ULI101: INTRODUCTION TO UNIX / LINUX AND THE INTERNET

WEEK 10: LESSON 2

POSITIONAL PARAMETERS /
COMMAND SUBSTITUTION / MATH OPERATIONS
TESTING CONDITIONS / CONTROL FLOW STATEMENTS (LOGIC / LOOPS)

LESSON 2 TOPICS

Positional Parameters

- Definition / Purpose / Usage / Demonstration

Command Substitution / Math Operations

- Definition / Purpose / Usage / Demonstration

Control Flow Statements

- Definition / Purpose
- Exit Status **$?** / Testing Conditions (**test**) / Demonstration
- Control Flow Statements (**if**, **if-else**, **for**) / Demonstration

Perform Week 10 Tutorial

- Investigation 2
- Review Questions (Questions Part A **#3,4** , Part B **Walk-Thru #2**)

Work on Assignment #3: (Sections 3 and **4)**

POSITIONAL PARAMETERS

arg1 arg2 arg3 ... argN

A positional parameter is a variable within a shell program;
its value is set from an ***argument*** *specified on the command line*
that invokes the program.

Positional parameters are numbered and are referred to
*with a preceding "**$**": **$1**, **$2**, **$3**, and so on.*

Reference: http://osr600doc.xinuos.com/en/SDK_tools/_Positional_Parameters.html

POSITIONAL PARAMETERS

arg1 arg2 arg3 ... argN

Assigning Values as Positional Parameters

There are **two methods** to **assign values** as positional parameters:

- Use the `set` command inside a shell script with values as **arguments**
- Run a shell script with **arguments** (i.e. like a command)

POSITIONAL PARAMETERS

arg1 arg2 arg3 ... argN

Using the set command:

```
set apples oranges bananas
```

You place a dollar sign ($) prior to the number corresponding to the position of the argument

Examples:

```
echo $1
echo $2
echo $3
```

```
set apples oranges bananas
echo $1
apples
echo $2
oranges
echo $3
bananas
echo $4
```

POSITIONAL PARAMETERS

arg1 arg2 arg3 ... argN

Running a Shell Script with Arguments:

You would use **positional parameters** in your shell script that would **expand** the positional parameters with its stored value.

Here are the contents of the shell script called **`myScript.bash`**:

```
#!/bin/bash

echo "First argument is $1"
echo "Second argument is $2"
```

You would then issue the **myScript.bash** shell script with **arguments** that would be used within the shell script. For Example:

```
./mySript.bash apples oranges
```

```
cat myScript.bash
#!/bin/bash

echo "First argument is $1"
echo "Second argument is $2"

chmod u+x myScript.bash
./myScript.bash
First argument is
Second argument is

./myScript.bash apples oranges
First argument is apples
Second argument is oranges
```

POSITIONAL PARAMETERS

arg1 arg2 arg3 ... argN

The positional parameter `$0` refers to either the **name of shell** where command was issued, or **name of shell script file** being executed.

If using positional parameters greater than 9, you need to include number within **braces** `{ }`

Examples:

```
echo $0
echo ${10}
```

```
cat positional.bash
#!/bin/bash

set 10 9 8 7 6 5 4 3 2 1

echo
echo "\$0 is: $0"
echo
echo "\$10 is: $10"
echo
echo "\${10} is: ${10}"

./positional.bash

$0 is: ./positional.bash

$10 is: 100

${10} is: 1
```

POSITIONAL PARAMETERS

arg1 arg2 arg3 ... argN

The `shift` command can be used with positional parameters to move positional parameters to the **left** by one or more positions.

Examples:

```
shift
shift 2
```

```
set canoe tent food water
echo $1
canoe

shift
echo $1
tent

shift 2
echo $1
water
```

SPECIAL PARAMETERS

$ $# $?*

There are a group of **special parameters** that can be used for shell scripting.

A few of these special parameters and their purpose are displayed in the table below.

Parameter	Purpose
`$*`	Display all positional parameters.
`"$*"`	Containing values of all arguments separated by a single space
`"$@"`	Multiple double-quoted strings, each containing the value of one argument
`$#`	Represents the number of parameters (not including the script name)
`$?`	Exit Status of previous command (discussed in next lesson)

```
set 1 2 3 4 5

echo $#
5
echo $*
1 2 3 4 5

pwd
/home/murray.saul
echo $?
0   # zero is true in Unix/Linux

PWD
-bash: PWD: command not found
echo $?
127  # non-zero is false in Unix/Linux
```

POSITIONAL AND SPECIAL PARAMETERS

Task:

Write a **Bash shell script** that accepts arguments from the shell script filename when executed (i.e., just like a regular Linux command).

The *Bash Shell* script will clear the screen and then display the following text (using **special parameters**):

```
Number of arguments are: (number of positional parameters)

The arguments are: (displays of all positional parameters)
```

COMMAND SUBSTITUTION

Command substitution *is a facility that allows a command to be run and its* ***output*** *to be pasted back on the command line as* ***arguments*** *to another command.*

Reference: https://en.wikipedia.org/wiki/Command_substitution

Usage:

`command1 $(command2)` or `command1 `command2``

Examples:

```
file $(ls)
mail –s “message” $(cat email-list.txt) < message.txt
```

```
echo "The current directory is $(pwd)"
echo "The current hostname is $(hostname)"
echo "The date is: $(date +'%A %B %d, %Y')"
```

```
echo "The current directory is $(pwd)"
The current directory is /home/murray.saul

echo "The current hostname is $(hostname)"
The current hostname is mtrx-node06pd.dcm.senecacollege.ca

echo "The date is: $(date +'%A %B %d, %Y')"
The date is: Tuesday March 02, 2021
```

COMMAND SUBSTITUTION

Task:

Write a **Bash** shell script that **sets** all files in your current directory as **positional parameters.**
Use **command substitution** to store all files in your current directory as **positional parameters.**

The *Bash Shell* script will clear the screen and then display the following text
(using special parameters):

```
Number of files in current directory are:
(number of positional parameters)

Here are the filenames:
(displays of all positional parameters)
```

MATH OPERATIONS

Performing **math** calculations can be an important element in shell scripting.

A problem you may experience in shell scripting (as opposed to other programming languages) is that in shell scripting, all characters (including numbers) are stored as **text**.

This can create **problems** when performing math operations.

Demonstration:

```
num1=5;num2=10
echo "$num1+$num2"
5+10
echo "$num1-$num2"
5-10
echo "$num1*$num2"
5*10
```

MATH OPERATIONS

In order to make math operations work in a Linux shell or shell script, you need to **convert** numbers stored as **text** into **binary numbers**.

We can do this by using using a **math construct** consisting two pairs of round brackets `(( ))`

Examples:

```
num1=5;num2=10
echo "$(( $num1 + $num2))"
15

echo "$((num1-num2))"
-5

((product=num1*num2))
echo "$product"
50
```

MATH OPERATIONS

Additional math operators are shown below.

Examples:

```
num1=2;num2=3
echo $((num1/num2))
0
echo $((num1%num2))
3
echo $((num1**num2))
8
echo $((num2++))
4
echo $((num1--))
1
```

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Remainder
**	Exponentiation
++	Increment (increase by 1)
--	Decrement (decrease by 1)

MATH OPERATIONS

Task 1:

Write a **Bash** shell script that prompts the user for the sale **price** of an item
and the **number** of items purchased.

The shell script will display the **total amount** (eg. **price** x **number** of items**)** of the sale.

For simplicity, you can assume prices are just **integers**.

Task 2:

Write a **Bash** shell script that prompts the user prompts the user for **two numbers**.

The shell script will then show the results from **addition**, **subtraction**, **multiplication**
and **division** of those numbers.

CONTROL FLOW STATEMENTS

So far, we have created Bash Shell Scripts that execute Linux commands in a **fixed sequence**.

Although those type of scripts can be useful, we can use **control flow statements** that will **control the sequence** of the running script based on various situations or conditions.

Control Flow Statements are used to make your shell scripts more **flexible** and allow them to **adapt** to *changing situations*.

CONTROL FLOW STATEMENTS

The `$?` (exit status) Special Parameter

The special parameter **`$?`** is used to determine the **exit status** of the previously issued **Linux command** or **Linux pipeline command**.

The exit status will either display a **zero** (representing **TRUE**) or a **non-zero number** (representing **FALSE**).

This method can be used with control-flow statements to **change the sequence** of your shell script execution. We will apply this when we discuss advanced shell scripting in two weeks.

Examples:

```
PWD
echo $?
pwd
echo $?
```

```
PWD
-bash: PWD: command not found
echo $?
127

pwd
/home/murray.saul
echo $?
0

echo "Hi there" | grep Hi
Hi there
echo $?
0

echo "Hi there" | grep Goodbye
echo $?
1
```

CONTROL FLOW STATEMENTS

The `test` Linux Command

The **test** Linux command is used to test conditions to see if they are **TRUE** (i.e. value **zero**) or **FALSE** (i.e. value **non-zero**).

This method can also be used with control-flow statements to **change the sequence** of your shell script execution.

Examples:

```
name="Murray"
test $name = "Murray"
echo $?
test $name = "David"
echo $?
```

```
name="Murray"
test $name = "Murray"
echo $?
0

test $name = "David"
echo $?
1

test $name != "David"
echo $?
0
```

CONTROL FLOW STATEMENTS

Numerical Comparisons with test Command

You **CANNOT** use the **>** or **<** symbols when using the **test** command since those are **redirection** symbols.

You need to use **options** when performing numerical comparisons. Refer to the table below for test options and their purposes.

Option	Purpose
`-eq`	Equal to
`-ne`	Not equal to
`-lt , -le`	Less than, Less than or equal to
`-gt, -ge`	Greater than, greater than or equal to

```
num1=5
num2=10
test $num1 -eq $num2
echo $?
1

test $num1 -lt $num2
echo $?
0

test $num1 -ne $num2
echo $?
0

test $num1 -ge $num2
echo $?
1
```

CONTROL FLOW STATEMENTS

The `test` Linux Command: Additional Options

There are other **comparison options** that can be used with the **test** command such as testing to see if a **regular file** or if **directory pathname exists**, or if the regular file pathname is **non-empty**.

Refer to the table below for some of those additional options.

Option	Purpose
`-f file_pathname`	Regular filename exists
`-d file_pathname`	Directory filename exists
`-s file_pathname`	Regular filename is non-empty
`-w file_pathname`	file exists / write permission is granted

```
mkdir mydir
test -d mydir
echo $?
0

touch myfile.txt
test -f myfile.txt
echo $?
0

test ! -f myfile.txt
echo $?
1

test -s myfile.txt
echo $?
1

test ! -s myfile.txt
echo $?
0
```

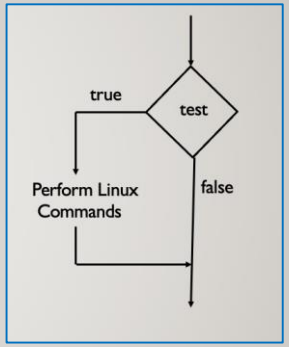
CONTROL FLOW STATEMENTS - LOGIC

Logic Statements

A **logic statement** is used to determine which Linux commands to be executed based on the result of a **test condition** or **command** (i.e. **TRUE** if zero value) or **FALSE** (if non-zero value).

There are **several logic statements**, but we will just concentrate on **if** statement and the **if-else** statements.

CONTROL FLOW STATEMENTS - LOGIC



if Control Flow Statement

If the **test** command returns a **TRUE** value,
then the Linux Commands between ***then*** and ***fi*** statements are executed.

If the **test** command returns a **FALSE** value,
the ***if*** statement is **by-passed**.

Usage:

```
if test condition
 then
    command(s)
fi
```

```
cat if.bash
#!/bin/bash

read -p "Enter First Number: " num1
read -p "Enter Second Number: " num2

if test $num1 -lt $num2
then
  echo "Less Than"
fi

./if.bash
Enter First Number: 5
Enter Second Number: 10
Less Than

./if.bash
Enter First Number: 10
Enter Second Number: 5
```

CONTROL FLOW STATEMENTS - LOGIC

Using [ ] to Represent test Command

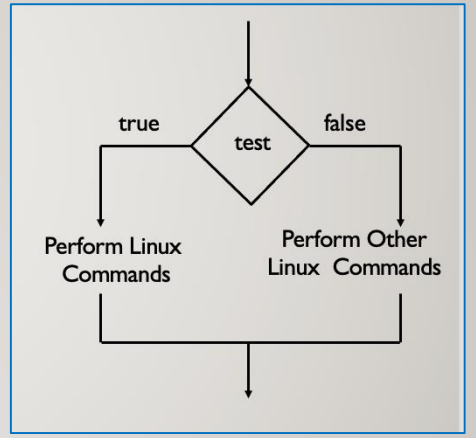
A set of square brackets **[ ]** can be used to represent the **test** command.

NOTE: There must be **spaces** between the **square brackets** and the **test** condition.

Example:

```
num1=5
num2=10
if [ $num1 -lt $num2 ]
 then
    echo "Less Than"
fi
```

CONTROL FLOW STATEMENTS - LOGIC



if-else Control Flow Statement

If the test condition returns a **TRUE** value, then the Linux Commands between the **then** and **else** statements are executed.

If the test returns a **FALSE** value, then the the Linux Commands between the **else** and **fi** statements are executed.

Usage:

```
if test condition
 then
    command(s)
 else
    command(s)
fi
```

```
cat if-else.bash
#!/bin/bash

read -p "Enter First Number: " num1
read -p "Enter Second Number: " num2

if [ $num1 -lt $num2 ]
then
   echo "Less Than"
else
   echo "Greater Than or Equal To"
fi

./if-else.bash
Enter First Number: 3
Enter Second Number: 5
Less Than

./if-else.bash
Enter First Number: 5
Enter Second Number: 3
Greater Than or Equal To
```

CONTROL FLOW STATEMENTS - LOGIC

Instructor Demonstration

Task1:

Write a **Bash** shell script that will first set a variable called **course** to the value **uli101** (lowercase). Then the shell script will clear the screen and prompt the user for the current course code. Use **logic** that if the user's entry does match the value contained in the variable **course**, the following text is displayed:

```
You are correct
```

Task2:

Modify the previous Bash Shell script to display the alternative message if the user's entry does NOT match the value (stored in the variable called **course**)
then the following alternative text is displayed:

```
You are incorrect
```

CONTROL FLOW STATEMENTS - LOOPS

Loop Statements (iteration)

*A **loop** statement is a series of steps or sequence of statements **executed repeatedly** zero or more times satisfying the given condition.*

Reference:
https://www.chegg.com/homework-help/definitions/loop-statement-3

CONTROL FLOW STATEMENTS - LOOPS

The `for` Loop

There are several loops, but we will look at the **for** loop using a **list**.

Usage:

```
for item in list
do
    command(s)
done
```

The variable **item** will hold one item from the list every time the loop iterates (repeats) the commands between the **do** and **done** reserved words.

A **list** can consist of a series of arguments (separated by spaces) or supplied by command substitution

CONTROL FLOW STATEMENTS - LOOPS

The **for** Loop

Example:

```
for x in apples oranges bananas
do
    echo "The item is: $x"
done
```

```
cat for.bash
#!/bin/bash

for x in apples oranges bananas
do
    echo "The item is: $x"
done

./for.bash
The item is: apples
The item is: oranges
The item is: bananas
```

CONTROL FLOW STATEMENTS - LOOPS

Task:

Write a **Bash shell script** that **sets** all files in your current directory as **positional parameters.** Use **command substitution** to store all files in your current directory as **positional parameters.**

The *Bash Shell* script will clear the screen and then display the following text (using special parameters). Use a for loop to display each filename on a SEPARATE line using a **for** loop:

```
Number of files in current directory are:
(number of positional parameters)

Here are the filenames:
(displays each positional parameters on a SEPARATE line)
```

HOMEWORK

Getting Practice

To get practice to help perform **assignment #3**, perform **Week 10 Tutorial:**

- INVESTIGATION 3: COMMAND SUBSTITUTION / MATH OPERATIONS
- INVESTIGATION 4: USING CONTROL FLOW STATEMENTS IN SHELL SCRIPTS
- LINUX PRACTICE QUESTIONS (Part A 3,**4 ,** Part B **Walk-Thru #2**)

Work on Assignment #3:

- **Section 3:** Interactive Shell Environment
- **Section 4:** Introduction To Scripting (phone)