Lab 5 - Advanced Bash Scripting

IF-ELSE_IF-ELSE

Similar to Python, else-if in bash is written as `elif`.

```
#!/bin/bash
## Example question: Ask the user for age, and check if they're less than 13
## then print "Child.", if they're less than or equal to 19, then print "Teenager"
## Otherwise print "Adult"

read -p "Enter your age: " age

if [[ $age -lt "13" ]]; then
  echo "Child"
elif [[ $age -lt "19" ]]; then
  echo "Teenager"
else
  echo "Adult"
fi
```

Multiple Conditions

Similar to C++, we can chain multiple conditions using `&&` and `||` operators.

> Single `&` and Single `|` in Bash have different meaning. In C++, they mean bitshifting, in bash, both of them are used for different purposes.

Keeping in mind the above question, we'll now implement limits to that question. So,

You need to check, if the user is greater than or equal to 5 and less than 13, then print `Child`, if they're less than 5 and greater than 1, print `Infant`, if they're greater than 13 or equal to and less than 20, print `Teenager`, if they're greater than 19, print `Adult.` Else, print `Invalid age`.

```
#!/bin/bash

read -p "Enter your age: " age

if [[ $age -ge 5 && $age -lt 13 ]]; then
```

```
  echo "Child"
elif [[ $age -ge 13 && $age -lt 20 ]]; then
  echo "Teenager"
elif [[ $age -gt 19 ]]; then
  echo "Adult"
else
  echo "Invalid age."
fi
```

Special Conditional Operators

There are certain operators in bash that are very useful for several different purpose. A table of those is:

Operator	Usage	Description
-n	-n <VARIABLE>	Checks if the lenght of variable is greater than zero
-z	-z <VARIABLE>	Checks if the variable is empty
-h	-h <FILE_NAME>	Checks if the file exists and is a symbolic link (shortcut)
-r	-r <FILE_NAME>	Checks if the file exists and is readable
-w	-w <FILE_NAME>	Checks if the file exists and is writeable
-x	-x <FILE_NAME>	Checks if the file exists and is executable
-d	-d <FOLDER>	Check if a folder exists
-f	-f <FILE_NAME>	Checks if a file exists

Example usage:

Let's consider a simple proble, where we'd like to see, if the `/tmp/test.txt` doesn't exist, then create it:

```
#!/bin/bash

if [[ ! -f "/tmp/test.txt" ]]; then
  touch "/tmp/test.txt"
else
  echo "File already exists"
fi
```

Pipes and Redirections

Pipes (`|`):

Pipes allow you to take the output of one command and use it as the input for another command. This is extremely useful for chaining commands together to perform complex operations.

Example:

```
ls -l | grep ".txt"
```

In this example, the `ls -l` command lists files in the current directory, and the `grep ".txt"` command filters the list to show only files with a ".txt" extension.

Redirections (`>` , `<` , `>>`):

Redirections are used to control input and output for commands. They allow you to redirect the output of a command to a file or read input from a file.

- `>`: Redirects command output to a file. If the file already exists, it will be overwritten.

  Example:

  ```
  echo "Hello, world!" > greeting.txt
  ```

- `>>`: Redirects command output and appends it to a file. If the file doesn't exist, it will be created.

  Example:

  ```
  echo "Appended text" >> greeting.txt
  ```

- `<`: Redirects input to a command from a file.

  Example:

  ```
  cat < input.txt
  ```

Special Variables in Bash

`$0`, `$1`, `$#`:

- `$0`: Represents the name of the script or the command itself. If you run a script, `$0` will be the script's name. If you run a command, it will be the command's name.
- `$1`, `$2`, ...: These represent the positional parameters or command-line arguments. `$1` is the first argument, `$2` is the second, and so on.
- `$#`: Represents the number of arguments passed to a script or command.

Example:

```
#!/bin/bash

echo "Script name: $0"
echo "First argument: $1"
echo "Second argument: $2"
echo "Number of arguments: $#"
```

If you run the script with `./myscript.sh arg1 arg2`, it will output:

```
Script name: ./myscript.sh
First argument: arg1
Second argument: arg2
Number of arguments: 2
```

This covers Pipes, Redirections, and some common special variables in Bash. These concepts are fundamental for scripting and working with the Bash shell.

Exit Status

Exit status (also known as exit code or return code) in Bash is a numeric value that a command or script returns to the calling process to indicate whether the execution was successful or encountered an error. The exit status is a way to communicate the outcome of a command's execution to the parent shell or calling script.

There are certain codes that we need to understand for this:

Code	Meaning
0	Success
Non-Zero (<0 | >0)	Failure

Inside a bash script, the Exit status of a command can be obtained using the `$?` .

Let’s just say, we ran `echo test > whoami`. But the command fails. In our program, we need to handle a scenario to do a certain task in case the command fails, otherwise do something else.

```
#!/bin/bash

echo test > whoami

if [[ $? -eq 0 ]]; then # Meaning the command succeeded
  echo "Command succeeded."
else
  echo "Command failed"
fi
```

> Using the same principal, we can also exit our own scripts using a different number to showcase success or failure of our script.

Class Task(s):

Create a Bash script that classifies a person's age based on the input provided. This script should:

1. Prompt the user to enter their age.
2. Check if the age is a valid positive integer.
3. If the age is valid, use conditional statements to classify the age as "Child" if it's between 5 and 12, "Teenager" if it's between 13 and 19, "Adult" if it's 20 or older, and "Invalid age" for any other value.
4. Display the classification to the user.
5. If the age is not a valid positive integer or the input is not a number, display an error message.