Chapter 15: Bash Shell

CEO Christine - Introduction & Learning Objectives

As the CEO for this project, it was my responsibility to introduce the team, the topics, as well as the learning objectives of Chapter 15 and Chapter 16.

The topic of Chapter 15 is the bash shell and basic scripting and the learning objectives are as follows:

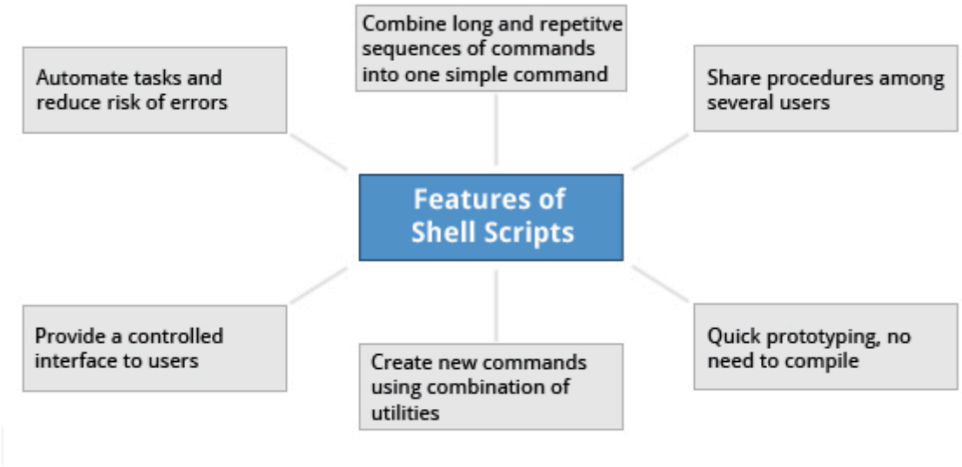
- Explain the features and capabilities of bash shell scripting.
- Know the basic syntax of scripting statements.
- Be familiar with various methods and constructs used.
- Test for properties and existence of files and other objects.
- Use conditional statements, such as if-then-else blocks.
- Perform arithmetic operations using scripting language.

I was also tasked with completing Lab 15.1 - the original code had an output of 0 and 2 to indicate a success or a failure respectively. But I switched it to print success and failure directly instead of printing the numbers. My Ubuntu was not working but it should print them successfully.

```
#!/bin/bash
#
# check for non-existent file, exit status will be “failure”.
#

if [ “$?” == 0 ]
then
        echo success
else
        echo failure
fi

ls SoMeFiLe.ext
echo “status: $?”

# create file, and do again, exit status will be “success”.
touch SoMeFiLe.ext
ls SoMeFiLe.ext
echo "status: $?"

# remove the file to clean up
rm SoMeFiLe.ext
```

PM Nadeen - Features & Capabilities

For this project, my role as PM was to first discuss the features and capabilities of bash shells. First off, the features of a shell script are pictured below:



As we can see, it is useful when we have commands that we repeat often, as it automates tasks which reduces errors. It is also quick since it does not need to compile, enables sharing procedures among several users, and many other benefits.

There are many typical shell interpreters that are used such as:

```
/bin/sh
/bin/bash
/bin/tcsh
/bin/csh
/bin/ksh
/bin/zsh
```

It is important to note that one of the main advantages of using scripts is that most information is saved in files, and so the use of a command interpreter becomes much easier and more efficient.

Below is an example of two identical commands, the first one is how we would write it with a script and the other without:

With :

```
#!/bin/bash
find . -name "*.c" -ls
```

Without:

```
find . -name "*.c" -ls
```

I typed the command into my terminal and this is what was returned:

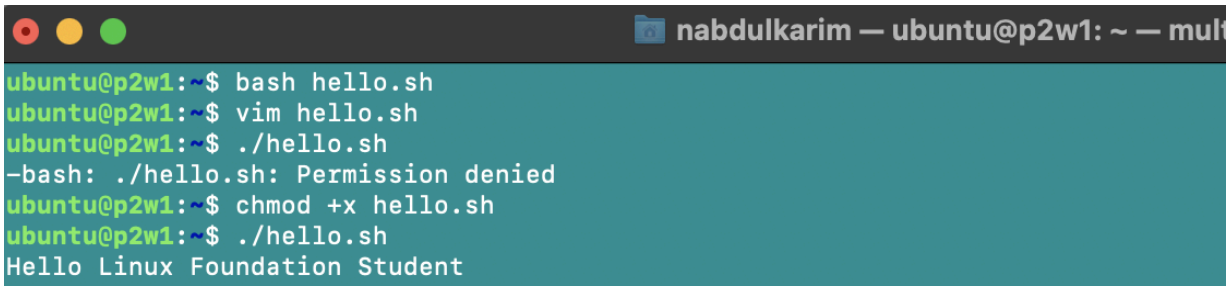
```
Last login: Thu Apr  8 22:11:56 on ttys002
(nabdulkarim@Nadeens-MacBook-Pro ~ % find . -name "*.c" -ls
8215263        8 -rw-rw-r--    1 nabdulkarim      staff             1771 Mar 13 15:14 ./p2-w1/gdb/valgrind_test.c
8223501       16 -rw-rw-r--    1 nabdulkarim      staff             5966 Mar 23 10:53 ./p2-w1/gdb/bugs.c
8215399       16 -rw-rw-r--    1 nabdulkarim      staff             5825 Mar 22 22:49 ./p2-w1/gdb/bug_start.c
8351285        8 -rw-rw-r--    1 nabdulkarim      staff             2184 Mar 26 23:17 ./p2-w1/omp/omp_hello_spin.c
8350376        8 -rw-rw-r--    1 nabdulkarim      staff             1319 Mar  7 09:54 ./p2-w1/omp/omp_dotprod_serial.c
8350396        8 -rw-rw-r--    1 nabdulkarim      staff              977 Mar  7 09:54 ./p2-w1/omp/omp_bug2.c
8350385        8 -rw-rw-r--    1 nabdulkarim      staff             1169 Mar  7 09:54 ./p2-w1/omp/omp_hello.c
8350379        8 -rw-rw-r--    1 nabdulkarim      staff              831 Mar  7 09:54 ./p2-w1/omp/omp_bug6.c
8350374        8 -rw-rw-r--    1 nabdulkarim      staff             2228 Mar  7 09:54 ./p2-w1/omp/omp_bug5fix.c
8350372        8 -rw-rw-r--    1 nabdulkarim      staff             1017 Mar  7 09:54 ./p2-w1/omp/omp_reduction.c
8350380        8 -rw-rw-r--    1 nabdulkarim      staff             1053 Mar  7 09:54 ./p2-w1/omp/omp_bug1.c
8350383        8 -rw-rw-r--    1 nabdulkarim      staff             2472 Mar  7 09:54 ./p2-w1/omp/omp_mm.c
8350400        8 -rw-rw-r--    1 nabdulkarim      staff             2062 Mar  7 09:54 ./p2-w1/omp/omp_bug5.c
8350377        8 -rw-rw-r--    1 nabdulkarim      staff             1408 Mar  7 09:54 ./p2-w1/omp/omp_getEnvInfo.c
8350398        8 -rw-rw-r--    1 nabdulkarim      staff             2725 Mar  7 09:54 ./p2-w1/omp/omp_dotprod_hybrid.c
8350397        8 -rw-rw-r--    1 nabdulkarim      staff             1557 Mar  7 09:54 ./p2-w1/omp/omp_workshare2.c
8350399        8 -rw-rw-r--    1 nabdulkarim      staff             1094 Mar  7 09:54 ./p2-w1/omp/omp_bug4.c
8350393        8 -rw-rw-r--    1 nabdulkarim      staff             1908 Mar  7 09:54 ./p2-w1/omp/omp_dotprod_openmp.c
8350375        8 -rw-rw-r--    1 nabdulkarim      staff             1215 Mar  7 09:54 ./p2-w1/omp/omp_bug1fix.c
8350392        8 -rw-rw-r--    1 nabdulkarim      staff             1901 Mar  7 09:54 ./p2-w1/omp/omp_dotprod_mpi.c
8350386        8 -rw-rw-r--    1 nabdulkarim      staff             1251 Mar  7 09:54 ./p2-w1/omp/omp_workshare1.c
8350387        8 -rw-rw-r--    1 nabdulkarim      staff             1819 Mar  7 09:54 ./p2-w1/omp/omp_hello_sleep.c
8350378        8 -rw-rw-r--    1 nabdulkarim      staff              971 Mar  7 09:54 ./p2-w1/omp/omp_orphan.c
8350395        8 -rw-rw-r--    1 nabdulkarim      staff             1891 Mar  7 09:54 ./p2-w1/omp/omp_bug3.c
```

Another thing we could use bash for, is creating a file, putting in information and printing it out. When using bash, it is important to make files executable before trying to run them. This is done using the command: **`chmod +x hello.sh`**

Below is again, two identical commands one which uses scripting and one which does not and what my terminal looked like when running it:

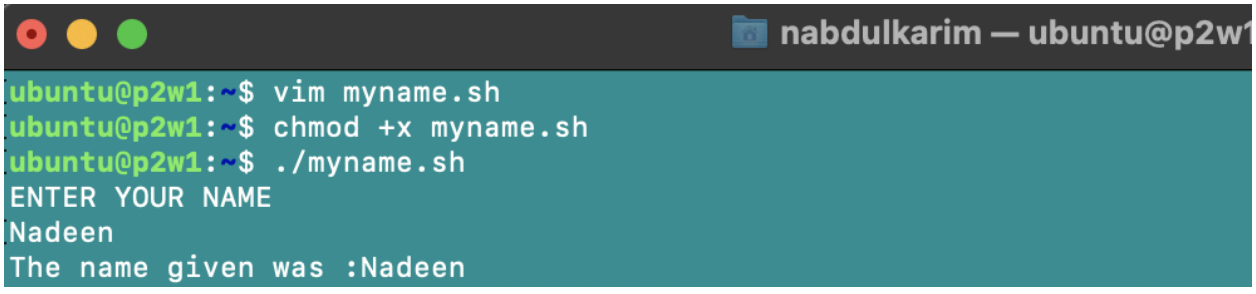
```
$ cat > hello.sh

#!/bin/bash

echo "Hello Linux Foundation Student"


chmod +x hello.sh
```

```
/hello.sh

OR

$ bash hello.sh
Hello Linux Foundation Student
```



We can also have more interactive examples by prompting users to put in information, below is the code we would use and how my terminal looked it running it:

```
#!/bin/bash

# Interactive reading of a variable

echo "ENTER YOUR NAME"

read name

# Display variable input

echo The name given was :$name
```



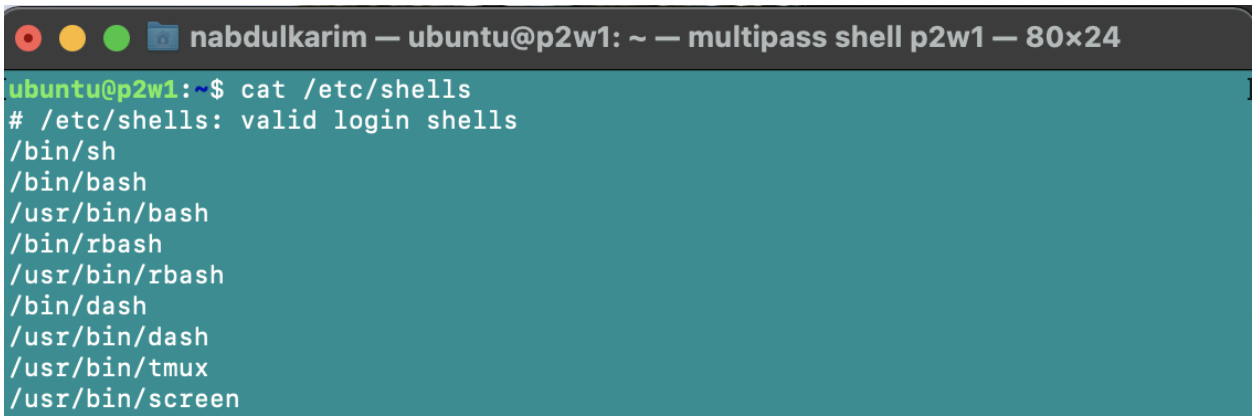
All shell scripts generate a return value upon finishing execution, which can be explicitly set with the exit statement. Return values permit a process to monitor the exit state of another process, often in a parent-child relationship. Knowing

how the process terminates enables taking any appropriate steps which are necessary or contingent on success or failure.

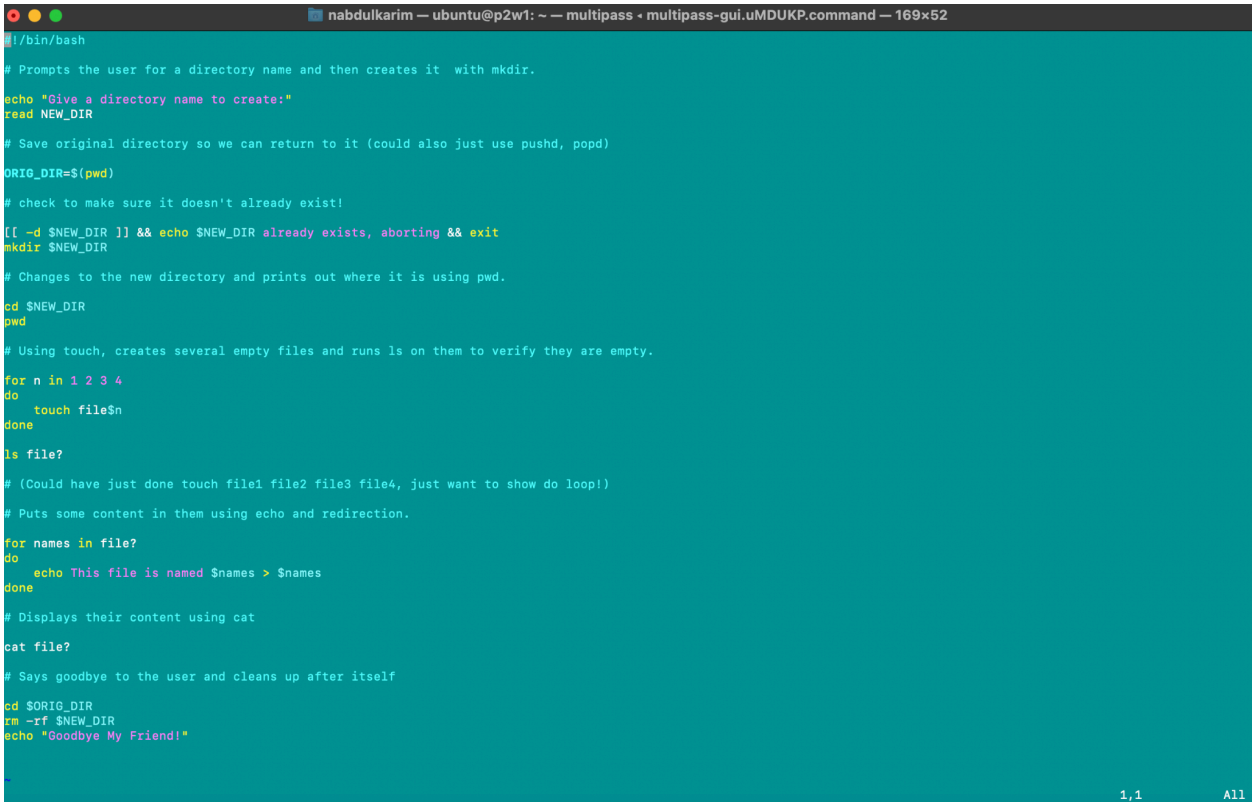
```
$ ls /etc/logrotate.conf

/etc/logrotate.conf

$ echo $?

0
```

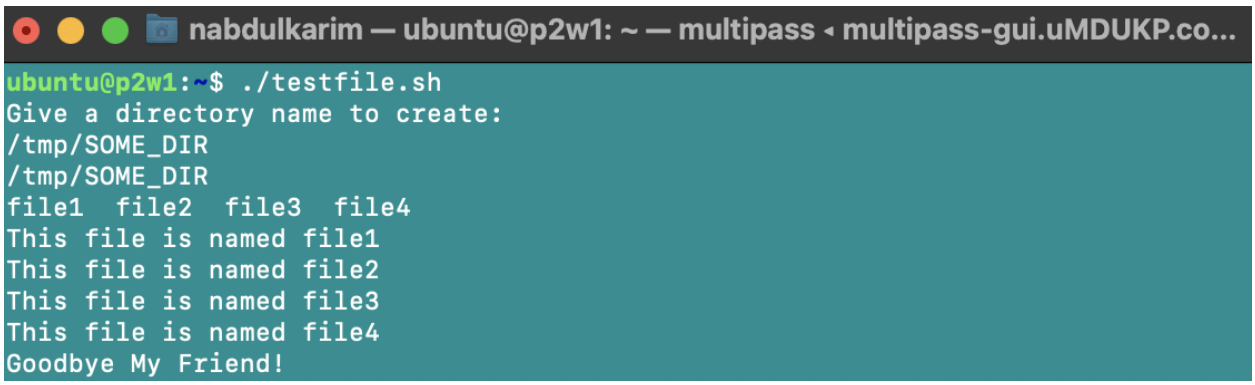
The return value being 0, means our code is running successfully.

One thing I learned from edX as well, is to look up the shells on my system. By running the code cat /etc/shells, I was able to print out the shells as shown below.

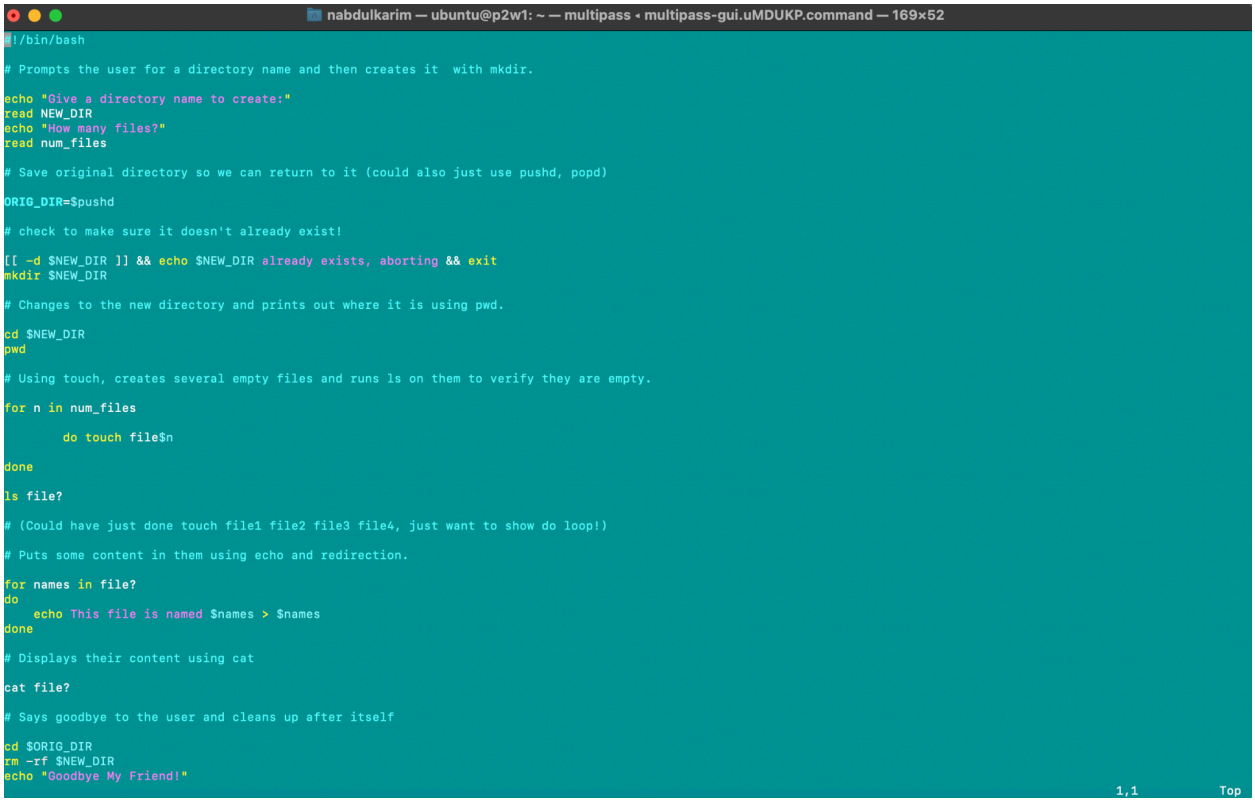


I was also responsible for changing lab15.2. Before changing the code, the lab prompted the user to enter a directory name they would like to create, check if the directory already exists, create several empty files, put some content in there and then return the files. To be more specific, it returned exactly 4 files. This is how the code looked like before, and how it returned.





After changing the code, I made it so it asks the user how many files they would like to create, and used the command PUSHD instead of pwd at the top to save the original directory. This is how my code looks and what it returns.



Dev0 Alfie - Syntax

My first role as Dev0 was to discuss the syntax of bash shells. First of which was the special characters used in scripts. Which are the following:

“#” Which is used to add a comment, except when used as \#, or as #! when starting a script.
“\” Which is used at the end of a line to indicate continuation on to the next line.
“;” Which is used to interpret what follows as a new command to be executed next.
“$” Which indicates what follows is an environment variable.
“>” which redirects outputs.
“>>” Which appends outputs.
“<” Which redirects inputs and lastly

“|” which is used to pipe the result into the next command

To provide examples:

- The “;” character is used to separate commands and execute them sequentially, as if they had been typed on separate lines. All commands will run regardless of the result of the others. Ex: $ make ; make install ; make clean
- The && (and) operator will abort subsequent commands when an earlier one fails. Ex: $ make && make install && make clean
- The || (or) operator will proceed running the commands until one of them succeeds. Ex: $ cat file1 || cat file2 || cat file3

Next are functions which must first be defined and are then executable in scripts which is done like so:

```
function_name () {
   command...
}
```

I was also responsible for completing lab 15.3 and 15.4 which are shown below.

Lab 15.3 code and execution:

```
#!/bin/bash
#
# check for an argument, print a usage message if not supplied.
#
if [ $# -eq 0 ] ; then
      echo "Usage: $0 argument"
      exit 1
fi
echo $1
exit 0
```

```
ubuntu@primary:~/Home/Desktop/COMP310$ ./test.sh
Usage: ./test.sh argument
ubuntu@primary:~/Home/Desktop/COMP310$ ./test.sh test
test
ubuntu@primary:~/Home/Desktop/COMP310$
```

Lab 15.4 code and execution:

```
#!/bin/bash

echo "Enter 1 or 2, to set the environmental variable EVAR to Yes or No"
read ans

# Set up a return code
RC=0

if [ $ans -eq 1 ]
then
    export EVAR="Yes"
else
    if [ $ans -eq 2 ]
    then
        export EVAR="No"
    else
# can only reach here with a bad answer
        export EVAR="Unknown"
        RC=1
    fi
fi
echo "The value of EVAR is: $EVAR"
exit $RC
```

```
ubuntu@primary:~/Home/Desktop/COMP310$ ./test2.sh
Enter 1 or 2, to set the environmental variable EVAR to Yes or No
1
The value of EVAR is: Yes
ubuntu@primary:~/Home/Desktop/COMP310$ ./test2.sh
Enter 1 or 2, to set the environmental variable EVAR to Yes or No
2
The value of EVAR is: No
ubuntu@primary:~/Home/Desktop/COMP310$ ./test2.sh
Enter 1 or 2, to set the environmental variable EVAR to Yes or No
5
The value of EVAR is: Unknown
ubuntu@primary:~/Home/Desktop/COMP310$ ./test2.sh
Enter 1 or 2, to set the environmental variable EVAR to Yes or No
10
The value of EVAR is: Unknown
ubuntu@primary:~/Home/Desktop/COMP310$
```

Dev1 Ben - Constructs

Lab 15.5: Working with Functions:

The original task of this lab was to write a script which:

1. Asks the user for a number (1, 2 or 3).
2. Calls a function with that number in its name. The function should display a message with its name included.

I modified it so that instead of asking the user to input a number they would have to input a color (red, blue or yellow) and the script would return a message which included the name of the color inputed. I accomplished this by simply changing the function names and the output messages to include colors instead of numbers like so:

```
#!/bin/bash

# Functions (must be defined before use)
funcRed() {
echo " This message is from function Red"
}
funcBlue() {
echo " This message is from function Blue"
}
funcYellow() {
echo " This message is from function Yellow"
}

# Beginning of the main script

# prompt the user to get their choice
echo "Enter a color Red, Blue, or Yellow"
read n

# Call the chosen function
func$n
```

```
ubuntu@primary:~/Home/Documents/University of San Diego/Spring 2021/COMP 310/Project 3$ ./testfun.sh
Enter a color Red, Blue, or Yellow
Red
 This message is from function Red
ubuntu@primary:~/Home/Documents/University of San Diego/Spring 2021/COMP 310/Project 3$ ./testfun.sh
Enter a color Red, Blue, or Yellow
Blue
 This message is from function Blue
ubuntu@primary:~/Home/Documents/University of San Diego/Spring 2021/COMP 310/Project 3$ ./testfun.sh
Enter a color Red, Blue, or Yellow
Yellow
 This message is from function Yellow
ubuntu@primary:~/Home/Documents/University of San Diego/Spring 2021/COMP 310/Project 3$ ./testfun.sh
Enter a color Red, Blue, or Yellow
Green
./testfun.sh: line 21: funcGreen: command not found
ubuntu@primary:~/Home/Documents/University of San Diego/Spring 2021/COMP 310/Project 3$
```

The if Statement:

Conditional decision making, using an **if** statement, is a basic construct that any useful programming or scripting language must have. When an **if** statement is used, the ensuing actions depend on the evaluation of specified conditions, such as:

- Numerical or string comparisons
- Return value of a command (0 for success)
- File existence or permissions.

Example:

```
if [ -f "$1" ]
then
    echo file "$1 exists"
else
    echo file "$1" does not exist
fi
```

```
ubuntu@primary:~/Home$ vi example
ubuntu@primary:~/Home$ sh ex.sh example
file example exists
ubuntu@primary:~/Home$
```

*Notice the use of the square brackets (**[]**) to delineate the test condition. There are many other kinds of tests you can perform, such as checking whether two numbers are equal to, greater than, or less than each other and make a decision accordingly.

The elif Statement:

You can use the **elif** statement to perform more complicated tests, and take action appropriate actions.

Example:

```
echo Give your name

read name

if [ "$name" == Jimmy ] ; then
        echo Hello $name
elif [ "$name" == Robert ] ; then
        echo Hello $name
elif [ "$name" == John ] ; then
        echo Hello $name
else
        echo Forget it $name, You are not on Led Zeppelin
fi
```

```
ubuntu@primary:~/Home$ bash ./ex.sh
Give your name
Ben
Forget it Ben, You are not in Led Zeppelin
ubuntu@primary:~/Home$
```

Testing for Files:

bash provides a set of file conditionals, that can be used with the **if** statement, including those in the table.You can use the **if** statement to test for file attributes, such as:

- File or directory existence
- Read or write permission
- Executable permission.

Full list of conditions:

Condition	Meaning
`-e file`	Checks if the file exists.
`-d file`	Checks if the file is a directory.
`-f file`	Checks if the file is a regular file (i.e. not a symbolic link, device node, directory, etc.)
`-s file`	Checks if the file is of non-zero size.
`-g file`	Checks if the file has **`sgid`** set.
`-u file`	Checks if the file has **`suid`** set.
`-r file`	Checks if the file is readable.
`-w file`	Checks if the file is writable.
`-x file`	Checks if the file is executable.

Example:

```
if [ -e "$1" ] && [ -r "$1" ] && [ -w "$1" ]
then
        echo file "$1 exits, is readable and writable"
else
        echo file "$1" does not exist
fi
```

```
ubuntu@primary:~/Home$ bash ./ex.sh example
file example exits, is readable and writable
ubuntu@primary:~/Home$
```

Boolean Expressions:

Boolean expressions evaluate to either TRUE or FALSE, and results are obtained using the various Boolean operators listed in the table:

Operator	Operation	Meaning
&&	**AND**	The action will be performed only if both the conditions evaluate to true.
||	**OR**	The action will be performed if any one of the conditions evaluate to true.
!	**NOT**	The action will be performed only if the condition evaluates to false.

We can even use such expressions when working with multiple data types, including strings or numbers, as well as with files.

Testing Strings:

You can use the **if** statement to compare strings using the operator **==** (two equal signs).

Example:

```
echo "Please Specify Window, Middle, or Aisle for your seat"
read CHOICE
if [ "$CHOICE" == Window ] ; then
        echo "you have a Window Seat, 29A"
elif [ "$CHOICE" == Middle ] ; then
        echo "you have a Middle Seat, 29B"
elif [ "$CHOICE" == Aisle ] ; then
        echo "you have a Aisle Seat, 29C"
else
        echo $CHOICE is not a valid answer
        echo Please try again
fi
```

```
ubuntu@primary:~/Home$ ./ex.sh
Please Specify Window, Middle, or Aisle for your seat
Middle
you have a Middle Seat, 29B
ubuntu@primary:~/Home$ ./ex.sh
Please Specify Window, Middle, or Aisle for your seat
Window
you have a Window Seat, 29A
ubuntu@primary:~/Home$ ./ex.sh
Please Specify Window, Middle, or Aisle for your seat
OnTheWing
OnTheWing is not a valid answer
Please try again
ubuntu@primary:~/Home$
```

Numerical Tests:

You can use specially defined operators with the **if** statement to compare numbers. The various operators that are available are listed in the table:

Operator	Meaning
`-eq`	Equal to
`-ne`	Not equal to
`-gt`	Greater than
`-lt`	Less than
`-ge`	Greater than or equal to
`-le`	Less than or equal to

Example:

```
ubuntu@primary:~/Home$ ./ex.sh 33
You are in your 30s
ubuntu@primary:~/Home$ ./ex.sh 21
You are in your 20s
ubuntu@primary:~/Home$ ./ex.sh 18
at AGE=18, you are not in the proper age of 21-50
ubuntu@primary:~/Home$
```

```
AGE=$1
if [[ $AGE -ge 20 ]] && [[ $AGE -lt 30 ]]; then
        echo "You are in your 20s"
elif [[ $AGE -ge 30 ]] && [[ $AGE -lt 40 ]]; then
        echo "You are in your 30s"
elif [[ $AGE -ge 40 ]] && [[ $AGE -lt 50 ]]; then
        echo "You are in your 40s"
else
        echo at AGE=$AGE, you are not in the proper age of 21-50
fi
```

Arithmetic Expressions:

Arithmetic expressions can be evaluated in the following three ways (spaces are important!):

- Using the **expr** utility
  expr is a standard but somewhat deprecated program. The syntax is as follows:
  expr 8 + 8
  echo $(expr 8 + 8)
- Using the **$((...))** syntax
  This is the built-in shell format. The syntax is as follows:
  echo $((x+1))
- Using the built-in shell command **let**. The syntax is as follows:
  let x=(1 + 2); echo $x

* In modern shell scripts, the use of **expr** is better replaced with **var=$((...))**.

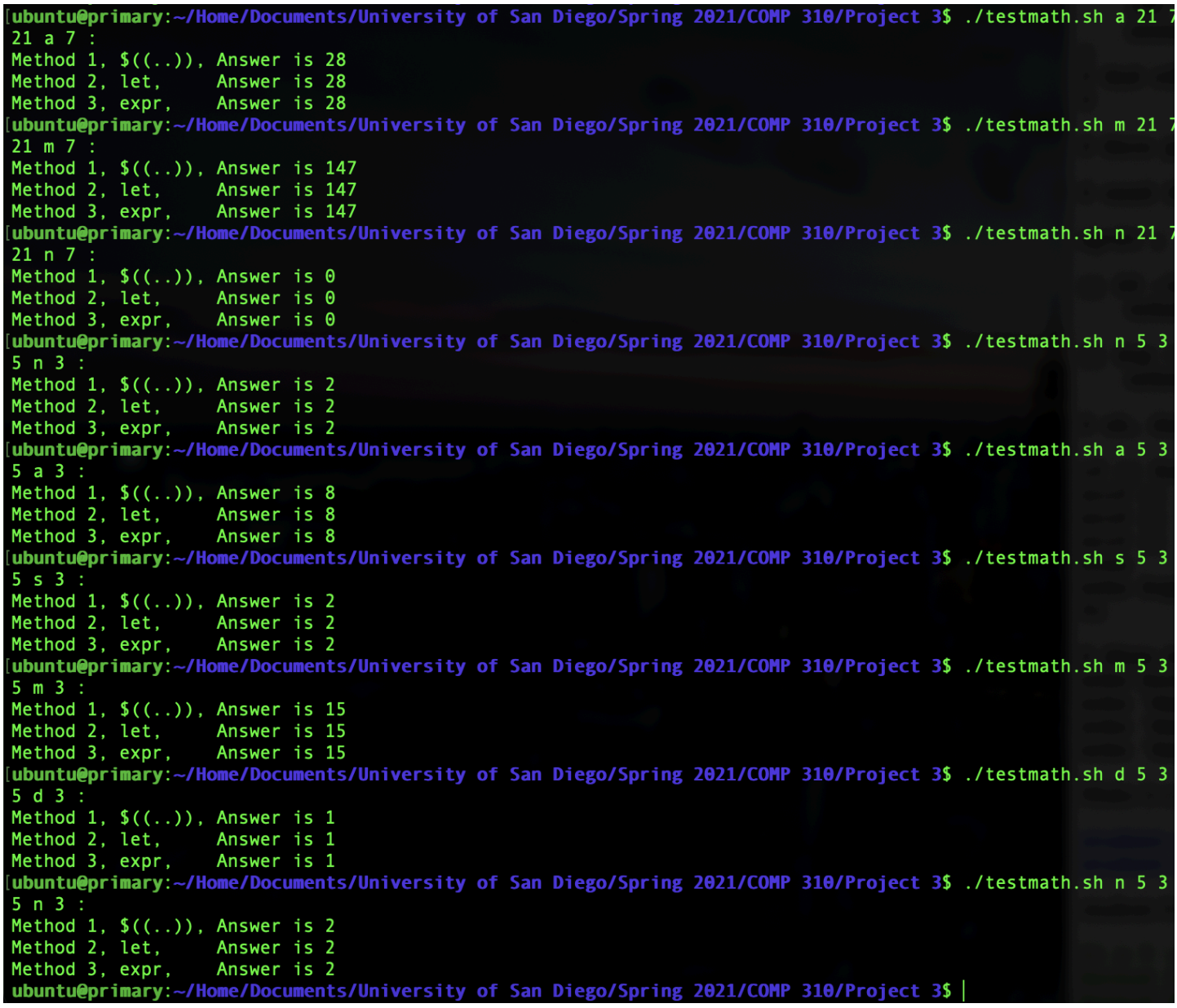
Lab 15.6: Arithmetic Functions:

The original task of this lab was to write a script that will act as a simple calculator for add, subtract, multiply and divide.

1. Each operation should have its own function.
2. Any of the three methods for bash arithmetic, (**$((..))**, **let** , or **expr**) may be used.
3. The user should give 3 arguments when executing the script:
   The first should be one of the letters **a**, **s**, **m**, or **d** to specify which math operation.
   The second and third arguments should be the numbers that are being operated on.
4. The script should detect for bad or missing input values and display the results when done.

I modified it to also include the mod (%) function as well. I did this by adding a function mod() with relatively the same cose as the other functions. The only difference is I used “%” instead of another operand. I also made it so that users would have to type in “n” to execute a mod operation between two numbers. The code is below:

```
add() {
    answer1=$(($1 + $2))
    let answer2=($1 + $2)
    answer3=`expr $1 + $2`
}
sub() {
    answer1=$(($1 - $2))
    let answer2=($1 - $2)
    answer3=`expr $1 - $2`
}
mult() {
    answer1=$(($1 * $2))
    let answer2=($1 * $2)
    answer3=`expr $1 \* $2`
}
div() {
    answer1=$(($1 / $2))
    let answer2=($1 / $2)
    answer3=`expr $1 / $2`
}
mod() {
    answer1=$(($1 % $2))
    let answer2=($1 % $2)
    answer3=`expr $1 % $2`
}
# End of functions
#

# Main part of the script

# need 3 arguments, and parse to make sure they are valid types

op=$1 ; arg1=$2 ; arg2=$3

[[ $# -lt 3 ]] && \
    echo "Usage: Provide an operation (a,s,m,d,n) and two numbers" && exit 1

[[ $op != a ]] && [[ $op != s ]] && [[ $op != d ]] && [[ $op != m ]] && [[ $op != n ]] && \
    echo operator must be a, s, m, d, or n, not $op as supplied

# ok, do the work!

if [[ $op == a ]] ; then add $arg1 $arg2
elif [[ $op == s ]] ; then sub $arg1 $arg2
elif [[ $op == m ]] ; then mult $arg1 $arg2
elif [[ $op == d ]] ; then div $arg1 $arg2
elif [[ $op == n ]] ; then mod $arg1 $arg2
else
echo $op is not a, s, m, d, or n, aborting ; exit 2
fi
```



Chapter 16: Bash Shell Scripting

CEO Christine - Introduction & Learning Objectives

The topic of Chapter 16 was bash shell scripting:

- Manipulate strings to perform actions such as comparison and sorting.
- Use Boolean expressions when working with multiple data types, including strings or numbers, as well as files.

- Use **case** statements to handle command line options.
- Use looping constructs to execute one or more lines of code repetitively.
- Debug scripts using set **-x** and set **+x**.
- Create temporary files and directories.
- Create and use random numbers.

PM Nadeen - String Manipulation

For this part of the project, I was responsible for discussing string manipulation. Strings are a sequence of characters which could be:

- Numbers
- Letters
- Symbols
- Punctuation

Using scripts, we can perform many operations on strings:

Operator	Meaning
`[[ string1 > string2 ]]`	Compares the sorting order of **`string1`** and **`string2`**.
`[[ string1 == string2 ]]`	Compares the characters in **`string1`** with the characters in **`string2`**.
`myLen1=${#string1}`	Saves the length of **`string1`** in the variable **`myLen1`**.

In my terminal, I implemented some of the string operations listed above by writing code which prompts the user to enter two strings which will then be compared to check if identical. Below is how my program was running:

nabdulkarim — ubuntu@p2w1: ~ — multipass ◂ multipass-gui.uMDUKP.comma...

```
#!/bin/bash
echo Give two strings to compare:
echo ""

read str1 str2
echo""

if [ "$str1" = "$str2" ] ; then
        echo "The first string: $str1, is the same as the second string: $str2"
else
        echo "The first string: $str1, is not the same as the second string: $str2"

fi
```

nabdulkarim — ubuntu@p2w1: ~ — multipass ◂ multipass-gui.uMDUKP.comma

```
ubuntu@p2w1:~$ ./stringmanipulation.sh
Give two strings to compare:

Nadeen Abdulkarim

The first string: Nadeen, is not the same as the second string: Abdulkarim
ubuntu@p2w1:~$ ./stringmanipulation.sh
Give two strings to compare:

Nadeen Nadeen

The first string: Nadeen, is the same as the second string: Nadeen
ubuntu@p2w1:~$ ./stringmanipulation.sh
Give two strings to compare:

Nadeen nadeen

The first string: Nadeen, is not the same as the second string: nadeen
```

In order to compare only certain parts of a string, we would use the command:

`${string:0:n}`

`0: offset (the character we begin with)`

`n: where we want to end the comparison (the last character we want)`

To extract the characters after a “.” we would use the command:

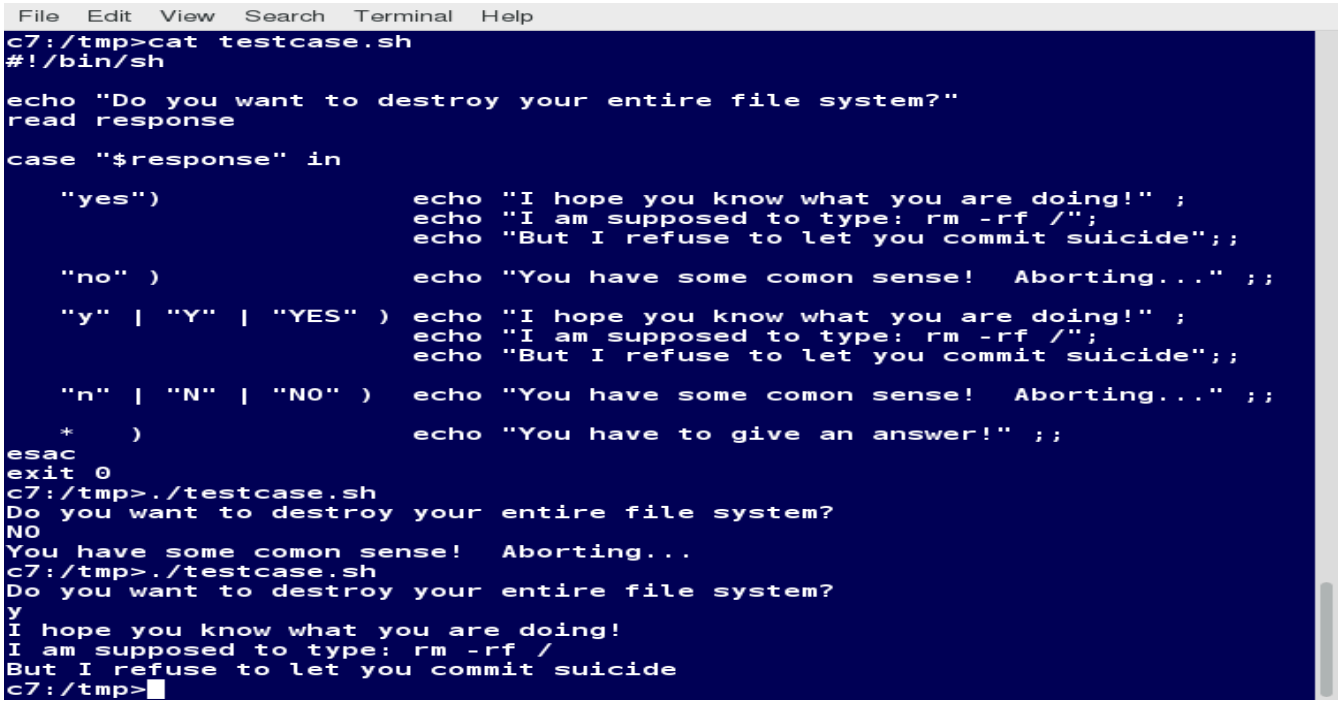
```
${string#*.}
```

Dev0 Alfie - The case Statement

My last responsibility as Dev0 was to discuss the case statement which is quite simple and follows this structure:

case expression in
  pattern1) execute commands;;
  pattern2) execute commands;;
  pattern3) execute commands;;
  pattern4) execute commands;;
  *) execute some default commands or nothing ;;
esac

And the following picture provides an example:



Dev1 Ben - Looping Constructs

By using looping constructs, you can execute one or more lines of code repetitively, usually on a selection of values of data such as individual files. Usually, you do this until a conditional test returns either true or false, as is required.

Three type of loops are often used in most programming languages:

- **for**
- **while**
- **until**

The for Loop:

The **for** loop operates on each element of a list of items.

Example:

```
sum=0

for j in 1 2 3 4 5 6 7 8 9 10
do
        sum=$(( $sum + $j ))
done
echo The sum is: $sum
echo The sum of numbers from 1 to n is: 'n*(n+1)/2'
echo Check Value = $(( ($j*($j+1))/2 ))
exit 0
```

```
ubuntu@primary:~/Home$ bash ./ex.sh
The sum is: 55
The sum of numbers from 1 to n is: n*(n+1)/2
Check Value = 55
ubuntu@primary:~/Home$
```

The while Loop:

The **while** loop repeats a set of statements as long as the control command returns true.

Example:

```
n=$1
[ "$n" == "" ] && echo please give a number and try again && exit

factorial=1 ; j=1

while [ $j -le $n ]
do
        factorial=$(( $factorial * $j ))
        j=$(($j+1))
done
echo The factorial of $n, "$n"'!' = $factorial
exit 0
```

```
ubuntu@primary:~/Home$ bash ./ex.sh 6
The factorial of 6, 6! = 720
ubuntu@primary:~/Home$
```

*The set of commands that need to be repeated should be enclosed between **do** and **done**. You can use any command or operator as the condition. Often, it is enclosed within square brackets (**[]**).

The until Loop:

The **until** loop repeats a set of statements as long as the control command is false (opposite of the **while** loop).

Example:

```
n=$1
[ "$n" == "" ] && echo please give a number and try again && exit

factorial=1 ; j=1

until [ $j -gt $n ]
do
        factorial=$(( $factorial * $j ))
        j=$(($j+1))
done
echo The factorial of $n, "$n"'!' = $factorial
exit 0
```

```
ubuntu@primary:~/Home$ bash ./ex.sh 6
The factorial of 6, 6! = 720
ubuntu@primary:~/Home$
```

*Similar to the **while** loop, the set of commands that need to be repeated should be enclosed between **do** and **done**. You can use any command or operator as the condition.

Tester Tyler - Script Debugging & Some Additional Useful Techniques

First, I was responsible for covering the script debugging and additional techniques sections of Chapter 16. First, script debugging is an important topic to learn because of the time it saves to locate and solve bugs in script code. To run debugging on an entire script, use the -x tag while running the script. Below is an example of a script and the command to debug the script.

```
sum=0
for i in 1 2 3 4
do
        sum=(($sum+$i))
done
echo "The sum of "$i" numbers is $sum"
```

```
ty@DESKTOP-3UTLJ0L:~$ bash -x ex.sh
+ sum=0
ex.sh: line 4: syntax error near unexpected token `('
ex.sh: line 4: `        sum=(($sum+$i))'
ex.sh: line 5: syntax error near unexpected token `done'
ex.sh: line 5: `done'
ty@DESKTOP-3UTLJ0L:~$
```

We can see from the + sum=0 line in the output that the command sum=0 was successfully run, but we get an error shortly after. The output says that we have a problem with our syntax on line 4.

Next, if we want to run debugging on certain lines of a script, we can use the set -x and set +x commands surrounding our code to turn on and off the debugger, respectively. A script is shown below with an example of running debugging on only the sum=0 command.

```
set -x
sum=0
set +x
for i in 1 2 3 4
do
        sum=$((sum+i))
done
echo "The sum of "$i" numbers is $sum"
```

```
ty@DESKTOP-3UTLJ0L:~$ bash ex.sh
+ sum=0
+ set +x
The sum of 4 numbers is 10
ty@DESKTOP-3UTLJ0L:~$
```

Finally, we can store error information in a file using the below command.

```
ty@DESKTOP-3UTLJ0L:~$ bash ex.sh 2> error.log
ty@DESKTOP-3UTLJ0L:~$ cat error.log
ex.sh: line 4: syntax error near unexpected token `('
ex.sh: line 4: `        sum=(($sum+$i))'
ex.sh: line 5: syntax error near unexpected token `done'
ex.sh: line 5: `done'
ty@DESKTOP-3UTLJ0L:~$
```

Here, the command runs the ex.sh script and feeds the stderr output, signified by the number 2, into a file named error.log. Because we use the > symbol to direct the stderr output to the file, we overwrite the existing data in the error.log file if there is any. Instead, we can use >> to append data to error.log. We can also use any other file name than error.log to store error information in this way.

Next, we talk about the additional techniques covered in Chapter 16. First, we cover the random environment variable. This variable is important to learn because it can be used for security

purposes, like scrambling patterns in data or overwriting important data so no one else can use it. Also generating random data can be useful for testing different programs.

```
ty@DESKTOP-3UTLJ0L:~$ echo $RANDOM
14937
ty@DESKTOP-3UTLJ0L:~$ echo $RANDOM
21450
ty@DESKTOP-3UTLJ0L:~$
```

We can access the random environment variable value by typing $RANDOM. Below is an example of using random variables, influenced by Lab 16.3. We take in an integer provided by a user and then multiply it by a random integer. We then print the random integer and the correct product.

```
randVal=$RANDOM
input=$1
num=$((input*randVal))
echo "random number is : $randVal"
echo "multiplied number is : $num"
exit 0
```

```
ty@DESKTOP-3UTLJ0L:~$ bash testrandom.sh 1000
random number is : 23385
multiplied number is : 23385000
ty@DESKTOP-3UTLJ0L:~$
```

Another useful topic regards tempfiles, or temporary files that can be use to temporarily store data. You can make temporary files and temporary directories to store certain information for a short time. You can use the commands below to creat temporary files and directories.

Command	Usage
`TEMP=$(mktemp /tmp/tempfile.XXXXXXXX)`	To create a temporary file
`TEMPDIR=$(mktemp -d /tmp/tempdir.XXXXXXXX)`	To create a temporary directory

The mktemp command replaces the xxxxxxxx with random characters to make the name of the temporary files hard to predict for hackers.

We can also discard output with a device node, a special file used for discarding data. This is important when we have commands that output a lot of data, which can overwhelm the console. Below is the command to send standard output and error to the device node.

```
$ ls -lR /tmp > /dev/null
```

Lastly, I was responsible for two labs from different sections, one focusing on parts of strings and one focusing on the case statement. First, for Lab 16.2 the original problem asked to take in two strings from the user and compare them in different ways. My spin on this lab was to take two strings/file names from the user and print whether they are of the same file type. This uses the parts of a string after a . symbol. Below is my coded solution to this lab along with a test run.

```
first=$1
second=$2

type1=${first#*.}
type2=${second#*.}

if [[ $type1 == $type2 ]] ; then
        echo "both files are of type "$type1""
else
        echo "files are of different types"
fi
```

```
ty@DESKTOP-3UTLJ0L:~$ bash strings.sh t.jpg j.jpg
both files are of type jpg
ty@DESKTOP-3UTLJ0L:~$ bash strings.sh t.txt j.jpg
files are of different types
```

As you can see when the input files are of the same type we print out “both files are of type <filetype>” and when the input file types are different we print that as well. So, the script prints the correct output using the ${name#*.} command to get parts of a string after the “.” symbol.

Next, I was also responsible for Lab 16.2, which originally asked to make a case statement that takes in an integer between 1 and 12 provided by the user and returns the corresponding month. My spin on this lab was to take a user integer between 0 and 10 and print the corresponding binary number. The code and test runs are shown below.

```
num=$1
case $num in
        0) echo "0000";;
        1) echo "0001";;
        2) echo "0010";;
        3) echo "0011";;
        4) echo "0100";;
        5) echo "0101";;
        6) echo "0110";;
        7) echo "0111";;
        8) echo "1000";;
        9) echo "1001";;
        10) echo "1010";;
        *)
        echo "Error. No number matches: $num"
        echo "Please pass a number between 0 and 10."
        exit 2
        ;;
esac
exit 0
```

```
ty@DESKTOP-3UTLJ0L:~$ bash case.sh 9
1001
ty@DESKTOP-3UTLJ0L:~$
```

Clearly, when we input a number between 0 and 10, we get the correct value. When we enter a number out of this range we notify the user to do so.