND ARGUMENTS)

Lab:

1. Create a script file called **"hello.sh"** with the following code:

```
#!/bin/bash

# Creating a readonly variable called "greeting"
readonly greeting="Hello"

# Creating an ordinary variable called "current_time"
current_time=`date +%X`

# Substitute the output of the command "whoami".
echo "Hi, I am $(whoami)."

# Prompt the user to provide their name
read -e -p "Who are you? " myName

# Substituting the variables and printing them
echo "${greeting:-Hi}, $myName. Now the time is $current_time"
```


2. Assign execution permissions to the script.

```
$ chmod +x hello.sh
```

3. Execute the script, type your name when prompted and press ENTER.

```
$ ./hello.sh
```

2.1.6 Function

Functions are used to define a set of actions and execute those actions multiple times across the program whenever they are called. Code duplication can be considerably reduced with the use of functions.

The action of functions can be customized based on the input to the function called as parameters or function arguments. Each argument can be accessed by using $1, $2, …, $n.

The function can return a value which can be used by the caller of the function. The **return** statement is used by the function to return a value.

Note that the function should be defined first before it is called. It can be defined by using the "function" statement followed by the function name and the actions to execute enclosed inside the braces.

Lab:

1. Create a script file called **"function.sh"** with the following code:

```
#!/bin/bash

# Defining the function "greet" that accepts a parameter
function greet {
    echo "Hello $1!"
}

# Calling the function "greet" with parameter "World"
greet "World"

# Calling the function "greet" with parameter "User"
greet "User"
```

2. Assign execution permissions to the script.

```
$ chmod +x function.sh
```


3. Execute the script.

```
$ ./function.sh
```

2.1.7 Sub-Shell

The shell is the terminal where we type the commands and execute the programs. This interactive shell can itself call a script that runs in a new process other than the parent shell's process. Likewise, a running script can run some other script in a new process and this is called as a sub-shell.

Running parts of code or other commands in sub-shell creates a new process forking from the parent process and copies the memory of its parent process. This provides an advantage of running parallel operations, which in effect can execute multiple subtasks simultaneously.

Subshell variables cannot be accessed from the parent shell since they are contained with the new process.

The commands should be wrapped in parenthesis to run them in a sub-shell.

Example: (COMMAND1; COMMAND2; … COMMANDn;)

2.1.8 Conditional statements

Conditional statements are the vital components in any scripting or programming languages. They enable the ability to take decisions based on the given Boolean values. This allows us to logically do actions based on some criteria.

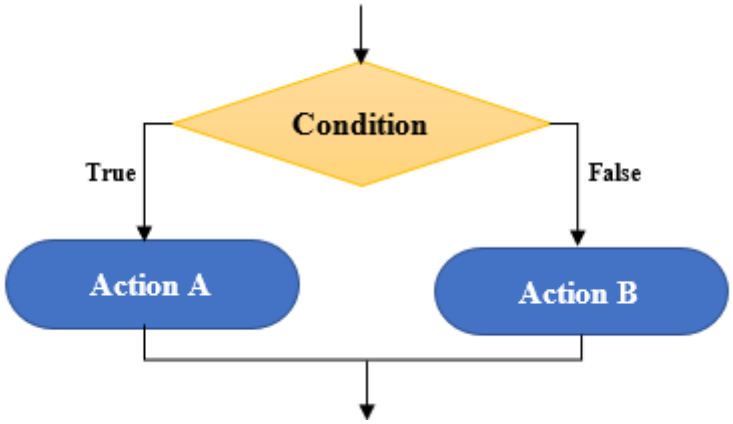
Much like regular programming languages, Bash supports two types of conditional statements:

- IF statements
- CASE statements

A part of the code (i.e. code to do some actions) is defined in these statements and condition to check will be given to them. If the given condition is true, then the statement executes the specified piece of code. Else it just skips to the next condition, if any.

2.1.8.1 IF statement

The following diagram illustrates the "If" statement.



The IF statement takes up a Boolean value directly and executes the given code if the given condition is true. Expressions that gets evaluated to Boolean can be given as inputs to the IF statements.

- The keyword **"if"** is used to define an IF statement and it is followed by the condition.
- The condition can be a Boolean value or an expression that gets evaluated to a Boolean value.
- Immediately, the action A is specified inside the braces. Optionally, the action B can be specified using the **"else"** keyword that gets executed if the given condition is false.
- Finally, the IF statement should be closed with the **"fi"** keyword.

Lab:

1. Create a script file called **"basicif.sh"** with the following code:

```
#!/bin/bash

# Initializing variables with a value
number1=5; number2=5

# Expression "[ $number1 -eq $number2 ]" that resolves to a Boolean value.
if [ $number1 -eq $number2 ]; then
    # Code to run if condition is true
    echo "$number1 is equal to $number2"
else
    # Code to run if condition is false
    echo "$number1 is not equal to $number2"
fi
```


2. Assign execution permissions to the script.

```
$ chmod +x basicif.sh
```

3. Execute the script.

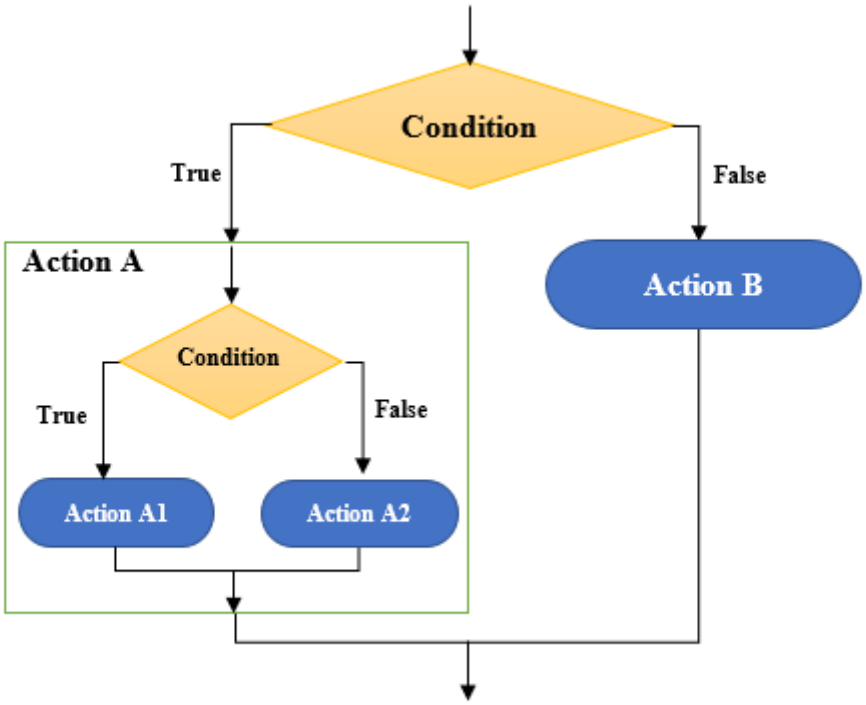
```
$ ./basicif.sh
```

Nested IF

An IF statement can have multiple IF statements in its action and this type is generally called as "Nested IF" statement. If the actions don't have any commands other than another IF statement, then the evaluation can be shrunk at one place rather than having a Nested IF statement.

Lab:

The following diagram illustrates the "Nested If" statement.



1. Create a script file called **"nestedif.sh"** with the following code:

```
#!/bin/bash

# Initializing variables with a value
number1=5; number2=5

# Evaluating the expression
if [ $number1 -ne $number2 ]; then
    if [ $number1 -gt $number2 ]; then
        echo "$number1 is greater than $number2"
    else
        echo "$number1 is lesser than $number2"
    fi
else
    echo "$number1 is equal to $number2"
fi
```

2. Assign execution permissions to the script.

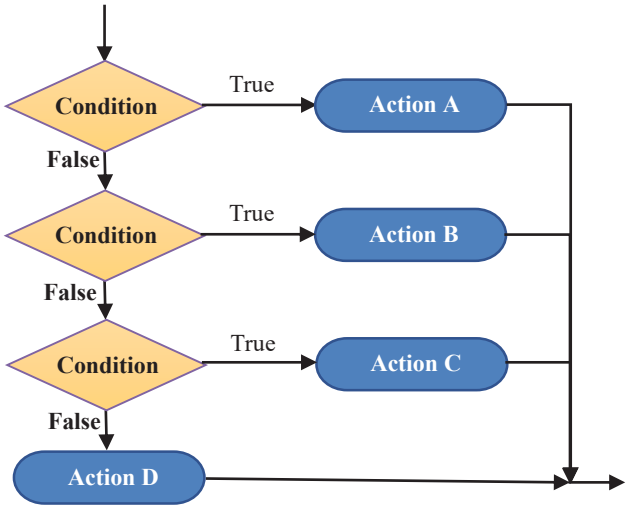
```
$ chmod +x nestedif.sh
```

3. Execute the script.

```
$ ./nestedif.sh
```

Ladder IF

In some scenarios, multiple conditions need to be checked one after another till a condition that resolves to true is met. Instead of having multiple IF statements inside the "else" clause of each parent IF statement, multiple **"elif"** keyword with conditions and code for each can be used. The following diagram illustrates the "Ladder If" statement.



Lab:

1. Create a script file called **"ladderif.sh"** with the following code:

```
#!/bin/bash

# Initializing variables with a value
number1=5; number2=5

# Evaluating the expression
if [ $number1 -gt $number2 ]; then
    echo "$number1 is greater than $number2"
elif [ $number1 -lt $number2 ]; then
    echo "$number1 is lesser than $number2"
else
    echo "$number1 is equal to $number2"
fi
```

2. Assign execution permissions to the script.

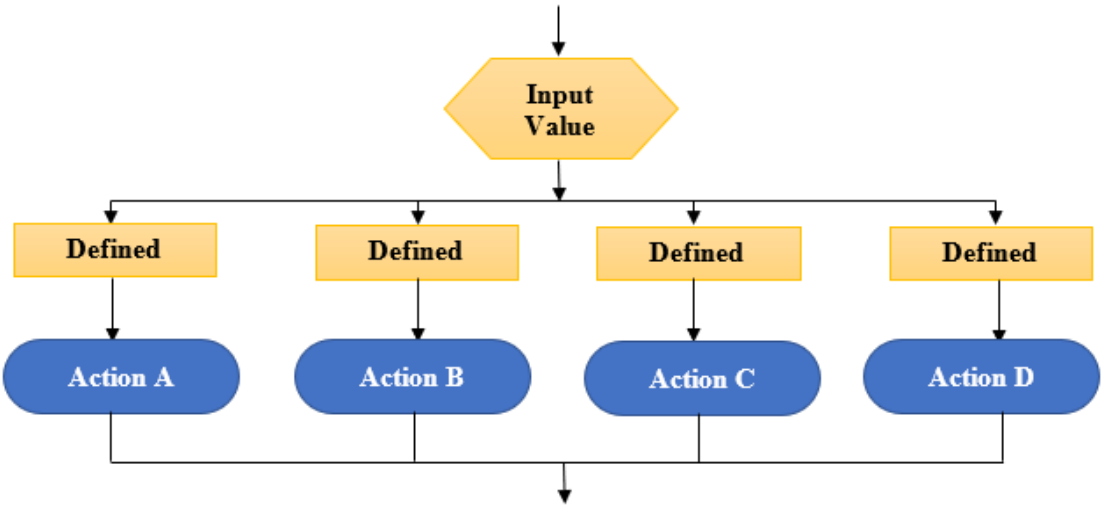
```
$ chmod +x ladderif.sh
```

3. Execute the script.

```
$ ./ladderif.sh
```

2.1.8.2 CASE statement

It is also a conditional statement that allows controlling the execution flow of a program. It is almost equivalent to the ladder IF statement from the previous lab in terms of functionality. But the syntax and way of providing conditions are slightly different than the ladder IF statement. The following diagram illustrates the "Case" statement.



- The keyword **"case"** and **"esac"** is used to start and end the switch statement.
- In each switch, the target value is kept first and is followed by a closing parenthesis.
- Following the condition, the action to execute for that specific condition is added and the switch is ended by using double colon symbol.

Lab:

1. Create a script file called **"switchcase.sh"** with the following code:

```
#!/bin/bash

name="john"

# A string is given as input to case statement.
case "$name" in

    # Checks for the switch that matches and executes that action.
     john)  echo "Welcome Admin"    ;;
    alexa)  echo "Welcome User"     ;;

    # If none is matched, "*" switch gets executed.
        *)  echo "Access Denied"    ;;
esac
```

2. Assign execution permissions to the script.

```
$ chmod +x switchcase.sh
```


3. Execute the script.

```
$ ./switchcase.sh
```

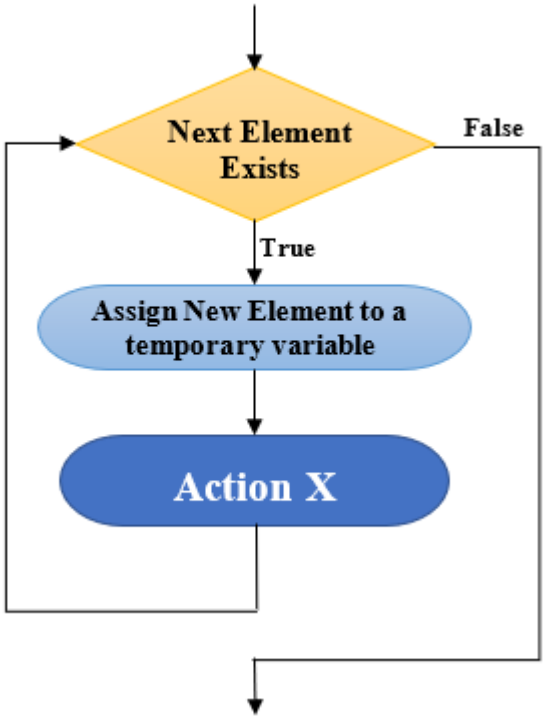
2.1.9 Loops

Loops are also a kind of control flow statement that enables the program to iterate a defined piece of code for a particular number of times or till a specified condition gets false.

These statements will be very helpful in places where multiple fields or data should be processed with a same set of action. The action can have loops in them as well thus allowing nested loops. Also, combining conditional statements inside loop statement allows performing different actions based on the given data.

2.1.9.1 FOR loop

FOR loop is a common type of looping statement used to iterate over a given number of items. Usually, an array of items will be given as input to FOR statement. This statement starts with the first item and switches to the next item in each iteration till it goes through all of them in the given array. The following diagram illustrates the “For loop” statement.



- The **"for"** keyword is used to begin a FOR statement and a temporary variable followed by the array variable separated by the **"in"** keyword is used to provide the condition.
- The action to execute in each iteration is placed between the **"do"** keyword and the **"done"** keyword.
- The keywords like **"break"** or **"continue"** can be used to exit the loop or jump to next iteration immediately.

In Bash, an array is a string composed of multiple items that are separated by a whitespace character.

Example: VARIABLE="ITEM1 ITEM2 ITEM3 … ITEMn"

Lab:

1. Create a script file called **"forloop.sh"** with the following code:

```
#!/bin/bash

# Loop through the items delimited by whitespace in the "names" variable.
names="Name1 Name2 Name3"
for name in $names; do
    echo "Hello $name"
Done

# Loop through the items in the substituted sequence command.
for number in `seq 1 10`; do
    echo "I am $number"
done
```

2. Assign execution permissions to the script.

```
$ chmod +x forloop.sh
```

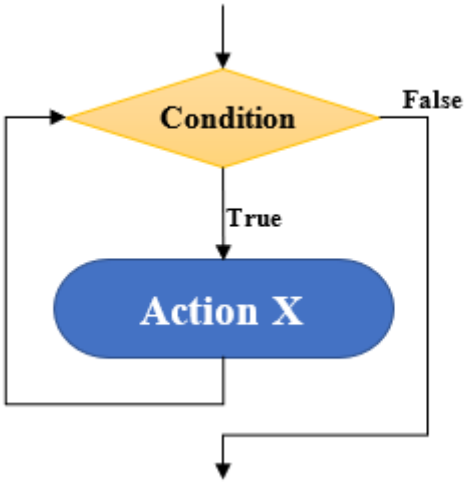
3. Execute the script.

```
$ ./forloop.sh
```

2.1.9.2 WHILE loop

WHILE loop is also a statement to do iterations same like FOR loop. The only difference is that WHILE loop does iteration till a given condition becomes false whereas the FOR statement does iteration over the given number of items.

The following diagram illustrates the "While loop" statement.



- The WHILE statement begins with the **"while"** keyword followed by the Boolean value.
- Any expression that evaluates to the Boolean can be used as an input to the while loop.
- Same like FOR statement, the action to be executed in the iteration is placed between the **"do"** keyword and the **"done"** keyword. Also, the **"break"** and the **"continue"** keywords can be used if required.

Lab:

1. Create a script file called **"whileloop.sh"** with the following code:

```
#!/bin/bash

count="9"

# Evaluates whether "count" is greater than zero and if true, enters loop.
while [ $count -gt 0 ]; do

    # Print the "count"
    echo "$count"

    # Decrease the value of count
    count=$[$count-1]
done
```

2. Assign execution permissions to the script.

```
$ chmod +x whileloop.sh
```

3. Execute the script.

```
$ ./whileloop.sh
```

2.1.10 Some Basic Shell Commands

1. Print the path of the current working directory.

```
$ pwd
```

2. List all the files and folders in the current directory.

```
$ ls -la
```

3. Change the current directory.

```
$ cd <ABSOLUTE_OR_RELATIVE_PATH_OF_THE_TARGET_DIRECTORY>
```

4. Update the last modified timestamp of a file. Can be used to create dummy file as well.

```
$ touch <FILE_NAME>
```

5. Print the contents of a file

```
$ cat <FILE_NAME>
```

6. Create a new directory.

```
$ mkdir <NEW_DIRECTORY_NAME>
```

7. Delete a directory.

```
$ rmdir <DIRECTORY_NAME>
```

8. Copy files to a directory.

```
$ cp <SOURCE_FILE> ... <SOURCE_FILE> <DESTINATION_DIRECTORY>
```

9. Move files to a directory.

```
$ mv <SOURCE_FILE> ... <SOURCE_FILE> <DESTINATION_DIRECTORY>
```

10. Delete files.

```
$ rm <FILE_NAME> ... <FILE_NAME>
```

11. Get Manual pages of all commands.

```
$ man <COMMAND_NAME>
```

12. Become a root user.

```
$ sudo su
```


13. Print available disk space of all partitions.

```
$ df -h
```

14. Print disk usage of files and folders.

```
$ du -s
```

15. Install new package to the operating systems.

```
$ apt-get install <PACKAGE_NAME>
```

16. Change file permissions.

```
$ chmod <PERMISSION> <FILE_NAME>
```

17. Change file ownership.

```
$ chown <USERNAME>:<GROUPNAME> <FILE_NAME>
```

18. Edit a file.

```
$ vim <FILE_NAME>
```

19. Search and Locate the files in the current directory that matches the given criteria.

```
$ find . -name <FILE_NAME> -user <FILE_OWNER> -size <FILE_SIZE> ...
```

20. Clears the text in the terminal.

```
$ clear
```

The output of these commands can be used as input to the other commands with the help of redirection and piping operators.

- The Input redirection operator **"<"** gets the name of the file from its right side and passes the content of the file to the command given to its left side.

```
$ head < name_of_the_file.txt
```

- The Output redirection operator **">"** gets the content of the commands specified on its left side and passes that content to the file specified on its right side.

```
$ ls -la > name_of_the_file.txt
```

- The Piping operator **"|"** passes the output of the program on its left side as an input to the program on its right side.

```
$ ls -la | head -5
```

2.2 Automation Scripts

In this topic, we will create the scripts for the various scenarios discussed in the second module of this semester.

2.2.1 Automation Scripts that save Time and Effort

2.2.1.1 Archive logs and move them to the backup directory.

Aim:

To Archive, all the matching logs are given in the input and to move the archived files to the specified backup directory.

Pre-Requisites:

- A set of sample log files in the file system.
- "tar" utility to create archives.

Steps:

1. Create a script file called "**lab-A1.sh**" with the following code:

```
#!/bin/bash

read -e -p "Log Directory: " log_directory
read -e -p "File Extension: " extension
read -e -p "Backup Directory: " backup_directory

tar czf archive.tar.gz $(find $log_directory -name "*.$extension")

mv archive.tar.gz $backup_directory/$(date +%F).tar.gz
rm $(find $log_directory -name "*.$extension")

exit 0
```

2. Assign execution permissions to the script.

```
$ chmod +x lab-A1.sh
```

3. Execute the script.

```
$ ./lab-A1.sh
```

4. Enter the location of the log directory, the extension of the log files and the location of the backup directory when prompted in the terminal.

2.2.1.2 Automatically delete archive files that are older than two days.

Aim:

To automatically delete the archive files that are older than two days. Make use of task schedulers like Cron to execute the script at periodic intervals.

Pre-Requisites:

- Archive files of the different timestamp.
- Permissions to schedule Cron jobs.

Steps:

1. Create a script file called "**lab-A2.sh**" with the following code:

```
#!/bin/bash

if [ -z "$1" ]; then
    echo "ERROR: No argument supplied" >&2; exit 1;
fi

archives_directory=$(realpath $1)
if [ ! -d "$archives_directory" ]; then
    echo "ERROR: Archives directory does not exist" >&2; exit 1;
fi

find $archives_directory -type f -name '*.tar.gz' -mmin +$((60 * 24 * 2)) -
exec rm {} \;

path_to_script=$(realpath "$0")
if ! (crontab -l | grep -Fxq "0 0 * * * $path_to_script $archives_directory");
then
    crontab -l | { cat; echo "0 0 * * * $path_to_script $archives_directory";
} | crontab -
    echo "Script added to Cron"
fi

exit 0
```

2. Assign execution permissions to the script.

```
$ chmod +x lab-A2.sh
```

3. Execute the script.

```
$ ./lab-A2.sh  <PATH_TO_THE_DIRECTORY_HAVING_LOG_FILES>
```

2.2.1.3 Take MySQL Backups every 12 hours and move them to the backup directory.

Aim:

To take backups of MySQL server in every 12 hours and to move them to the backup directory to keep the backups safe.

Pre-Requisites:

- MySQL server with MySQL client in the lab environment.
- Permissions to schedule Cron jobs.

Steps:

1. Create a script file called "**lab-A3.sh**" with the following code:

```
#!/bin/bash

if [ -z "$1" ]; then
    echo "ERROR: Credentials file not specified" >&2; exit 1;
elif [ -z "$2" ]; then
    echo "ERROR: Backup directory not specified" >&2; exit 1;
fi

credentials_file=$(realpath $1)
backup_directory=$(realpath $2)
if [ ! -f "$credentials_file" ]; then
    echo "ERROR: Credentials file does not exist" >&2; exit 1;
elif [ ! -d "$backup_directory" ]; then
    echo "ERROR: Backup directory does not exist" >&2; exit 1;
fi

source $credentials_file
if [ -z ${hostname:+word} ]; then
    echo "ERROR: hostname is not set" >&2; exit 1;
elif [ -z ${username:+word} ]; then
    echo "ERROR: username is not set" >&2; exit 1;
elif [ -z ${password:+word} ]; then
    echo "ERROR: password is not set" >&2; exit 1;
fi

mysqldump -h$hostname -u$username -p$password --all-databases > backup.sql
if [[ $? != 0 ]]; then
    echo "ERROR: Error in taking mysql backup" >&2; exit 1;
fi

mv backup.sql $backup_directory/$(date +%F_%R).sql
path_to_script=$(realpath "$0")
if ! (crontab -l | grep -Fxq "0 */12 * * * $path_to_script $credentials_file
$backup_directory"); then
    crontab -l | { cat; echo "0 */12 * * * $path_to_script $credentials_file
$backup_directory"; } | crontab -
    echo "Script added to Cron"
fi

exit 0
```

2. Assign execution permissions to the script.

```
$ chmod +x lab-A3.sh
```

3. Create a file that has the credentials to connect to MySQL in the following format.

```
# Credentials File

hostname=<DATABASE_ENDPOINT>
username=<DATABASE_USERNAME>
password=<DATABASE_PASSWORD>
```

4. Execute the script.

```
$ ./lab-A3.sh  <PATH_TO_CREDENTIALS_FILE>  <PATH_TO_BACKUP_DIRECTORY>
```

2.2.1.4 E-mail the summary of the web server requests every day.

Aim:

To email the summary of the web server such as the aggregates of all the HTTP status codes and the IP Addresses with top hits, to the given email ID every day.

Pre-Requisites:

- MailUtils should be configured.
- Permissions to schedule Cron jobs.
- Apache log files should exist locally on the filesystem.

Steps:

i. Create a script file called "**lab-A4.sh**" with the following code:

```
#!/bin/bash

if [ -z "$1" ]; then
  echo "ERROR: Location of the web server's log is not specified" >&2; exit 1;
fi

log_file=$(realpath $1)
email_id=admin@example.com
if [ ! -f "$log_file" ]; then
    echo "ERROR: Log file does not exist" >&2; exit 1;
fi

(
    echo -e 'Apache Web Server Access Logs - Summary\n'
    echo -e 'STATUS\t\t-\tCOUNT'
    cat $log_file | sed 's/.*HTTP\/1\.1\" \(...\).*/\1/g' | sort | uniq -c |
awk '{printf " %s\t\t-\t %s\n", $2, $1}'
    echo -e '\nIP ADDRESS\t-\tHITS'
    cat $log_file | sed 's/ .*//g'  | sort | uniq -c | awk '{printf "%s\t-\t
%s\n", $2, $1}'
) > /tmp/log_summary


cat /tmp/log_summary | mail -s "Apache Web Server Access Logs - Summary"
$email_id
if [[ $? != 0 ]]; then
    echo "ERROR: Error in sending email to $email" >&2; exit 1;
fi

path_to_script=$(realpath "$0")
if ! (crontab -l | grep -Fxq "0 * * * * $path_to_script $log_file"); then
    crontab -l | { cat; echo "0 * * * * $path_to_script $log_file"; } |
crontab -
    echo "Script added to Cron"
fi
exit 0
```

ii. Assign execution permissions to the script.

```
$ chmod +x lab-A4.sh
```

iii. Execute the script.

```
$ ./lab-A4.sh  <PATH_TO_THE_LOG_FILE>
```

2.2.1.5 Continuously monitor and Restart the web server if it is not running.

Aim:

To monitor the web server's status continuously at particular intervals and to restart the web server if it has stopped running.

Pre-Requisites:

- Apache web server installed and configured.
- Permissions to schedule Cron jobs.

Steps:

1. Create a script file called "**lab-A5.sh**" with the following code:

```
#!/bin/bash

if [ -z "$1" ]; then
   echo "ERROR: Webserver port is not specified in the arguments" >&2; exit 1;
fi

listening_port=$1
netstat -lnt | grep -q ":$1 "
if [[ $? != 0 ]]; then
    echo "ERROR: Web server is not running";
    /etc/init.d/apache2 restart
fi

path_to_script=$(realpath "$0")
if ! (crontab -l | grep -Fxq "*/1 * * * * $path_to_script $listening_port");
then
    crontab -l | { cat; echo "*/1 * * * * $path_to_script $listening_port"; }
| crontab -
    echo "Script added to Cron"
fi

exit 0
```

2. Assign execution permissions to the script.

```
$ chmod +x lab-A5.sh
```

3. Execute the script.

```
$ ./lab-A5.sh  <LISTENING_PORT_OF_THE_WEBSERVER>
```


2.2.1.6. Block executing the forbidden commands.

Aim:

To block the users from running forbidden commands by continuous validation of the given commands. Other commands should be executed normally whereas the forbidden commands should throw an error.

Pre-Requisites:

- Root permission to the Linux Operating System.

Steps:

1. Create a script file called "**lab-A6.sh**" with the following code:

```
#!/bin/bash

if [[ $(id -u) -ne 0 ]] ; then
    echo "ERROR: Please run as root."; exit 1;
fi

if [ ! -f "/opt/forbidden_commands.txt" ]; then
    echo "ERROR: The file /opt/forbidden_commands.txt does not exist" >&2;
exit 1;
fi

for user in $(getent passwd | cut -d : -f 6 | grep '/home' | sed
's:$:/.bashrc:'); do
    cat $user | grep -q 'validate_commands'
    if [[ $? != 0 ]]; then
        echo '
function validate_commands {
    cat /opt/forbidden_commands.txt | while read command; do
        if [ "$BASH_COMMAND" == "$command" ]; then
            echo "Sorry! \"$BASH_COMMAND\" cannot be executed";
            exit 1;
        fi
    done
}
trap validate_commands DEBUG' >> $user
    fi
done

exit 0
```

2. Assign execution permissions to the script.

```
$ chmod +x lab-A6.sh
```

3. Execute the script.

```
$ ./lab-A6.sh
```

2.2.1.7 Monitor the disk usage and alert if it is beyond the given threshold.

Aim:

To monitor the usage of the given disk and alert the user if it is beyond the given threshold.

Pre-Requisites:

- MailUtils should be configured.
- Permissions to schedule Cron jobs.

Steps:

1. Create a script file called "**lab-A7.sh**" with the following code:

```
#!/bin/bash

if [ -z "$1" ]; then
    echo "ERROR: Device name is not specified in the arguments" >&2; exit 1;
elif [ -z "$2" ]; then
    echo "ERROR: Threshold is not specified in the arguments" >&2; exit 1;
fi

device_name=$1
thresold_limit=$2
email_id=admin@example.com
percentage_used=$(df -H | grep "$device_name" | awk '{ print $5 }' | cut -d'%'
-f1)

if [ $percentage_used -ge $thresold_limit ]; then
    echo "Running out of space \"$device_name ($percentage_used%)\" on
$(hostname) as on $(date)" |
    mail -s "Disk usage breached the thresold limit" $email_id
fi

path_to_script=$(realpath "$0")
if ! (crontab -l | grep -Fxq "*/1 * * * * $path_to_script $device_name
$thresold_limit"); then
    crontab -l | { cat; echo "*/1 * * * * $path_to_script $device_name
$thresold_limit"; } | crontab -
    echo "Script added to Cron"
fi
exit 0
```

2. Assign execution permissions to the script.

```
$ chmod +x lab-A7.sh
```


3. Execute the script.

```
$ ./lab-A7.sh  <DEVICE_NAME>  <THRESHOLD_LIMIT>
```

2.2.1.8 Moves the deleted files/folders to the recycle bin

Aim:

To move the files/folders to the recycle bin when deleted.

Steps:

1. Create a script file called "**lab-A8.sh**" with the following code:

```
#!/bin/bash

recycle_bin="$HOME/.recycle_bin"
rm="/bin/rm -r "
copy="/bin/cp -r "

if [ $# -eq 0 ] ; then
    echo "ERROR: Please enter the file path to delete." >&2; exit 1;
fi
flags=""
while getopts "dfiPRrvW" args; do
    case $args in
        f ) exec $rm "$@"           ;;
        * ) flags="$flags -$args"   ;;
    esac
done
shift $(( $OPTIND - 1 ))

if [ ! -d $recycle_bin ] ; then
    mkdir $recycle_bin
fi

for arg; do
    newname="$recycle_bin/$(date "+%S.%M.%H.%d.%m").$(basename "$arg")"
    if [ -f "$arg" ] ; then
        $copy "$arg" "$newname"
    elif [ -d "$arg" ] ; then
        $copy "$arg" "$newname"
    fi
done

exec $rm $flags "$@"
exit 0
```

2. Assign execution permissions to the script.

```
$ chmod +x lab-A8.sh
```

3. Execute the script.

```
$ ./lab-A8.sh [OPTIONS]  <FILE1>  <FILE2>  ...  <FILEn>
```

2.2.1.9 Restores the deleted files/folders from the recycle bin

Aim:

To restore the deleted files/folders from the recycle bin.

Steps:

1. Create a script file called "**lab-A9.sh**" with the following code:

```
#!/bin/bash

recycle_bin="$HOME/.recycle_bin"
rm="/bin/rm"
move="/bin/mv"
destination=$(pwd)
if [ ! -d $recycle_bin ] ; then
    echo "ERROR: Recycle Bin was not found in home directory" >&2; exit 1;
fi
cd $recycle_bin
if [ $# -eq 0 ] ; then
    echo "Deleted files in the recycle bin:"
    ls -FC | sed -e 's/\([[:digit:]][[:digit:]]\.\)\{5\}//g' -e 's/^/  /'
    exit 0
fi
pattern_matches="$(ls *"$1" 2> /dev/null | wc -l)"
if [ $pattern_matches -eq 0 ] ; then
    echo "ERROR: No match for the pattern \"$1\"" >&2; exit 1;
fi
if [ $pattern_matches -gt 1 ] ; then
    echo "More than one file or directory match in the archive:"
    index=1
    for name in $(ls -td *"$1"); do
        datetime="$(echo $name | cut -c1-14| awk -F. '{ print $5"/"$4" at
"$3":"$2":"$1 }')"
        if [ -d $name ] ; then
            size="$(ls $name | wc -l | sed 's/[^0-9]//g')"
            echo " $index)   $1  (contents = ${size} items, deleted =
$datetime)"
        else
            size="$(ls -sdk1 $name | awk '{print $1}')"
            echo " $index)   $1  (size = ${size}Kb, deleted = $datetime)"
        fi
        index=$(( $index + 1))
    done
    echo ""
```



```
    read -e -p "Which version of $1 to restore ('0' to quit)? [1] : " desired

    if [ ${desired:=1} -ge $index ] ; then
        echo "ERROR: Restore cancelled by user: index value too big." >&2;
        exit 1;
    fi
    if [ $desired -lt 1 ] ; then
        echo "ERROR: Restore cancelled by user." >&2; exit 1;
    fi
    restore="$(ls -td1 *"$1" | sed -n "${desired}p")"
    if [ -e "$destination/$1" ] ; then
        echo "ERROR: Already exists in this directory. Cannot overwrite." >&2;
        exit 1;
    fi
    echo -n "Restoring file \"$1\" ..."
    $move "$restore" "$destination/$1"
    echo "done."

    read -e -p "Delete the additional copies of this file? [y] " answer
    if [ ${answer:=y} = "y" ] ; then
        $rm -rf *"$1"
        echo "deleted."
    else
        echo "additional copies retained."
    fi
else
    if [ -e "$destination/$1" ] ; then
        echo "ERROR: Already exists in this directory. Cannot overwrite." >&2;
        exit 1;
    fi
    restore="$(ls -d *"$1")"
    echo -n "Restoring file \"$1\" ... "
    $move "$restore" "$destination/$1"
    echo "done."
fi
exit 0
```

2. Assign execution permissions to the script.

```
$ chmod +x lab-A9.sh
```

3. Execute the script.

```
$ ./lab-A9.sh  <NAME_OF_THE_FILE_TO_RECOVER>
```

2.2.1.10 Log all the delete operations

Aim:

To Log all the delete operations made through the script.

Steps:

1. Create a script file called "**lab-A10.sh**" with the following code:

```
#!/bin/bash

removelog="/var/log/remove.log"
if [ $# -eq 0 ] ; then
  echo "Usage: $0 [-s] list of files or directories"; exit 0;
fi

if [ "$1" = "-s" ] ; then
  shift
else
  echo "$(date): ${USER}: $@" >> $removelog
fi

/bin/rm "$@"
exit 0
```

2. Assign execution permissions to the script.

```
$ chmod +x lab-A10.sh
```

3. Execute the script.

```
$ ./lab-A10.sh [OPTIONS] <FILE1> <FILE2> ...<FILEn>
```

4. View the deletion log.

```
$ cat /var/log/remove.log
```

2.2.1.11 Bulk file Downloader

Aim:

To download a list of files from the given URL list.

Steps:

1. Create a script file called "**lab-A11.sh**" with the following code:

```
#!/bin/bash

downloads_directory="$HOME/Downloads"
if [ -z "$1" ]; then
    echo "ERROR: Download List file is not specified in the arguments" >&2;
    exit 1;
fi

if ! [ -z "$2" ]; then
    downloads_directory=$2
fi

if [ ! -d $downloads_directory ] ; then
    mkdir -p $downloads_directory
fi

download_list=$1

cat $download_list | while read url; do
    echo "------------------------------------------------------------------------"
    echo "$url"
    wget -P $downloads_directory $url
done

exit 0
```

2. Assign execution permissions to the script.

```
$ chmod +x lab-A11.sh
```

3. Execute the script. The URLs file should have exactly one URL per line and it can have any number of lines required.

```
$ ./lab-A11.sh <PATH_TO_THE_FILE_HAVING_URLS> <OPTIONAL_DOWNLOAD_LOCATION>
```

4. View the downloaded files.

```
$ ls <DOWNLOAD_LOCATION>
```

2.2.1.12 Install LAMP stack.

Aim:

To Install Apache, MySQL, PHP on Linux machine (LAMP stack).

Pre-Requisites:

- Apt repository configured with valid mirrors
- Root permission to the Linux Operating System

Steps:

1. Create a script file called "**lab-A12.sh**" with the following code:

```
#!/bin/bash

echo -e "\n\nUpdating Apt Packages and upgrading latest patches\n"
sudo apt-get update -y && sudo apt-get upgrade -y

echo -e "\n\nInstalling Apache2 Web server\n"
sudo apt-get install apache2 apache2-doc apache2-mpm-prefork apache2-utils
libexpat1 ssl-cert -y

echo -e "\n\nInstalling PHP & Requirements\n"
sudo apt-get install libapache2-mod-php7.0 php7.0 php7.0-common php7.0-curl
php7.0-dev php7.0-gd php-pear php7.0-mcrypt php7.0-mysql -y

echo -e "\n\nInstalling MySQL\n"
sudo apt-get install mysql-server mysql-client -y

echo -e "\n\nPermissions for /var/www\n"
sudo chown -R www-data:www-data /var/www
echo -e "\n\n Permissions have been set\n"

echo -e "\n\nEnabling Modules\n"
sudo a2enmod rewrite
sudo phpenmod mcrypt

echo -e "\n\nRestarting Apache\n"
sudo service apache2 restart

echo -e "\n\nLAMP Installation Completed"
exit 0
```

2. Assign execution permissions to the script.

```
$ chmod +x lab-A12.sh
```


3. Execute the script.

```
$ ./lab-A12.sh
```

2.2.2 Automation Scripts that prevent errors.

2.2.2.1 Find commands executed by all users that matches the given pattern.

Aim:

Find the commands executed by all users that match the given pattern.

Pre-Requisites:

- Root permission to the Linux operating system.

Steps:

1. Create a script file called "**lab-B1.sh**" with the following code.

```
#!/bin/bash

if [[ $(id -u) -ne 0 ]] ; then
    echo "ERROR: Please run as root." ; exit 1 ;
fi

if [ -z "$1" ]; then
    echo "ERROR: Search pattern not specified" >&2; exit 1;
fi

pattern=$1
getent passwd | cut -d : -f 6 | sed 's:$:/.bash_history:' | xargs -d '\n' grep
-s -n -e "$pattern" > /tmp/search
cat /tmp/search | sed 's/.*\/\(.*\)\/\.bash_history\:\(.*\)\:/\1 \t \2 \t/g'

exit 0
```

2. Assign execution permissions to the script.

```
$ chmod +x lab-B1.sh
```

3. Execute the script.

```
$ ./lab-B1.sh <SEARCH_PATTERN>
```

2.2.2.2 Search and Replace a given text with a new text across multiple files

Aim:

To Search and Replace a given text with a new text across multiple files.

Steps:

1. Create a script file called "**lab-B2.sh**" with the following code:

```
#!/bin/bash

if [ "$#" -eq 0  ]; then
    echo "ERROR: Files/Folders to search & replace is not specified" >&2;
    exit 1;
fi

item_paths=""
for i in $@; do
    item_path=$(realpath $i)
    if [ -f "$item_path" ]; then
        item_paths+="$item_path "
    elif [ -d "$item_path" ]; then
        item_paths+="$item_path/* "
    else
       echo "ERROR: The given path \"$item_path\" is not a suitable type" >&2;
       exit 1;
    fi
done

read -e -p "Search For: " search_text
read -e -p "Replace With: " replace_text

sed -i "s/$search_text/$replace_text/g" $(grep -lz "$search_text" $item_paths)

exit 0
```

2. Assign execution permissions to the script.

```
$ chmod +x lab-B2.sh
```

3. Execute the script and provide the text to search and replace when prompted.

```
$ ./lab-B2.sh  <PATH_TO_FILE_OR_FOLDER> ... <PATH_TO_FILE_OR_FOLDER>
```


2.2.2.3 Check whether the given input is a valid AlphaNumeric Character

Aim:

To check whether the given input is a valid AlphaNumeric Character.

Steps:

1. Create a script file called "**lab-B3.sh**" with the following code:

```
#!/bin/bash

is_valid_alpha_numeric_text()
{
  filtered_input="$(echo $1 | sed -e 's/[^[:alnum:]]//g')"
  if [ "$filtered_input" == "$1" ]; then
    return 0
  else
    return 1
  fi
}

read -e -p "Enter Input to validate: " input_text
if [[ "$input_text" == '' ]]; then
    echo "ERROR: Please enter any text to validate." >&2; exit 1;
fi

if ! is_valid_alpha_numeric_text "$input_text" ; then
  echo "Input contains characters other than alphabets and numbers"
else
  echo "Input is absolutely valid."
fi

exit 0
```

2. Assign execution permissions to the script.

```
$ chmod +x lab-B3.sh
```

3. Execute the script.

```
$ ./lab-B3.sh  <TEXT_TO_VALIDATE>
```

2.2.2.4 Truncate the given file to the specified size

Aim:

To truncate the given file to the specified size.

Steps:

1. Create a script file called "**lab-B4.sh**" with the following code:

```
#!/bin/bash

if [ "$#" != "2" ]; then
    ( echo "
Usage: `basename $0` <size> <path>

Truncate the <path> to exactly <size> bytes.
* If <path> doesn't exist it is created.
* <size> is a number which can be optionally followed by K, M, G, etc." ) >&2;
    exit 1;
fi

size=$1
file="$2"

dd bs=1 seek=$size if=/dev/null of="$file"
exit 0
```

2. Assign execution permissions to the script.

```
$ chmod +x lab-B4.sh
```

3. Execute the script.

```
$ ./lab-B4.sh  <SIZE>  <PATH_TO_FILE>
```


2.2.2.5 Colourize the standard output by traditional diff tool.

Aim:

To colourize the standard output by the traditional diff tool.

Steps:

1. Create a script file called "**lab-B5.sh**" with the following code:

```
#!/bin/bash

if [ "$#" -ne "2" ]; then
    echo "ERROR: Two files/directories must be passed as arguments" >&2;
    exit 1;
fi

removed=$(echo -en '\033[31m')
added=$(echo -en '\033[32m')
changed=$(echo -en '\033[34m')
file=$(echo -en '\033[47m')
reset=$(echo -en '\033[0m')

diff -Naru "$@" | sed "
s/^\*\{3\}.*\*\{4\}/$changed&$reset/;t
  s/^-\{3\}.*-\{4\}/$changed&$reset/;t
              s/^@.*/$changed&$reset/;t
          s/^[0-9].*/$changed&$reset/;t
              s/^!.*/$changed&$reset/;t

              s/^-.*/$removed&$reset/;t
              s/^<.*/$removed&$reset/;t

             s/^\*.*/$added&$reset/;t
             s/^\+.*/$added&$reset/;t
              s/^>.*/$added&$reset/;t

        s/^Only in.*/$file&$reset/;t
        s/^Index: .*/$file&$reset/;t
          s/^diff .*/$file&$reset/;t
"
exit 0
```

2. Assign execution permissions to the script.

```
$ chmod +x lab-B5.sh
```

3. Execute the script.

```
$ ./lab-B5.sh  <FILE1>  <FILE2>
```

2.2.2.6 Transform the given input text.

Aim:

To transform the given input text to lowercase, uppercase and first letter uppercase text.

Steps:

1. Create a script file called "**lab-B6.sh**" with the following code:

```
#!/bin/bash

first_letter_uppercase() {
    echo $1 | awk '{print toupper(substr($0,0,1))tolower(substr($0,2))}'
}

everything_uppercase() {
    echo $1 | awk '{print toupper($0)}'
}

everything_lowercase() {
    echo $1 | awk '{print tolower($0)}'
}

if [ -z "$1" ]; then
    echo "ERROR: Input text is not supplied in the arguments" >&2; exit 1;
fi

input=$1
echo "First Letter Capital: $(first_letter_uppercase "$input")
Uppercase: $(everything_uppercase "$input")
Lowercase: $(everything_lowercase "$input")"

exit 0
```

2. Assign execution permissions to the script.

```
$ chmod +x lab-B6.sh
```

3. Execute the script.

```
$ ./lab-B6.sh  <TEXT_TO_TRANSFORM>
```


2.3 Working with Cron

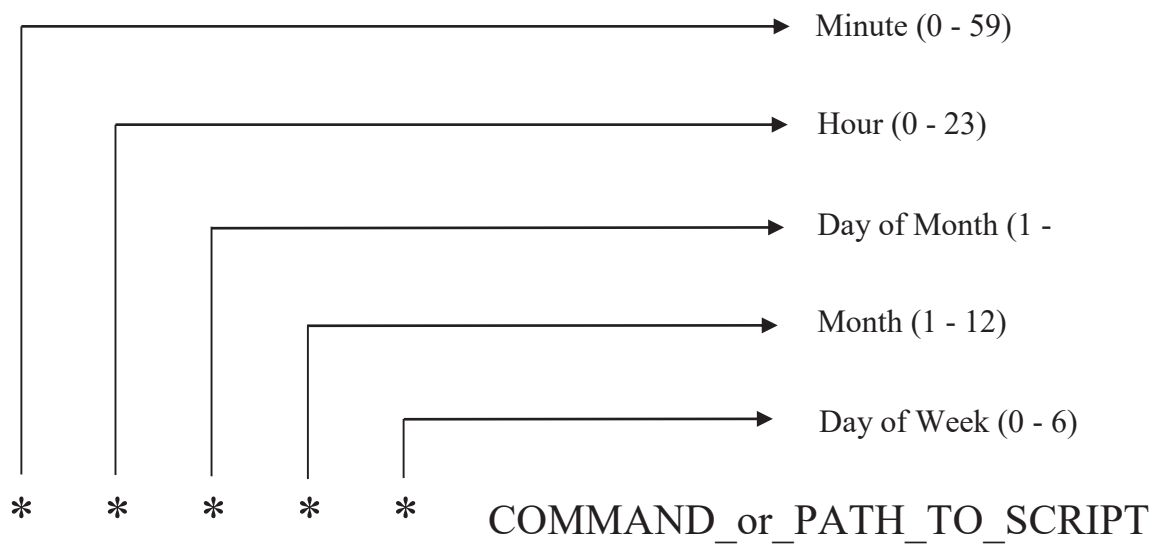
2.3.1 Introduction

Cron is a time-based task scheduler that continuously runs as a service in the Linux Operating System. It is commonly used to run commands or scripts periodically at particular intervals. These jobs are generally scheduled for system maintenance and housekeeping operations.

Each job in Cron follows a syntax usually called Cron Expression. These jobs are scheduled using crontab (Cron Table).

2.3.2 Structure of Cron Expression

The following diagram illustrates the structure of Cron expression.



All Cron expressions have first five parts separated by whitespaces to specify the interval in which the command specified in the sixth part will be executed.

1. The first part denotes the minute of the interval.
2. The second part denotes the hour of the interval.
3. The third part is used to specify the day of the month.
4. The fourth part is used to mention the month itself.
5. The fifth part of the expression denotes the Day of the week.
6. Finally, the command to be executed at the intervals should be specified.
7. Optionally, the year can be added prior to the command.

Each part of the cron allows some special characters as given below.

- Asterisk (*) is used as wildcard i.e. all possible values are used.
- Comma (,) is used to separate items of the list for each field.
- Hyphen (-) is used to define a range of values.
- Slash (/) is used to specify step values.

Apart from this, Cron also has some non-standard predefined definitions that get translated to an interval. These definitions are as follows.

Definition	Description	Translated to
`@yearly`	Run once a year at midnight of 1 January	`0 0 1 1 *`
`@monthly`	Run once a month at midnight of the first day of the month	`0 0 1 * *`
`@weekly`	Run once a week at midnight on Sunday morning	`0 0 * * 0`
`@daily`	Run once a day at midnight	`0 0 * * *`
`@hourly`	Run once an hour at the beginning of the hour	`0 * * * *`
`@reboot`	Run at startup	N/A

2.3.3 Scheduling Cron Expressions

- The Cron expressions can be scheduled using the crontab utility available in the Linux Operating System.
- The following command is used to add a new job or to edit an existing job in cron,

```
$ crontab -e
```


- The following command is used to list the jobs scheduled in cron,

```
$ crontab -l
```

- The following command is used to remove the jobs from cron,

```
$ crontab -r
```

- The following command is used to do the above operations for a specific user,

```
$ crontab -u <USERNAME>
```

2.3.4 Examples

- Executes the script at 1 AM daily.

```
0     1     *     *     *     <PATH_TO_SCRIPT>
```

- Executes the script twice a day.

```
0     9,21  *     *     *     <PATH_TO_SCRIPT>
```

- Executes the script at 9 AM on every Monday.

```
0     9     *     *     mon   <PATH_TO_SCRIPT>
```

- Executes the script every minute.

```
*     *     *     *     *     <PATH_TO_SCRIPT>
```

- Executes the script every 5 minutes.

```
*/5   *     *     *     *     <PATH_TO_SCRIPT>
```

- Scheduling multiple tasks in a single expression.

```
*/5   *     *     *     *     <PATH_TO_SCRIPT>; <PATH_TO_SCRIPT>
```

- Scheduling tasks to execute on the reboot of the system.

```
@reboot     <PATH_TO_SCRIPT>
```

- Redirect the cron results to a specific email account.

```
MAIL=<TARGET_USER>
*     *     *     *     *     <PATH_TO_SCRIPT>
```

2.3.5 Best Practices

The following points should be remembered while working with cron:

- Have a source version control to track and maintain the changes to the cron expressions.
- Organize the scheduled jobs based on its importance or the frequency and group them by their action or the time range.
- Test the scheduled job by having a high frequency initially.
- Do not write complex code or several pipings and redirection in the cron expression directly. Instead, write them to a script and schedule the script to the cron tab.
- Use aliases when the same set of commands are frequently repeated.
- Avoid running commands or scripts through cron as a root user.

2.4 Working with Make and Makefiles

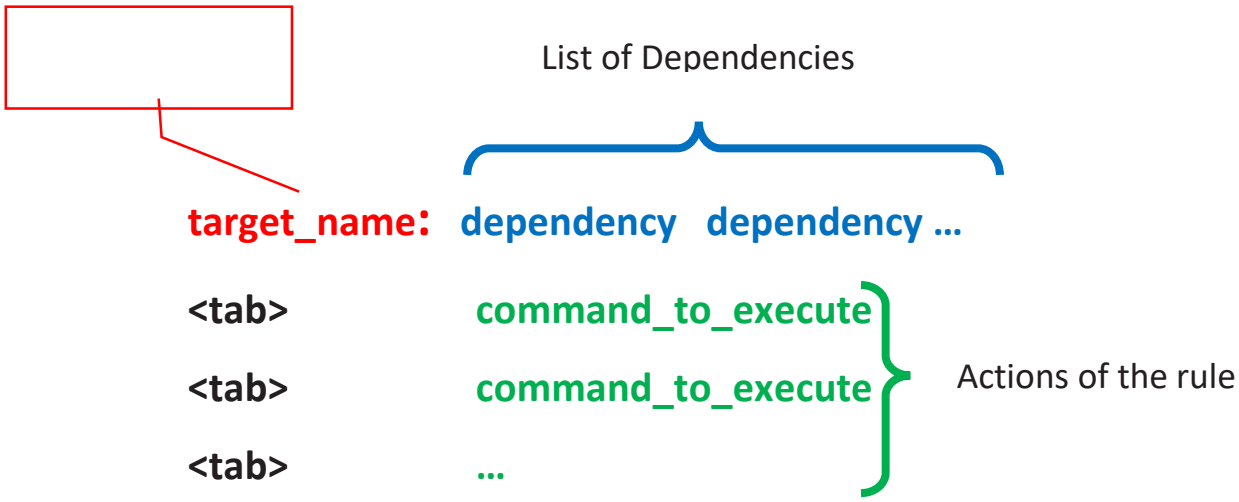
2.4.1 Introduction

It is an automation tool mainly used to compile and construct the binary executable files from a complex source code involving multiple imports and libraries. It uses “Makefile” to know the steps to build the binaries from the source code.

The “Makefile” is just a plain text file containing a series of steps that conform to a predefined syntax. It is also a predecessor of some popular language-specific tools like Ant, Gradle, Rake, etc. Besides, this file provides a way to record building binaries that involve complex processes.

Comparing Bash script, “Makefile” follows the declarative paradigm that can automatically sort out dependencies without explicitly writing logic for all of them. Also, only the modified files will be rebuilt while the unmodified one will be skipped. Thus, the build time and efforts can be minimized a lot.

2.4.2 Structure of Makefile



- A "Makefile" is composed of multiple statements called Rules.
- Each rule should have the target name and a set of actions called recipe prefixed by the tab character. The dependencies are optional and they point to the target of other rules.

2.4.2.1 Target

- A target is the starting point in a rule. Generally, the target is a file and if that file doesn't exist or the file's modified timestamp is newer than the last Make build, then the recipe of that rule gets executed to rebuild the code.
- If the target doesn't represent a file, then it is called as Phony target. Phony targets are the ones in which the target file does not gets generated even after executing its recipe. So, whenever a build is done, the recipe of the Phony target gets executed.
- The "Makefile" has special Built-in target names and some of them are listed below:
  - **.PHONY** is used to explicitly declare that a target is a phony target.
  - **.DEFAULT** will execute its rule if no targets are executed.
  - To quickly rollback on failure, **.DELETE_ON_ERROR** can be used.
  - To continue executing the makefile even in occurrence of errors, **.IGNORE** target name can be used.
  - By default, every line of a recipe will be executed in its own shell in Make. Using **.ONESHELL** will run all the commands of a recipe in the same shell.

2.4.2.2 Macro

- Macro is a feature available in “Makefile” that is similar to a variable in Bash. Just like Bash, a Macro can be defined with a key, equals symbol and a value for the key. i.e. KEY=VALUE
- Global Macros can be accessed anywhere inside the “Makefile”. Their values can be simply referred by enclosing the Macro name within parentheses or braces prefix by a dollar symbol i.e. $(KEY) or ${ KEY }.
- Macros can also have command substitution as value by enclosing the commands within backtick symbol i.e. `COMMAND`. These commands are lazily evaluated and that means, whenever the value is required, then only the command gets executed.

2.4.2.3 Automatic Variables

- These are variables that are automatically available and defined by default for all the rules.
- The value of the variable differs for each and every rule we have. In other words, these variables are local to the rule.
- Some of the important variables are given below:
    - **$@** - filename of the target of the rule. It is used in recipes to create a file with the same name of the target.
    - **$%** - Member name of the target if it is an archive member. It is used along with archive files.
    - **$<** - name of the first specified dependency of a rule. Used to get the dependency of the current rule to do actions on the dependency.
    - **$?** - list of names of dependencies that are newer than the target specified in a rule.
    - **$^** - list of names of all the dependencies specified in a rule.

2.4.2.4 Suffix & Pattern rules

- A suffix rule is used to include all the files ending with a specific extension as the dependency and the same file with a given extension as a target. For example, if there are 50 C language files and you want to compile all of them, instead of adding a rule for each

one, a single suffix rule having .c as dependency extension and .o as the target extension is sufficient.

Example:

```
.c.o:
    cc -c $<
```

- Pattern rules replace the suffix rules since they are more general and supports wildcard characters anywhere in the name of the target. It exactly looks like an ordinary rule. The only difference is that the pattern rule has one "%" character in the target name.
- The "%" character can match any files with a non-empty substring in them. This wildcard pattern can also be used in specifying the dependencies of the rule as well.

  Example:

```
%.o : %.c
   cc -o $@ -c $<
```

- Comparing Suffix rules, the main advantage is that the pattern rules can have prerequisites of their own whereas suffix rules cannot have.

2.4.3 The "Make" command

- The "Make" command is the binary executable file that does the following activities
  - Reads the "Makefile"
  - Resolve the dependencies
  - Form the dependency tree
  - Order the dependencies topologically
  - Do the actions specified in the rules
- Accepts optional arguments to customize the building process of a "Makefile"
- It searches for the "makefile" in the following order:
  i) A file named "makefile" in the current directory
  ii) A file named "Makefile" in the current directory
  iii) Checks for Makefile passed through the "–f" argument.
- Some important arguments to know in "Make"
  - **"–B"** – Unconditionally run the actions of all targets.
  - **"–d"** – Print the debugging information to troubleshoot errors.

- **"–e"** – Supplies the Environment Variables to override the default.
- **"–f"** – Use the given file as the "Makefile".
- **"–i"** – Ignore all errors in command execution while building.
- **"–t"** – Touch files instead of making them by running commands.
- **"–o"** – Don't remake files even if it is older than its dependencies.
- **"–w"** – Debugging the recursive make commands.

2.4.4 Best Practices

The following points should be remembered while using the "make" command:

- Avoid manual user intervention as much as possible.
- Be portable across most of the common operating systems.
- Do not hardcode anything especially inside the recipes of the rules.
- Any static values must be kept as Macros and these macros should be used everywhere.
- Have a separate rule "Install" as a phony target to setup the build environment.
- Never assume that the libraries will exist in the system path. Instead, define macros and provide a complete path to the libraries.
- Avoid compiler assumptions as much as possible.
- Prevent overloading of variables. Instead have all of them defined in macros.
- The "Makefile" should clean the temporary file it creates.
- Avoid platform specific rules as much as possible.
- Reduce using file directories as the target names.
- The tests should always be added in conditionals so that they can be easily controlled.

2.4.5 Examples

2.4.5.1 Basic Setup

Aim:

To create the C source code for building them with Make.

Pre-Requisites:

GCC compiler.

Steps:

1. Create a new folder called “sourcecode”.

```
$ mkdir sourcecode
```

2. Change the current directory to the folder created in the last step.

```
$ cd sourcecode
```

3. Create a file called “**add.c**” with the following code:

```
#include "mymath.h"

int add(int a, int b) {
        return a + b;
}
```

4. Create a file called “**subtract.c**” with the following code:

```
#include "mymath.h"

int subtract(int a, int b) {
        return a - b;
}
```

5. Create a file called “**multiply.c**” with the following code:

```
#include "mymath.h"

int multiply(int a, int b) {
        return a * b;
}
```

6. Create a file called "**divide.c**" with the following code:

```
#include "mymath.h"

int divide(int a, int b) {
        return a / b;
}
```

7. Create a file called "**mymath.h**" with the following code:

```
int add(int a, int b);
int subtract(int a, int b);
int multiply(int a, int b);
int divide(int a, int b);
```

8. Create a file called "**main.c**" with the following code:

```
#include<stdio.h>
#include "mymath.h"

int main() {
        int a, b;
        printf("Hello World\n\n");

        printf("Enter Two Numbers(A B): ");
        scanf("%d %d", &a, &b);

        printf("Addition: %d\n", add(a, b));
        printf("Subtraction: %d\n", subtract(a, b));
        printf("Multiplication: %d\n", multiply(a, b));
        printf("Division: %d\n\n", divide(a, b));

        return 0;
}
```

9. Run the GCC compiler to build the executable from the source file.

```
$ gcc -o mycalculator main.c mymath.h add.c subtract.c multiply.c divide.c
```

10. Execute the above-generated binary to check if it is working fine. Provide two numbers when prompted.

```
$ ./mycalculator
```


2.4.5.2 Building binary from source code

Aim:

To build the binary file from the C source code.

Pre-Requisites:

- GNU Make utility
- C compiler

Steps:

1. Change the current directory to the folder created in the basic setup where the C source files are located.

```
$ cd sourcecode
```

2. Create a file called "**Makefile**" with the following code:

```
mycalculator: main.c mymath.h add.c subtract.c multiply.c divide.c
    gcc -o mycalculator main.c mymath.h add.c subtract.c multiply.c divide.c
```

3. Run the make command and let it generate the binary for you.

```
$ make
```

4. Execute the generated binary and provide two numbers when prompted.

```
$ ./mycalculator
```

2.4.5.3 Transforming the Makefile to adhere to Best Practices

Aim:

To transform the Makefile from the previous lab to implement the best practices.

Pre-Requisites:

- GNU Make utility
- C compiler

Steps:

i. Change the current directory to the folder created in the basic setup where the C source files are located.

```
$ cd sourcecode
```

ii. Edit the "**Makefile**" with the following code:

```
INCL    = mymath.h
SRC     = main.c add.c subtract.c multiply.c divide.c
OBJ     = $(SRC:.c=.o)
EXE     = mycalculator

# Compiler, Linker Defines
CC      = /usr/bin/gcc
CFLAGS  = -ansi -pedantic -Wall -O2
LIBPATH = -L.
LDFLAGS = -o $(EXE) $(LIBPATH) $(LIBS)
CFDEBUG = -ansi -pedantic -Wall -g -DDEBUG $(LDFLAGS)
RM      = /bin/rm -f

# Compile and Assemble C Source Files into Object Files
%.o: %.c
        $(CC) -c $(CFLAGS) $*.c

# Link all Object Files with external Libraries into Binaries
$(EXE): $(OBJ)
        $(CC) $(LDFLAGS) $(OBJ)

# Objects depend on these Libraries
$(OBJ): $(INCL)

# Create a gdb/dbx Capable Executable with DEBUG flags turned on
debug:
        $(CC) $(CFDEBUG) $(SRC)

# Clean Up Objects, Exectuables, Dumps out of source directory
clean:
        $(RM) $(OBJ) $(EXE) core a.out
```

iii. Run the make command and let it generate the binary for you.

```
$ make
```

iv. Execute the generated binary and provide two numbers when prompted.

```
$ ./mycalculator
```

v. Finally, to remove the temporary files, run the following command:

```
$ make clean
```


2.4.5.4 Archiving logs with Make

Aim:

To archive the log files and to move them to separate directory.

Pre-Requisites:

- GNU Make utility
- Tar utility

Steps:

1. Create a file called "**Makefile**" with the following code:

```
SHELL = /bin/bash
LOGDIR = /var/log
LOGEXT = log
BACKUPDIR = /archives

start: createarchive movearchive deletelogs
    echo "Logs files are archived and deleted"

createarchive:
    tar czf archive.tar.gz $(find $(LOGDIR) -name "*.$(LOGEXT)")

movearchive:
    mv archive.tar.gz $(BACKUPDIR)/$(date +%F).tar.gz

deletelogs:
    rm $(find $(LOGDIR) -name "*.$(LOGEXT)")
```

2. Run the make command with the "start" phony target.

```
$ make start
```

3. The above command calls that "start" phony target that depends on few other phony targets. In the end, all the targets get executed so that our logs files will be archived, archives will be moved and the actual logs files get deleted.

2.4.5.5 Conditionals in Makefile

Aim:

To use the conditional in Makefile to enable/disable the debug mode based on the value of a macro.

Pre-Requisites:

- GNU Make utility

Steps:

1. Change the current directory to the folder created in the basic setup where the C source files are located.

```
$ cd sourcecode
```

2. Create a file called "**newmake.mk**" with the following code:

```
DEBUG = True

SHELL = /bin/bash
CC = /usr/bin/gcc
OBJECTS = main.o add.o subtract.o multiply.o divide.o

ifeq ($(DEBUG),True)
    CFLAG = -Wall -g
else
    CFLAG = ""
endif

mycalculator: ${OBJECTS}
    ${CC} ${CFLAGS} -o $@ ${OBJECTS}

%.o : %.c
    ${CC} ${CFLAGS} -o $@ -c $<
```

3. Run the make command with the argument that points to the new makefile.

```
$ make -f newmake.mk
```

4. Execute the generated binary and provide two numbers when prompted.

```
$ ./mycalculator
```


2.5 Error messages for users

2.5.1 Standard Output and Standard Error

- Standard output is a stream that can be used to display any general information such as the running process or its progress, recommendations or suggestions to the user, the help message or even the low-level warnings.
- All the debug information should always be sent to the standard output stream.
- The Standard Error is a stream specifically meant for having error messages that can help the users to debug and troubleshoot the errors.
- This stream (*stderr*) is independent of the standard output and can be redirected independently.
- This is very helpful when the output of a program is piped to another program. In this case, only the *stdout* of the first program will be fed into the second one while the *stderr* messages will be directly sent to the terminal.

2.5.2 Redirecting standard streams

- To redirect the *stdout* to a file

```
$ command > FILE
```

- To redirect the *stderr* to a file

```
$ command 2> File
```

- To redirect *stderr* to *stdout* and the *stdout* to a file

```
$ command > FILE  2>&1
```

- To append both *stdout* and *stderr* to the same file

```
$ command &>> FILE
```

2.5.3 Linux exit codes

- An exit code, also known as a return code, is the code returned to the parent process by an executable program on exit.
- On POSIX compliant systems, the standard exit code for a successful execution is 0 and for any errors, the exit code can be in the range of 1 to 255.

- If no exit code is specified on exit, then the exit code of the last execution will be passed.
- The variable (**$?**) can be used to access the exit code of the last execution.
- The **exit** command is used to exit the program with an exit code.
  Example: **exit 0**
- The standard error codes for Linux are defined as given below:
  - 1 - Catchall for general errors
  - 2 - Misuse of shell builtins (according to Bash documentation)
  - 126 - Command invoked cannot execute
  - 127 - "command not found"
  - 128 - Invalid argument to exit
  - 128+n - Fatal error signal "n"
  - 130 - Script terminated by Control-C
  - 255* - Exit status out of range
- The following method can be used to suppress any errors and the error exit code:

```
$ command 2>/dev/null || exit 0
```

2.5.4 Good error messages

- The error message should clearly define what was the problem and how to fix it.
- The message should be polite, understandable and user-friendly.
- Should not have any difficult words that are uncommon unless required.
- The message should not be generic. It should be specific about the exact reason of the error.
- Show the error message in the right place at the right time.
- If ANSI colours are supported by the terminal, highlight the errors with bright colours to get the attention of the users.
- Use appropriate Linux exit codes so that other programs can get a significant idea about the reason of the error.

2.6 Creating reusable library scripts for automation scripts

In most of the scripts, some set of functions should be repeated again and again multiple times in multiple scripts. Rewriting the same logic everywhere is tedious and becomes a nightmare if some portion of the logic should be modified in all the scripts.

Splitting a huge script into multiple smaller scripts greatly helps in managing them and it enables reusability. Functions provide the way to isolate the actions and load them anywhere we need.

2.6.1 Examples

2.6.1.1 Formatting the Terminal

Aim:

To write a script that provides a set of functions to format the terminal as listed below:

- Colourize the given text
- Show progress bar that fills over a specified time
- Transform the case of the given text

Steps:

1. Create a new folder called "**/opt/lib**" to store our reusable scripts.

```
$ mkdir /opt/lib
```

2. Change the current directory to the folder created in the last step.

```
$ cd /opt/lib
```

3. Create a file called "**format.sh**" with the following code:

```
#!/bin/bash

style() {
    echo "\033[$1m"
}

red_text() { printf "$(style "31")$1$(style "0")"; }
green_text() { printf "$(style "32")$1$(style "0")"; }
yellow_text() { printf "$(style "33")$1$(style "0")"; }
blue_text() { printf "$(style "34")$1$(style "0")"; }
cyan_text() { printf "$(style "36")$1$(style "0")"; }



progress_bar() {
    if [ -z "$1" ] ; then
        echo "ERROR: The duration of the progress bar is required" >&2;
        exit 1;
    fi
    local duration=$1
    local increment=$((100/$duration))
    for (( elapsed=0; elapsed<=100; elapsed=elapsed+increment )); do
        for ((done=0; done<elapsed; done=done+1)); do
            printf "$(green_text "")";
        done
        for ((remain=elapsed; remain<100; remain=remain+1)); do
            printf "$(cyan_text "-")";
        done
        printf "| $(yellow_text "$elapsed") %%" ;
        sleep 1
        printf "\r";
    done
    echo -e "\n";
}

camelcase() {
    echo $1 | awk '{print toupper(substr($0,0,1))tolower(substr($0,2))}'
}

uppercase() { echo $1 | awk '{print toupper($0)}' }

lowercase() { echo $1 | awk '{print tolower($0)}' }
```


4. Assign execution permissions to the script.

```
$ chmod +x format.sh
```

5. Create a new script file called “test.sh” that reuses the functions in the format.sh script.

```
#!/bin/bash

. /opt/lib/format.sh

uppercase "hello world"
camelcase "how are you?"
yellow_text "Wait for 10 seconds"
echo ""
progress_bar 10
```

6. Assign execution permissions to the test script.

```
$ chmod +x test.sh
```

7. Execute the test script.

```
$ ./test.sh
```

2.6.1.2 Utilities library to reduce logic duplication

Aim:

To write a script that provides a list of utilities to reuse across the scripts. The utilities are listed below:

- Check whether the given user exists
- Check whether the root permission is given
- Exit the function with an error message
- Check whether a given path is valid and is a file

Steps:

1. Create a new folder called “**/opt/lib**” to store our reusable scripts if it is not available.

```
$ mkdir /opt/lib
```

2. Change the current directory to the folder created in the last step.

```
$ cd /opt/lib
```

3. Create a file called "**utils.sh**" with the following code:

```
#!/bin/bash

declare -r TRUE=0
declare -r FALSE=1

die() {
    echo "$1" >&2; exit 1
}

has_root_permission() {
   [ $(id -u) -eq 0 ] && return $TRUE || return $FALSE
}

does_user_exit() {
    local username="$1"
    grep -q "^${username}" /etc/passwd && return $TRUE || return $FALSE
}

is_valid_file() {
    local given_path="$1"
    [ -f "$(realpath $given_path)" ] && return $TRUE || return $FALSE
}
```

4. Assign execution permissions to the script.

```
$ chmod +x utils.sh
```

5. Create a new script file called "test.sh" that reuses the functions in the format.sh script.

```
#!/bin/bash

. /opt/lib/utils.sh


has_root_permission && echo "You are a root user" || echo "You are not a root
user."

does_user_exit "ubuntu" && echo "User exist" || echo "User doesn't exist"

is_valid_file "/etc/passwd" &&  echo "Valid File" || echo "Invalid File"

die "Some random error here"
```


6. Assign execution permissions to the test script.

```
$ chmod +x test.sh
```

7. Execute the test script.

```
$ ./test.sh
```

