Lecture 10A: Introduction to Bash Scripting

Note: All relevant examples can be copied from:
`~eric.brauer/uli101/scripts`

Note #2: Scripts often require debugging. If you working on the assignment and it isn't allowing you to proceed, be sure to **run the script by itself in the terminal**. Pay attention to the error messages that Bash gives you. Also, turn on line numbers!

- Use `cat -n` to match line numbers to errors, or
- use `:set number` in Vim.

If you don't have error messages or the output of your script with line numbers, I can't help you with debugging!

Introduction to Scripting

So far we've been talking about Bash as being our shell, a place where we type in commands, press Enter, and have them executed. In this section we will discuss Bash as a *scripting language*. Bash lacks many features of more dedicated programming languages– for example there are no classes or objects– but it is a fantastic tool for many jobs.

- Bash is interpreted at runtime. The interpreter will try its best to figure what you want it to do.
- Unlike C, there is no compiler. There is no real way to know if the script has errors until you try running it.
- Bash scripts are basically plaintext documents that include any of the shell commands that you already know.
- For that reason, it is best practice to define how we want our script interpreted. We do this by including a *shebang* at the top of the script.
- The shebang will include the absolute path to the Bash interpreter. (You can use `which bash` to check this)
- Our shebang on Matrix (and in 99% of cases) will be: `#!/bin/bash`
- It is also best practice to include a file suffix so that people will recognize your scripts. It's best to use either `.sh` or `.bash`.

Example:

Follow these steps to create your very first script:

1. Run command: `echo "Hello World!"` What happens?

2. Now let's put that same command into a new file. Use `vim hello.sh` or `nano hello.sh` to create a new file.
3. Insert the following text:

```
#!/bin/bash

# This is a comment. All comments begin with #.

echo "Hello World!"
```

1. Let's give this file execute permission: `chmod u+x test.sh` .
2. Use `ls -l` check the permissions.`
3. Try running it. Type in: `test.sh` . What happens?

Variables

Let's open the file again in Vim/Nano and edit it so that it looks like this:

```
#!/bin/bash

name=Eric

echo "Hello $name!"
```

Use *your name*, not mine.

The first line after the shebang will create a variable called `name` and assign a it a *value* of `Eric` . Note that variables in Bash are treated like strings, and don't need to be declared.

With the second line, we want the user to be seeing the *value* of the variable, not the name of the variable. In order for the interpreter to understand that we want this substitution to occur, we use `$` followed by the variable. (It's also acceptable to put the variable name in `{ }` s like this: `${name}` .

Notes on Quotes

1. Change the line to your full name, ex: `name=Eric Brauer` . What happens?
2. You'll see: `Brauer: command not found` .
3. Change to "Eric Brauer". It works.

The space isn't being interpreted as the contents of the variable, but as a separate command. We need to make it clear where the assignment of the variable begins and ends, and to do this we will use double quotes.

Note: Enclosing characters in double quotes (") preserves the literal value of all characters within the quotes, with the exception of $, `, ₩, and, when history expansion is enabled, !.

Another Special Note:

Look at a file called `SIMPLE-VARS.sh` in my `uli101/scripts` directory.

```
cat -n ~eric.brauer/uli101/scripts/SIMPLE-VARS.sh
```

```
 1  #/bin/bash
 2
 3
 4  # This is a simple variable assignment:
 5  variable_name="Variable Value"
 6
 7  # Echo will print:
 8  echo "Hello World!"
 9
10  # Echo with double quotes will allow substitution:
11  echo "This prints the value of $variable_name."
12
13  # Echo with single quotes will prevent substitution:
14  echo 'This does not print the value of $variable_name.'
```

When you run this script, you should see this output:

`~eric.brauer/uli101/scrdipts/SIMPLE-VARS.sh

```
Hello World!
This prints the value of Variable Value.
This does not print the value of $variable_name.
```

- Single quotes: everything is literal.
- Double quotes: everything is literal except for "$! ` ₩"

So the takeaway is: double quotes are best used when you need to use special characters like `$` s to perform substitutions.

Read

Take a look at a file called `SIMPLE-READ.sh` :

```
cat -n ~eric.brauer/uli101/scripts/SIMPLE-READ.sh
```

```
 1  #!/bin/bash
 2
 3  # Let's start with a variable:
 4  variable="nothing"
 5
 6  # We'll use an echo command to ask the user for input. -n will suppress a
     newline:
 7  echo -n "Enter a new value for our variable: "
 8  # Read will store the input we give it.
 9  # We'll also give it the name of the variable where we're storing the user
     input.
10  read variable
11
12  echo "You entered $variable."
```

`read` is a simple way to ask the user for input. Whatever the user types will be stored into the variable that you specify.

However, we often don't want to create our scripts like this, since it requires the user to be sitting at a chair, reading the message and typing in their response. Remember,

a lot of times our scripts are supposed to run automatically.

Passing In Arguments

You remember arguments, they are the usual way that we have interacted with commands such as `cp`, `rm`, `ls` ...

```
cat -n ~eric.brauer/uli101/scripts/SIMPLE-ARGS.sh
```

```
 1  #/bin/bash
 2
 3  # Arguments come from the user. Such as this example:
 4
 5  # `rm scripts`
 6  #  ^  ^-----------argument
 7  #  \command
 8
 9  # Try running this script with zero arguments.
10
11  echo "Number of arguments from the user: $#."
12
13  # Now Try running this script again. This time enter an argument after the
     script name.
14
15  # `SIMPLE-ARGS.sh dog`
16  echo "The first argument passed in from the user: $1."
17
18  # Now try running this script with two arguments.
19  echo "The second argument passed in from the user: $2."
```

Notice that `$#` is a special, built-in variable. It contains a number which represents the number of arguments that Bash has seen passed into the script.

When we run the script without arguments, we see output that looks like this:

```
~eric.brauer/uli101/scripts/SIMPLE-ARGS.sh
```

```
 Number of arguments from the user: 0.
The first argument passed in from the user: .
The second argument passed in from the user: .
```

Now let's follow the instructions inside the script. Run the script with one and then two arguments:

```
~eric.brauer/uli101/scripts/SIMPLE-ARGS.sh dog
```

```
 Number of arguments from the user: 1.
The first argument passed in from the user: dog.
The second argument passed in from the user: .
```

```
~eric.brauer/uli101/scripts/SIMPLE-ARGS.sh dog cat
```

```
 Number of arguments from the user: 2.
The first argument passed in from the user: dog.
The second argument passed in from the user: cat.
```

`$1` and `$2` are more special builtin variables that contain the contents of the first and second arguments. We can pass in up to nine arguments like this: $1 $2... $9. This should be enough for most scripts. If the user doesn't pass in an argument, then the contents of its builtin variable are *null* (empty).

IF Statements

You'll notice that the example above contains different output based on the arguments that the user entered. We can use this behaviour to demonstrate *conditional logic*. If you are familiar with programming, then this should be clear to you. The only thing different is the *syntax*, that is, the keywords that Bash uses to interpret your logic.

```
cat -n ~eric.brauer/uli101/scripts/SIMPLE-IF.sh
```

```
     1  #/bin/bash
     2
     3  # Arguments come from the user. Such as this example:
     4
     5  # `rm scripts`
     6  #  ^  ^-----------argument
     7  #  \command
     8
     9  # Try running this script with zero arguments.
    10
    11  echo "Number of arguments from the user: $#."
    12
    13  # Now Try running this script again. This time enter an argument after the
         script name.
    14
    15  # `./SIMPLE-ARGS.sh dog`
    16
    17  if [ $# = 1 ]
    18  then
    19      echo "The argument passed in from the user: $1."
    20  fi
    21
    22  # Now try running this script with two arguments.
    23
    24  if [ $# = 2 ]
    25  then
    26      echo "The two arguments passed in from the user: $2 and $1."
    27  fi
    28
    29  # Notice those last two are reversed.
```

```
 if [ CONDITION ]
then
    # ACTION that runs if the condition is TRUE
fi
```

In our example above, we check the number of arguments, and run different `echo` commands based on whether the number of arguments is zero, one, or two. If the condition is false, then we skip every line between `then` and `fi`.

We can combine this with an `else` statement. In this case, instead of just skipping over actions, we can specify alternative actions to take if the condition is false.

```
cat -n ~eric.brauer/uli101/scripts/SIMPLE-IF-ELSE.sh
```

```
 1  #!/bin/bash
 2
 3  condition=1
 4
 5  if [ $condition = 1 ]
 6  then
 7          echo "The condition is true."
 8          echo "Action 1."
 9  else
10          echo "The condition is false."
11          echo "Action 2."
12  fi
13
14  echo "Open this script in Vim or Nano, and change the value of 'condition'."
```

Be sure to open this file, and modify if so that you can see how it works when the condition is false. The main idea is, your script will *either* reach lines 7 and 8 *or* lines 10 and 11, but never both.

Regardless of whether the condition is true or false, after the `fi` on line 12, the script will run each line normally. That means your script will *always* reach the action on line 14.

Summary

Be sure to remember the syntax for:

- print output
- read input
- read arguments
- if statements

Quotes:

- "": weak quotes (substitution of $ occurs)
- '': strong quotes (substitution of $ does not occur)
- ₩: escape character

Variables:

- `$#` : Returns the number of arguments as an integer
- `$1..$9` : Returns positional arguments