Lab 4 - Text Editors and Bash Scripting

Introduction to Command Line Text Editors

In Linux, we have multiple commands that act as the `notepad` of linux. Such softwares include (but are not limited to):

- nano
- vim
- emacs

In this course, we'll be studying nano.

Nano

Nano is a command line text editor that we can use to write, read and delete data from within a file.

In order to open a file in nano, we type the following command:

```
nano <file-name>
```

Now, in order to save data into the file, we will firstly press `CTRL+S` and then, in order to exit, we need to press `CTRL+X`.

⇒ Shortcuts for Nano:

```
CTRL + S => Save
CTRL + W => Search
CTRL + X => Exit
CTRL + K => CUT Entire Line
CTRL + U => Paste line from Buffer
```

Writing your first Bash Script:

The first line in a bash script must be `#!/bin/bash` and is called as `SHEBANG` line.

> A she-bang is set of sequence that begins with #! and then the interpreter is specified. In our case, we'll be using /bin/bash as the interpreter.

Then, we will use `echo` command in order to print data into the stdout.

```
#!/bin/bash

echo "Hello World"
```

Now, in order to execute this file, we need to give it executable permissions. We can do that by using the `chmod` command.

```
chmod +x <file-name>
```

Now, we can execute the file by using the following command:

`./<file-name>`

> **Quick Task: Write a simple Bash Script that will Print out your roll number and Name.**
> **Example:**
> **231526 - Ahmed**

Variables

Variables are used to store data in a program. In bash, we can declare a variable by using the following syntax:

```
# NOTE: There should be no space between the variable name and the equal sign
variable_name=value
```

In order to display a variable's value, we can use `echo` command. For expansion, we always prefix a variable with a `$` sign.

```
echo $variable_name
```

Expansions:

Variable expansion in bash scripting is done using the `$` sign followed by the variable name. This allows us to access the value stored in the variable.
For example, if we have a variable named `name` with the value "John", we can expand it using `$name` to get the value "John" in our script.
Command substitution is done using the backtick character (`) or the `$()` syntax. This allows us to execute a command and use its output as part of our script.
For example, if we want to store the current date in a variable named `current_date`, we can use the following command substitution:

```
current_date=`date`
```

Now, the value of `current_date` will be set to the current date and time.

We can also use command substitution within a string by enclosing the command in `$()`.

For example, if we want to print the output of the `ls` command within a string, we can use the following:

```
echo "The files in the current directory are: $(ls)"
```

> For variable expansion, we **MUST** use `"` as `'` quotes won't allow for variable expansion.

> **NOTE:** `!`, `&`, `|`, and `$` are special symbols in Bash and may not produce desired output when placed inside `"`.

Variable Types

There are two types of variables in bash:

- System Variables
- User Defined Variables

System Variables

System variables are the variables that are defined by the system and are used to store system related information.

Some of the system variables are:

`$HOME`: Stores the path to the home directory of the user

`$SHELL`: Stores the name of the shell currently being utilized

`$PWD`: Stores the path to the current working directory

`$BASH`: Stores the path to the bash shell

`$BASH_VERSION`: Stores the version of the bash shell

`$LOGNAME`: Stores the name of the user

> These system variables are sometimes also referred to as Environment variables.

User Defined Variables

User defined variables are the variables that are defined by the user and are used to store user related information.

Example:

Any variable that we create to hold a certain value.

Read Input from User

In order to read input from the user, we can use the `read` command.

For the `read` command, we need to specify the variable name after the command:

```
read <variable-name>
```

This will halt the screen and wait for user to input data. (Similar to `cin` in C++).

Now, if we want to print a message before asking for input, we can specify it `-p` - which stands for prompt.

```
read -p "MSG" <variable-name>
```

```
## Example: Asking the user for their name:
read -p "Enter your name: " name
```

Now, the value that the user enters will be stored in the variable.

Unsetting a variable

In order to unset a variable, we can use the `unset` command.

```
unset <variable-name>
```

IF-ELSE in Bash

In order to use if-else statements in bash, we can use the following syntax:

```
if [[ <condition> ]]
then
    <statements>
else
    <statements>
fi
```

Comparison Operators

There are multiple operators in Bash, similar to the ones in other languages, such as C++. One unique thing in bash is, different types of operators are used to compare different datatypes.

If we want to compare two numbers (one variable with a number), we will use the following convention:

```
if [[ $var <op> num ]];

# Valid <op> include:

## -lt => [<]  Less Than
## -le => [<=] Less Than or equal to
## -ge => [>=] Greater Than or equal to
## -gt => [>]  Greater Than
## -eq => [==] Equal to
## -ne => [!=] Not Equal to

## Example usage:
var=5
if [[ $var -ge 5 ]]; then
```

```
  echo "I am greater than or equal to 5"
else
  echo "I am less than 5"
fi
```

Now, if we want to compare strings, we use the normal comparison operators:

```
if [[ $var <op> num ]];

# Valid <op> include:
## !  NOT
## == Equal to
## != Not Equal to

## Example usage:
name="Ali"
if [[ $name == "Ali" ]]; then
  echo "Welcome Ali"
else
  echo "Welcome $name"
fi
```

> When a variable is read, even if we enter an input, the input will still be a string.

Example:

```
read -p "Enter age: " age
## If we want to see if we're above 18 or not:

if [[ $age == "18" ]]; then
  echo "We're 18"
else
  echo "We're $age"
fi
```

Difference between `[` and `[[`:

The main difference between `[` and `[[` is that `[` is the traditional test command, while `[[` is an enhanced version of it. `[[` is preferred in bash scripting because it provides additional features such as pattern matching and the ability to use logical operators directly within the conditional expression.

One key difference is, when we're using Strings that have whitespaces (" "), we need to specify them like this: `[ -f "$file" ]`, whereas, with `[[`, we can easily use: `[[ -f`

`$file ]]`

Another difference is that `[` requires spaces between the brackets and the operands and operators, while `[[` does not. For example, with `[`, we would write `[ $age -eq 18 ]`, whereas with `[[`, we can write `[[ $age -eq 18 ]]`.

In general, it is recommended to use `[[` for conditional expressions in bash scripting, as it provides greater flexibility and ease of use.

Class Tasks:

1.A. Write a simple bash script that will prompt user for their name, age and section and then print them out, on new lines. Output should be somewhat similar to this:

```
Name: Ahmed
Age: 19
Section: A
```

1.B. Keeping the first script, add a check for user that the Age must be greater than or equal to 19. Write this in a new script. If they're younger than 19, show them the following message:

```
You're younger than 19. Age is <age>.
```
