Lab Reports

OS (Operating System)

BSSE- 4 B (Morning)

Submitted By:

- Maheen Nasir (12315)
- Soma Nasir (12322)

Submitted To:

Sir Talal-Bin-Afzal

Date: 30th Dec 2021

Lab Report #01

Objective:

Objective of this lab is to give some background of Linux

- Introduction to Linux
- Introduction to file system
- Advantages of Linux

Topics discussed:

- Introduction to Linux
- Introduction to File system
- Advantages of Linux

Introduction to Linux:

"LINUX is an operating system or a kernel distributed under an open-source license. Its functionality list is quite like UNIX".
The kernel is a program at the heart of the Linux operating system that takes care of fundamental stuff, like letting hardware communicate with software.
Linux is an operating system or a kernel which germinated as an idea in the mind of young and bright Linus Torvalds when he was a computer science student. He used to work on the UNIX OS (proprietary software) and thought that it needed improvements.
However, when his suggestions were rejected by the designers of UNIX, he thought of launching an OS which will be receptive to changes, modifications suggested by its users.

Introduction to File System:

A file is a collection of correlated information which is recorded on secondary or non-volatile storage like magnetic disks, optical disks, and tapes. It is a method of data collection that is used as a medium for giving input and receiving output from that program.
In general, a file is a sequence of bits, bytes, or records whose meaning is defined by the file creator and user. Every File has a logical location where they are located for storage and retrieval.

Advantages of Linux:

The overall benefit of Linux is that;

- Its open source, which means its source code, is accessible to anyone who wants it.
- Anyone with the ability to code can contribute, modify, enhance, and distribute Linux for any intended use.
- Linux has a huge online community with vast numbers of forums providing support and sharing knowledge extensively.

Exercise:

Q1: What is the file system used by the Linux?

Ans: ext4 file system

The majority of modern Linux distributions default to the ext4 filesystem

Q2) Name two types of boot loaders available.

Ans: If a computer is to be used with Linux; a special boot loader must be installed. For Linux, the two most common boot loaders are known as LILO (Linux Loader) and LOADLIN (LOAD Linux)

Q3: What are the names of partitions created for Linux?

Ans: There are two kinds of major partitions on a Linux system:

- Data partition: normal Linux system data, including the root partition containing all the data to start up and run the system; and.
- Swap partition: expansion of the computer's physical memory, extra memory on hard disk.

Lab Report #02

Lab objective:

This lab introduces few of the basic commands of Linux.

Linux Command Line

The shell is where commands are invoked

A command is typed at a shell prompt

- Prompt usually ends in a dollar sign ($)

After typing a command press Enter to invoke it

- The shell will try to obey the command

Another prompt will appear

Creating Files with cat

There are many ways of creating a file

One of the simplest is with the cat command:

~$ cat> file

Soma

Maheen

```
~$ cat> file
Soma
Maheen
~$
```

Displaying file contents with cat

~$ cat file

Soma

Maheen

```
~$ cat file
Soma
Maheen
~$
```

Deleting Files with rm

To delete a file, use the rm (‘remove’) command

Simply pass the name of the file to be deleted as an argument:

~$ rm file

```
~$ rm file
~$
```

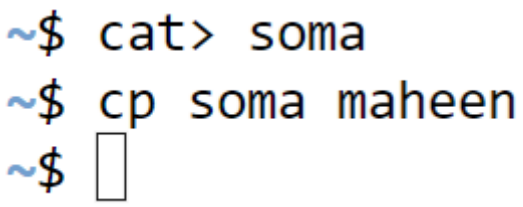
Copying and Renaming Files with cp and mv

To copy the contents of a file into another file, use the cp command:

~$ cat> soma

~$ cp soma maheen

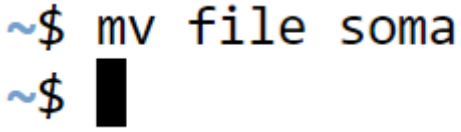
~$



To rename a file use the mv ('move') command:

~$ mv file soma

~$



Lab Report #03

Introduction to Shell:

The shell is a program that provides the user with an interface to use the operating system's functions through some commands. A shell script is a program that is used to perform specific tasks.

Introduction to bash:

Bash is an acronym of "Bourne-Again Shell". It is a default command-line interpreter for UNIX and Linux based operating systems. In UNIX and Linux based operating systems, a terminal window is consisting of a shell and Bash.

To execute the bash scripting file, first we need to change the permissions of the file and make it executable. The permission is changed by chmod +x command.

The bash script file can be executed in two ways:

Bash filename

. /filename

To execute the bash file, navigate to the folder or directory where the bash file is saved. In the first way of execution simply write bash and then the filename on the terminal and hit enter. The bash script file will be executed.

In the second way, write ". /filename" on the terminal and press enter.

Commands:

- **ls(List directory contents):**

ls are probably the most common command. A lot of times, you'll be working in a directory and you'll need to know what files are located there. The ls command allows you to quickly view all files within the specified directory.

- Syntax: ls [option(s)] [file(s)]
- Common options: -a, -l

echo (Prints text to the terminal window):

echo prints text to the terminal window and is typically used in shell scripts and batch files to output status text to the screen or a computer file. Echo is also particularly useful for showing

the values of environmental variables, which tell the shell how to behave as a user works at the command line or in scripts.

- Syntax: echo [option(s)] [string(s)]
- Common options: -e, -n

mkdir(Create a directory)

mkdir is a useful command you can use to create directories. Any number of directories can be created simultaneously which can greatly speed up the process.

- Syntax: mkdir [option(s)] directory_name(s)
- Common options: -m, -p, -v

Pwd: Print working directory

```
~$ pwd
/home/user
~$
```

pwd is used to print the current directory you're in. As an example, if you have multiple terminals going and you need to remember the exact directory you're working within, then pwd will tell you.

- Syntax: pwd [option(s)]
- Common options: options aren't typically used with pwd

cd(Change directory)

cd will change the directory you're in so that you can get info, manipulate, read, etc. the different files and directories in your system.

- Syntax: cd [option(s)] directory
- Common options: options aren't typically used with cd

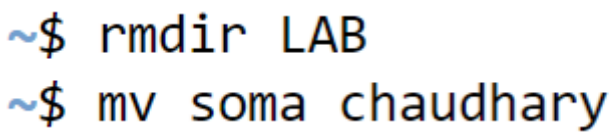
mv(Move or rename directory)

```
~$ mkdir OS
~$ mv OS LAB
~$
```

mv is used to move or rename directories. Without this command, you would have to individually rename each file which is tedious. mv allows you to do batch file renaming which can save you loads of time.

- Syntax: mv [option(s)] argument(s)
- Common options: -i, -b

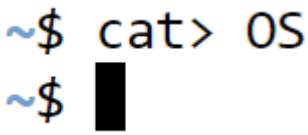
rmdir(Remove directory)



rmdir will remove empty directories. This can help clean up space on your computer and keep files and folders organized. It's important to note that there are two ways to remove directories: rm and rmdir. The distinction between the two is that rmdir will only delete empty directories, whereas rm will remove directories and files regardless if they contain data or not.

- Syntax: rmdir [option(s)] directory_names
- Common options: -p

cat(Read a file, create a file, and concatenate files)



cat is one of the more versatile commands and serves three main functions: displaying them, combining copies of them, and creating new ones.

- Syntax: cat [option(s)] [file_name(s)] [-] [file_name(s)]
- Common options: -n

exit(Exit out of a directory)

The exit command will close a terminal window, end the execution of a shell script, or log you out of an SSH remote access session.

- Syntax: exit
- Common options: n/a

cp — copy files and directories

Use this command when you need to back up your files.

- Syntax: cp [option(s)] current_name new_name
- Common options: -r, -i, -b

Lab Report #04

Using Dot Directories in Paths

The special and .directories can be used in paths just like any other directory name:

$ cd ../other-dir/

Meaning "the directory other-dir in the parent directory of the current directory"

It is common to see .. used to 'go back' several directories from the current directory:

$ ls ../../../../far-away-directory/

The . directory is most commonly used on its own, to mean "the current directory"

Hidden Files:

Make ls display all files, even the hidden ones, by giving it the -a (all) option:

$ ls -a

```
~$ ls -a
.                               .bash_profile           2022-01-04-194404.term
..                              .bashrc                 2022-01-04-195837.term
.2022-01-04-191958.term-0.term  .jupyter-blobs-v0.db    2022-01-04-205158.term
.2022-01-04-193940.term-0.term  .smc                    OS
.2022-01-04-194404.term-0.term  .snapshots              chaudhary
.2022-01-04-195837.term-0.term  .ssh                    lab
.2022-01-04-205158.term-0.term  2022-01-04-191958.term  labb
.bash_history                   2022-01-04-193940.term  maheen
~$
```

Hidden files are often used for configuration files

Usually found in a user's home directory

Running program

Programs under Linux are files, stored in directories like /bin and /usr/bin

Run them from the shell, simply by typing their name

Many programs take options, which are added after their name and prefixed with

For example, the -l option to ls gives more information, including the size of files and the date they were last modified:

$ ls –l

```
~$ ls -1
2022-01-04-191958.term
2022-01-04-193940.term
2022-01-04-194404.term
2022-01-04-195837.term
2022-01-04-205158.term
OS
chaudhary
lab
labb
maheen
~$
```

Exercise:

Q1: Start another shell. Enter each of the following commands in turn.

- **date**

```
~$ date
Tue Jan  4 16:15:43 UTC 2022
~$
```

- **whoami**

```
~$ whoami
user
~$
```

- **hostname**

```
~$ hostname
project-c7bfc82f-0df8-4dc5-8ef8-8b363b6d62e0
~$
```

- **uname**

```
~$ uname
Linux
~$
```

- **uptime**

```
~$ uptime
 16:18:20 up 19:58,  0 users,  load average: 1.28, 1.55, 1.50
~$
```

Q2:

a. Use the ls command to see if you have any files.

```
~$ ls
2022-01-04-191958.term  2022-01-04-194404.term  2022-01-04-205158.term  chaudhary  labb
2022-01-04-193940.term  2022-01-04-195837.term  OS                      lab        maheen
~$
```

b. Create a new file using the cat command as follows:

$ cat > hello.txt
Hello world!
This is a text file.

```
~$ cat> hello.txt
Hello World!
This is a text file
```

c. Press Enter at the end of the last line, then Ctrl+D to denote the end of the file.

```
~$ cat> hello.txt
Hello World!
This is a text file
~$
```

d. Use ls again to verify that the new file exists.

```
~$ cat > hello.txt
Hello world!
This is text file
~$ ls
2022-01-04-191958.term  2022-01-04-194404.term  2022-01-04-205158.term  chaudhary  lab   maheen
2022-01-04-193940.term  2022-01-04-195837.term  OS                      hello.txt  labb
~$
```

e. Display the contents of the file.

```
~$ cat hello.txt
Hello World!
This is a text file
```

f. Display the file again, but use the cursor keys to execute the same command again without having to retype it.

```
~$ cat hello.txt
Hello World!
This is a text file
```

Press upper arrow key and then enter key.

Q3:

a. Create a second file. Call it *secret-of-the-universe*, and put in Whatever content you deem appropriate.

```
~$ cat> secret-of-the-universe
```

a. Check its creation with ls.

```
~$ ls
2022-01-04-191958.term  2022-01-04-194404.term  2022-01-04-205158.term  chaudhary  lab   maheen
2022-01-04-193940.term  2022-01-04-195837.term  OS                      hello.txt  labb  secret-of-the-universe
~$
```

c. Display the contents of this file. Minimize the typing needed to do this:

i. Scroll back through the command history to the command
you used to create the file.

Use the up arrow key, there you can find cat > secret-of-the-universe

```
~$ cat> secret-of-the-universe
```

ii. Change that command to display **secret-of-the-universe**

instead of creating it.

Just remove >

```
~$ cat secret-of-the-universe
~$
```

Q4: After each of the following steps, use ls and cat to verify what has happened.

a. Copy *secret-of-the-universe* to a new file called *answer.txt*. Use Tab to avoid typing the existing file's name in full.

```
~$ cp secret-of-the-universe answer.txt
~$
```

b. Now copy *hello.txt* to *answer.txt*. What's happened now?

```
~$ cp hello.txt answer.txt
~$
```

c. Delete the original file, *hello.txt*.

```
~$ rm hello.txt
~$ ls
2022-01-04-191958.term  2022-01-04-195837.term  chaudhary  maheen
2022-01-04-193940.term  2022-01-04-205158.term  lab        secret-of-the-universe
2022-01-04-194404.term  OS                      labb
~$
```

d. Rename *answer.txt* to *message*.

```
~$ mv answer.txt message
~$
```

e. Try asking rm to delete a file called *missing*. What happens?

```
~$ rm missing
rm: cannot remove 'missing': No such file or directory
~$
```

f. Try copying *secret-of-the-universe* again, but don't specify a filename to which to copy. What happens now.

```
~$ cp secret-of-the-universe
cp: missing destination file operand after 'secret-of-the-universe'
Try 'cp --help' for more information.
~$
```

Lab No. 05

Lab objective

This lab will give an overview of The Shell as a Programming Language

Background

Shells

- A **shell** provides an interface between the user and the operating system kernel
- Either a **command interpreter** or a graphical user interface
- Traditional Unix shells are **command-line interfaces** (CLIs) Usually started automatically when you log in or open a terminal There are two ways of writing shell programs.
- You can type a sequence of commands and allow the shell to execute them interactively.
- Or you can store those commands in a file that you can then invoke as a program.

Creating a Script

- Using any text editor, you need to create a file containing the commands; create a file called first that looks like this:

```
1  #!/bin/sh
2  # first
3  # This file looks through all the files in the current
4  # directory for the string POSIX, and then prints the names of
5  # those files to the standard output.
6  for file in *
7  do
8  if grep -q POSIX $file
9  then
10 echo $file
11 fi
12 done
13 ••••••••••••••••••••••••••••••••exit 0
14
```

- Comments start with a # and continue to the end of a line.
- Conventionally, though, # is kept in the first column.
- The first line, #!/bin/sh, is a special form of comment;

- The #! characters tell the system that the argument that follows on the line is the program to be used to execute this file.
- In this case, /bin/sh is the default shell program.
- The exit command ensures that the script returns a sensible exit code (more on this later).
- This is rarely checked when programs are run interactively, but if you want to invoke this script from another script and check whether it succeeded, returning an appropriate exit code is very important.
- Even if you never intend to allow your script to be invoked from another, you should still exit with a reasonable code.
- A zero denotes success in shell programming. Since the script as it stands can't detect any failures, it always returns success.
- Notice that this script does not use any filename extension or suffix; Linux, and UNIX in general, rarely makes use of the filename extension to determine the type of a file.
- You could have used .sh or added a different extension, but the shell doesn't care. Most preinstalled scripts will not have any filename extension,

Making a Script Executable

- The simpler way is to invoke the shell with the name of the script file as a parameter:

$ /bin/sh first

- Do this by changing the file mode to make the file executable for all users using the chmod command:

```
~$ chmod +x file
```

Execute a Script

```
~$ ./file
grep: Lab: No such file or directory
grep: Reports: No such file or directory
grep: .term: No such file or directory
file
```

Shell Syntax

- You can use the bash shell to write quite large, structured programs.

  The next few sections cover the following:

- Variables: strings, numbers, environments, and parameters
- Conditions: shell Booleans
- Program control: if, elif, for, while, until, case
- Lists
- Functions

Variables

- You don't usually declare variables in the shell before using them.
- Instead, you create them by simply using them (for example, when you assign an initial value to them). By default, all variables are considered and stored as strings, even when they are assigned numeric values.
- The shell and some utilities will convert numeric strings to their values in order to operate on them as required.
- Linux is a case-sensitivesystem, so the shell considers the variable foo to be different from Foo, and both to be different from FOO.

Access the contents of a variable

- By Within the shell you can access the contents of a variable by preceding its name with a $.
- Whenever you extract the contents of a variable, you must give the variable a preceding $.
- When you assign a value to a variable, just use the name of the variable, which is created dynamically if necessary.
- An easy way to check the contents of a variable is to echo it to the terminal, preceding its name with a $.
- On the command line, you can see this in action when you set and check various values of the variable salutation:

```
~$  salutation=Hello
~$  echo $salutation
Hello
~$  salutation="Yes Dear"
~$  echo $salutation
Yes Dear
~$  salutation=7+5
~$  echo $salutation
7+5
```

- Note how a string must be delimited by quote marks if it contains spaces. In addition, there can’t be any spaces on either side of the equals sign.

Lab No. 06

Assign user input to a variable

- You can assign user input to a variable by using the **read** command.
- This takes one parameter, the name of the variable to be read into, and then waits for the user to enter some text.
- The read normally completes when the user presses Enter. When reading a variable from the terminal, you don't usually need the quote marks:

```
~$ read salutation
Hello World!
~$ echo $salutation
Hello World!
```

Quoting

- Normally, parameters in scripts are separated by whitespace characters (e.g., a space, a tab, or a newline character).
- If you want a parameter to contain one or more whitespace characters, you must quote the parameter.
- The behavior of variables such as **$foo** inside quotes depends on the type of quotes you use.
- If you enclose a $ variable expression in double quotes, then it's replaced with its value when the line is executed.
- If you enclose it in single quotes, then no substitution takes place. You can also remove the special meaning of the $ symbol by prefacing it with a \.
- Usually, strings are enclosed in double quotes, which protects variables from being separated by white space but allows $ expansion to take place.

```
1  #!/bin/sh
2  myvar="Hi there"
3  echo $myvar
4  echo "$myvar"
5  echo '$myvar'
6  echo \$myvar
7  echo Enter some text
8  read myvar
9  echo '$myvar'
10 read myvar
11 echo '$myvar' now equals $myvar
12 exit 0
```

This behaves as follows:

$ ./variable

```
Hi there
"Hi there"
$myvar
$myvar
Enter some text
Hello World
$myvar
Hello
$myvar now equals Hello
```

How It Works

- The variable myvar is created and assigned the string Hi there.
- The contents of the variable are displayed with the echo command, showing how prefacing the variable with a $ character expands the contents of the variable.
- You see that using double quotes doesn't affect the substitution of the variable, while single quotes and the backslash do.
- You also use the read command to get a string from the user.

Environment Variables

- When a shell script starts, some variables are initialized from values in the environment.
- These are normally in all uppercase form to distinguish them from user-defined (shell) variables in scripts, which are conventionally lowercase.
- The variables created depend on your personal configuration. Many are listed in the manual pages, but the principal ones are listed in the following table:

Environment Variable	Description
$HOME	The home directory of the current user
$PATH	A colon-separated list of directories to search for commands
$PS1	A command prompt, frequently $, but in bash you can use some more complex values; for example, the string [\u@\h \W]$ is a popular default that tells you the user, machine name, and current directory, as well as providing a $ prompt.
$PS2	A secondary prompt, used when prompting for additional input; usually >.
$IFS	An input field separator. This is a list of characters that are used to separate words when the shell is reading input, usually space, tab, and newline characters.
$0	The name of the shell script
$#	The number of parameters passed
$$	The process ID of the shell script, often used inside a script for generating unique temporary filenames; for example /tmp/tmpfile_$$

Parameter Variables

- If your script is invoked with parameters, some additional variables are created. If no parameters are passed, the environment variable $# still exists but has a value of 0.
- The parameter variables are listed in the following table:

Parameter Variable	Description
$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. If IFS is modified, then the way $* separates the command line into parameters will change.
$@	A subtle variation on $*; it doesn't use the IFS environment variable, so parameters are not run together even if IFS is empty.

It's easy to see the difference between $@ and $* by trying them out:

```
~$ IFS=''
~$ set foo bar bam
~$ echo "$*"
foobarbam
~$ unset IFS
~$ echo "$*"
foo bar bam
```

- Within double quotes, $@ expands the positional parameters as separate fields, regardless of the IFS value.
- In general, if you want access to the parameters, $@ is the sensible choice.
- In addition to printing the contents of variables using the echo command, you can also read them by using the read command.

Lab No. 07

Manipulating Parameter and Environment Variables

- The following script demonstrates some simple variable manipulation.

```
#!/bin/sh
IFS=''
set foo bar bam
echo "$*"
foo bar bam
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 you run this script, you get the following output:

```
Hello
The program ./file is now running
The second parameter was bar
The first parameter was foo
The parameter list was foobarbam
The user's home directory is /home/user
Please enter a new greeting
Hey
Hey
The script is now complete
```

Conditions

- Fundamental to all programming languages is the ability to test conditions and perform different actions based on those decisions.
- A shell script can test the exit code of any command that can be invoked from the command line, including the scripts that you have written yourself. That's why it's important to always include an exit command with a value at the end of any scripts that you write.

The test or [Command

- In practice, most scripts make extensive use of the [or test command, the shell's Boolean check.
- On some systems, the [and test commands are synonymous, except that when the [command is used, a trailing] is also used for readability.
- We'll introduce the test command using one of the simplest conditions: checking to see whether exists. The command for this is test -f <filename>, so within a script you can write

```
if test -f
fred.c then
.
.
.
f
i
```

You can also write it like this:

```
if [ -f fred.c
] then
.
.
.
f
i
```

- The test command's exit code (whether the condition is satisfied) determines whether the conditional code is run.
- Note that you must put spaces between the [braces and the condition being checked.
- You can remember this by remembering that [is just the same as writing test, and you would always leave a space after the test command.

- If you prefer putting then on the same line as if, you must add a semicolon to separate the test from the then :

if [-f fred.c]; then

.

.

.

f

i

- The condition types that you can use with the test command fall into three types: string comparison, arithmetic comparison, and file conditionals.
- The following table describes these condition types:

String Comparison	Result
`string1 = string2`	True if the strings are equal
`string1 != string2`	True if the strings are not equal
`-n string`	True if the string is not null
`-z string`	True if the string is null (an empty string)

Arithmetic Comparison	Result
`expression1 -eq expression2`	True if the expressions are equal
`expression1 -ne expression2`	True if the expressions are not equal
`expression1 -gt expression2`	True if `expression1` is greater than `expression2`
`expression1 -ge expression2`	True if `expression1` is greater than or equal to `expression2`
`expression1 -lt expression2`	True if `expression1` is less than `expression2`
`expression1 -le expression2`	True if `expression1` is less than or equal to `expression2`

File Conditional	Result
`-d file`	True if the file is a directory
`-e file`	True if the file exists. Note that historically the `-e` option has not been portable, so `-f` is usually used.
`-f file`	True if the file is a regular file
`-g file`	True if `set-group-id` is set on `file`
`-r file`	True if the file is readable
`-s file`	True if the file has nonzero size
`-u file`	True if `set-user-id` is set on `file`
`-w file`	True if the file is writable
`-x file`	True if the file is executable

- Following is an example of how you would test the state of the file /bin/bash, just so you can see what these look like in use:

```
1  #!/bin/sh
2  if [ -f /bin/bash ]
3  then
4  echo "file /bin/bash exists"
5  fi
6  if [ -d /bin/bash ]
7  then
8  echo "/bin/bash is a directory"
9  else
10 echo "/bin/bash is NOT a directory"
11 fi
```

Output :-

```
file /bin/bash exists
/bin/bash is NOT a directory
```

Control Structures

- The shell has a set of control structures, which are very similar to other programming languages.

if

The if statement is very simple: It tests the result of a command and then conditionally executes a group of statements:

if

condition

then

statement

s

else

statement

s

fi

- A common use for if is to ask a question and then make a decision based on the answer:

```
#!/bin/sh
echo "Is it morning? Please answer yes or no"
read timeofday
if [ $timeofday = "yes" ];
then
echo "Good morning"
else
echo "Good afternoon"
fi
exit 0
```

This would give the following output:

```
Is it morning? Please answer yes or no
no
Good afternoon
```

elif

- It allows you to add a second condition to be checked when the else portion of the if is executed.
- You can modify the previous script so that it reports an error message if the user types in anything other than yes or no. Do this by replacing the else with elif and then adding another condition:

```
#!/bin/sh
echo "Is it morning? Please answer yes or no"
read timeofday
if [ $timeofday = "yes" ]
then
echo "Good morning"
elif [ $timeofday = "no" ];
then
echo "Good afternoon"
else
echo "Sorry, $timeofday not recognized. Enter yes or no"
exit 1
fi
exit 0
```

```
Is it morning? Please answer yes or no
no
Good afternoon


Is it morning? Please answer yes or no
dk
Sorry, dk not recognized. Enter yes or no
```

Lab No. 08

for

- Use the for construct to loop through a range of values, which can be any set of strings.
- They could be simply listed in the program or, more commonly, the result of a shell expansion of filenames.
- The syntax is simple: **for** variable **in** values **do** statements

done

Using a for Loop with Fixed Strings

- The values are normally strings, so you can write the following:

```
#!/bin/sh
for foo in bar fud 43
do
echo $foo
done
exit 0
```

That results in the following output:

```
bar
fud
43
```

How It Works

- This example creates the variable foo and assigns it a different value each time around the for loop.
- Since the shell considers all variables to contain strings by default, it's just as valid to use the string 43 as the string fud.

Lab No. 09

while

- Because all shell values are considered strings by default, the for loop is good for looping through a series of strings, but is not so useful when you don't know in advance how many times you want the loop to be executed.
- When you need to repeat a sequence of commands, but don't know in advance how many times they should execute, you will normally use a **while loop**, which has the following syntax:

while condition do

statements

done

- For example, here is a rather poor password-checking program:

```
#!/bin/sh
echo "Enter password"
read trythis
while [ "$trythis" != "secret" ];
do
echo "Sorry, try again"
read trythis
done
exit 0
```

An example of the output from this script is as follows:

```
Enter password
abcde
Sorry, try again
secret
```

- Clearly, this isn't a very secure way of asking for a password, but it does serve to illustrate the while statement.

- The statements between do and done are continuously executed until the condition is no longer true. In this case, you're checking whether the value of trythis is equal to secret.
- The loop will continue until $trythis equals secret. You then continue executing the script at the statement immediately following the done.
- This means using a wildcard for the string value and letting the shell fill out all the values at run time.

case

- The case construct is a little more complex than those you have encountered so far. Its syntax is as follows:
- case variable in pattern [| pattern] ...) statements;; pattern [| pattern] ...) statements;; ...

  Esac
- This may look a little intimidating, but the case construct enables you to match the contents of a variable against patterns in quite a sophisticated way and then allows execution of different statements, depending on which pattern was matched.
- It is much simpler than the alternative way of checking several conditions, which would be to use multiple if, elif, and else statements.
- Notice that each pattern line is terminated with double semicolons (;;). You can put multiple statements between each pattern and the next, so a double semicolon is needed to mark where one statement ends and the next pattern begins.
- The capability to match multiple patterns and then execute multiple related statements makes the case construct a good way of dealing with user input.
- The best way to see how case works is with an example.

Case I : User Input

- You can write a new version of the input-testing script and, using the case construct, make it a little more selective and forgiving of unexpected input:

```
#!/bin/sh
echo "Is it morning? Please answer yes or no"
read timeofday
case "$timeofday"
in
yes)
echo "Good Morning";;
no )
echo "Good Afternoon";;
y )
echo "Good Morning";;
n )
echo "Good Afternoon";;
* )
echo "Sorry, answer not recognized";;
esac
exit 0
```

Output :

```
~$ ./file
Is it morning? Please answer yes or no
no
Good Afternoon
~$ ./file
Is it morning? Please answer yes or no
*
Sorry, answer not recognized
```

How It Works

- When the case statement is executing, it takes the contents of timeofday and compares it to each string in turn.
- As soon as a string matches the input, the case command executes the code following the) and finishes.
- The case command performs normal expansion on the strings that it's using for comparison. You can therefore specify part of a string followed by the * wildcard. Using a single * will match all possible strings, so always put one after the other matching strings to ensure that the case statement ends with some default action if no other strings are matched.

- This is possible because the case statement compares against each string in turn. It doesn’t look for a best match, just the first match.
- The default condition often turns out to be the impossible condition, so using * can help in debugging scripts.

Lab No. 10

The AND List

- The AND list construct enables you to execute a series of commands, executing the next command only if all the previous commands have succeeded.
- The syntax is statement1 **&&** statement2 **&&** statement3 **&& ...**
- Starting at the left, each statement is executed; if it returns true, the next statement to the right is executed.
- This continues until a statement returns false, after which no more statements in the list are executed.
- The && tests the condition of the preceding command.
- Each statement is executed independently, enabling you to mix many different commands in a single list, as the following script shows. The AND list as a whole succeeds if all commands are executed successfully, but it fails otherwise.

```
1  #!/bin/sh
2  touch file_one
3  rm -f file_two
4  if [ -f file_one ] && echo "hello" && [ -f file_two ] && echo "there"
5  then
6  echo "in if"
7  else
8  echo "in else"
9  fi
10 exit 0
```

Try the script and you'll get the following result:

```
hello
in else
```

The OR List

- The OR list construct enables us to execute a series of commands until one succeeds, and then not execute any more.
- The syntax is as follows: statement1 || statement2 || statement3 || **...**
- Starting at the left, each statement is executed. If it returns false, then the next statement to the right is executed.

- This continues until a statement returns true, at which point no more statements are executed.
- The || list is very similar to the && list, except that the rule for executing the next statement is that the previous statement must fail.

```
#!/bin/sh
rm -f file_one
if [ -f file_one ] || echo "hello" || echo "there"
then
echo "in if"
else
echo "in else"
fi
exit 0
```

- This results in the following output:

```
hello
in if
```

Functions

- You can define functions in the shell; and if you write shell scripts of any size, you'll want to use them to structure your code.
- To define a shell function, simply write its name followed by empty parentheses and enclose the statements in braces:

function_name () {

statements

}

- Let's look at a really simple function:

```
1  #!/bin/sh
2  foo(){
3  echo "Function foo is executing"
4  }
5  echo "script starting"
6  foo
7  echo "script ended"
8  exit 0
```

Running the script will output the following:

```
script starting
Function foo is executing
script ended
```