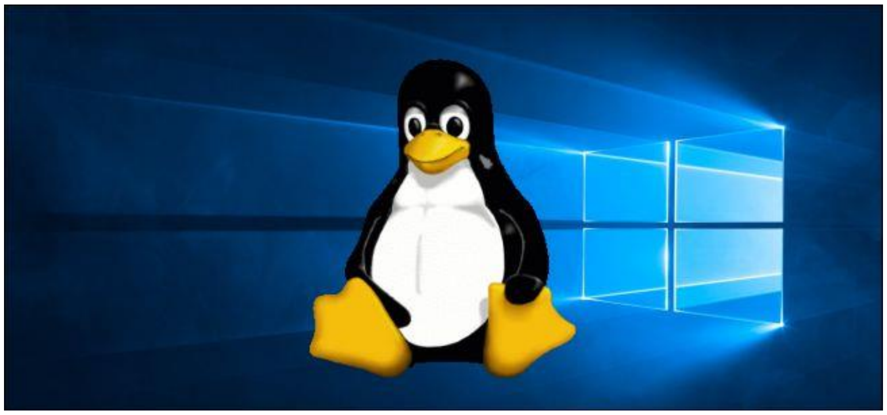
UNIVERSITY OF ENGINEERING AND TECHNOLOGY, TAXILA

FACULTY OF TELECOMMUNICATION AND INFORMATION ENGINEERING

SOFTWARE ENGINEERING DEPARTMENT



Operating Systems

Experiment 6

Shell Programming –II

Switch, Loops and Functions

CLO 2. Use modern tools and languages.
CLO 3. Demonstrate an original solution of problem under discussion.
CLO 4. Work individually as well as in teams

Shell Programming-II

The switch statement:

The case statement is good alternative to multilevel **if-then-else-fi** statement. It enables you to match several values against one variable. It’s easier to read and write.

Syntax:

```
case $variable-name in
   pattern1)
      Statement(s) to be executed if pattern1 matches
      ;;
   pattern2)
      Statement(s) to be executed if pattern2 matches
      ;;
   pattern3)
      Statement(s) to be executed if pattern3 matches
      ;;
   *)
     Default condition to be executed
     ;;
esac
```

The **$variable-name** is compared against the **patterns** until a match is found. The shell then executes all the statements up to the **two semicolons** that are next to each other. The **default is *)** and its executed if no match is found. There is no maximum number of patterns, but the minimum is one.
You must include **;;** at the end of each command. The shell executes all the statements up to the two semicolons that are next to each other. The **esac** is always required to indicate end of case statement This is similar to break in the C programming language.

Example:

```
#!/bin/sh

FRUIT="kiwi"

case "$FRUIT" in
   "apple") echo "Apple pie is quite tasty."
   ;;
   "banana") echo "I like banana nut bread."
   ;;
   "kiwi") echo "New Zealand is famous for kiwi."
   ;;
esac
```

Output:

```
New Zealand is famous for kiwi.
```

Example:

Create a shell script called rental.sh:

```
#!/bin/bash

# if no command line arg given
# set rental to Unknown
if [ -z $1 ] // -z flag causes test to check whether a string is empty.
then
  rental="*** Unknown vehicle ***"
elseif [ -n $1 ]// -n  to test for non-null strings
then
# otherwise make first arg as a rental
  rental=$1
fi

# use case statement to make decision for rental
case $rental in
   "car") echo "For $rental rental is Rs.20 per k/m.";;
   "van") echo "For $rental rental is Rs.10 per k/m.";;
   "jeep") echo "For $rental rental is Rs.5 per k/m.";;
   "bicycle") echo "For $rental rental 20 paisa per k/m.";;
   "enfield") echo "For $rental rental Rs.3  per k/m.";;
   "thunderbird") echo "For $rental rental Rs.5 per k/m.";;
   *) echo "Sorry, I can not get a $rental rental  for you!";;
esac
```

Save and close the file. Run it as follows:

```
chmod +x rental.sh
./rental.sh
./rental.sh jeep
./rental.sh enfield
./rental.sh bike
```

Explanation:

Here first we will check, that if $1(first command line argument) is not given set value of rental variable to "*** Unknown vehicle ***", if value given then set it to given value. The $rental is compared against the patterns until a match is found. Here for first run its match with van and it will show output For van Rs.10 per k/m. For second run it print, "For car Rs.20 per k/m". And for last run, there is no match for Maruti-800, hence default i.e. *) is executed and it prints, "Sorry, I cannot get a Maruti-800 for you". Note that esac is always required to indicate end of case statement.

The for Loop:
The for loop provides a tool for processing a list of input. The input to the for loop is a list of values. Each iteration through the loop it extracts one value into a variable and then enters the body of the loop. The loop stops when the extract fails because there are no more values in the list. Consider the following example which prints each of the command line arguments, one at a time. We'll extract them from "**$@**" into **$var:**

```
for var in "$@" //$@ refers to all of a shell script's command-line arguments. $1 , $2 , etc.
do
printf "%s\n" $var //%s means, "insert the first argument, a string, right here
done
```

Break Statement:
Much like C or Java, shell has a *break* command, also. It can be used to break out of a loop. Consider this example which stops printing command line arguments, when it gets to one whose value is "quit":

```
for var in "$@"
do
if [ "$var" = "quit" ]
then
break
fi
printf "%s\n" $var
done
```

Continue Statement:
Shell has a *continue* that works just like it does in C or Java.

```
for var in "$@"
do
if [ "$var" = "me" ]
then
continue
elif [ "$var" = "mine" ]
then
continue
elif [ "$var" = "myself" ]
```

```
then
continue
fi
if [ "$var" = "quit" ]
then
break
fi
printf "%s\n" $var
done
```

The While loop

Syntax:

```
while [ condition ]
do
command1
command2
command3
..
....
Done
```

Loop is executed as long as given condition is true. For eg. Above for loop program can be written using while loop as

Example:

```
$cat > nt1
#!/bin/sh
#
#Script to test while statement
##
if [ $# -eq 0 ]
then
echo "Error - Number missing form command line argument"
echo "Syntax : $0 number"
echo " Use to print multiplication table for given number"
exit 1
fi

n=$1
i=1
while [ $i -le 10 ]
do
echo "$n * $i = `expr $i \* $n`"
i=`expr $i + 1`
```

```
done
Save it and try as
$ chmod +x nt1
$./nt1 7
```

From the above discussion note following points about loops
(a) First, the variable used in loop condition must be initialized; Next execution of the loop begins.
(b) A test (condition) is made at the beginning of each iteration.
(c) The body of loop ends with a statement that modifies the value of the test (condition) variable.

Conditional execution i.e. && and ||

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

An AND list has the
Syntax: command1 && command2
Here command2 is executed if, and only if, command1 returns an exit status of zero.

An OR list has the
Syntax: command1 || command2
Here command2 is executed if and only if command1 returns a non-zero exit status.

You can use both as follows
command1 && comamnd2 if exist status is zero || command3 if exit status is non-zero
Here if command1 is executed successfully then shell will run command2 and if command1 is not successful then command3 is executed.

Example:

$ rm myf && echo File is removed successfully || echo File is not removed
If file (myf) is removed successful (exist status is zero) then "echo File is removed successfully" statement is executed, otherwise "echo File is not removed" statement is executed (since exist status is non-zero)

Functions:

Functions enable you to break down the overall functionality of a script into smaller, logical subsections, which can then be called upon to perform their individual task when it is needed.

Creating Functions:

To declare a function, simply use the following syntax:

```
function_name () {
   list of commands
}
```

The name of your function is function_name, and that's what you will use to call it from elsewhere in your scripts. The function name must be followed by parentheses, which are followed by a list of commands enclosed within braces.

Example:

Following is the simple example of using function:

```
#!/bin/sh
# Define your function here
Hello () {
   echo "Hello World"
}
# Invoke your function
Hello
```

When you would execute above script it would produce following result:

```
$./test.sh
Hello World
$
```

Pass Parameters to a Function:

You can define a function which would accept parameters while calling those function. These parameters would be represented by $1, $2 and so on.

Following is an example where we pass two parameters Zara and Ali and then we capture and print these parameters in the function.

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

```
# Define your function here

Hello () {

   echo "Hello World $1 $2"

}

# Invoke your function

Hello Zara Ali
```

This would produce following result:

```
$./test.sh

Hello World [redacted]

$
```

Returning Values from Functions:

If you execute an **exit command** from **inside a function**, its effect is not only to **terminate** execution of the **function** but **also** of the **shell program** that called the function. If you instead want to just terminate execution of the function, then there is **way to come out of a defined function**. Based on the situation you can **return any value** from your function using the return command whose syntax is as follows:

```
return code
```

Here code can be anything you choose here, but obviously you should choose something that is meaningful or useful in the context of your script as a whole.

Example:

Following function returns a value 1:

```
#!/bin/sh

# Define your function here

Hello () {

   echo "Hello World $1 $2"

   return 10

}
```

```
# Invoke your function
Hello Zara Ali
# Capture value returned by last command
ret=$?
echo "Return value is $ret"
```

This would produce following result:

```
$./test.sh
Hello World Zara Ali
Return value is 10
$
```

Nested Functions:

One of the more interesting features of functions is that they can call themselves as well as call other functions. **A function that calls itself is known as a recursive function.**

Following simple example demonstrates a **nesting of two functions:**

```
#!/bin/sh
# Calling one function from another
number_one () {
   echo "This is the first function speaking..."
   number_two
}
number_two () {
   echo "This is now the second function speaking..."
}
# Calling function one.
number_one
```

This would produce following result:

```
This is the first function speaking...
This is now the second function speaking...
```

Special Parameters $* and $@:

There are special parameters that allow accessing all of the command-line arguments at once. $* and **$@** both will act the same unless they are enclosed in double quotes, **"".**

Both the parameter **specifies all command-line arguments** but the **"$*"** special parameter takes the **entire list as one argument with spaces between** and the **"$@"** special parameter takes the **entire list and separates it into separate arguments**.

We can write the shell script shown below to process an unknown number of command-line arguments with either the $* or $@ special parameters:

```
#!/bin/sh

for TOKEN in $*
do
  echo $TOKEN
done
```

There is one sample run for the above script:

```
$./test.sh Zara Ali 10 Years Old
Zara
Ali
10
Years
Old
```

Task 1:

Write Script, using function and case statement to perform basic math operation as follows + Addition, - Subtraction, x Multiplication, / Division.

Task2:

- **Write a shell program to generate Fibonacci series using loop.**
- **Sample Output:**

- Enter the no: 6
- Sample Output: 0 1 1 2 3 5