Lecture 10B: Introduction to Bash Scripting II

**Note: If you are using vi(m), you can turn on line numbers by entering command mode with `:`, and then entering `set number`. Do this before showing your scripts to the professor. You can make this persistent by adding the line `set number` to the file `.vimrc`.

Environmental Variables

In the previous lecture, we discussed some *builtin variables*, that is, variables that exist without the having to create them or assign them value. There exist quite a few more *environment variables*. We don't have to use a script to see these variables, we can call them directly from the shell.

```
echo "Hello $USER.
```

You didn't need to assign your username to this variable, it was already stored in memory. To see a full list of environment variables, use this command:

```
env
```

This should return a long list of variables. You won't be interacting with most of these, but you might recognize the value for some of these.

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

```
    1  #!/bin/bash
    2
    3  # Notice that we don't assign values to any variables. These are all
        environment.
    4
    5  echo "Hello $USER."
    6
    7  if [ $PWD = $HOME ]
    8  then
    9      echo "Welcome home!"
   10  else
   11      echo "You might be lost! Use 'cd ~' to get back home."
   12  fi
   13
   14  # Now copy this script elsewhere to see if it works.
```

The PATH Variable

The PATH is a variable that contains all the paths where the shell expects to find programs or commands. For example, it contains the path `/bin/`. If you look there, you'll find a lot of the important commands you've been using, such as `rm`, `cp`, and so on. On *nix systems we want to limit where scripts can be run as a security measure. Most likely, your current directory is your home, and thats probably not in $PATH. There's a couple ways we can resolve this. We can move our script into one of the directories included in PATH, we can add our current location to PATH (which is what you will do in Assignment 3), or you can *specify your current location when calling the script*.

```
./test.sh
```

It works with ./ because POSIX specifies that a command name that contain a / will be used as a filename directly, overriding whatever's in $PATH. You could have used absolute path for the exact same effect, but ./ is shorter and easier to write.

One thing you'll see on Matrix if you `echo $PATH`:

```
:.:
```

Each directory is delimited by colons. `.` here means your current directory. So by default on Matrix, your current directory is always an accepted location for scripts. **Note that this isn't common, and in most cases if you try to run a script without adding it to PATH, you will get a "file not found error!**

Some Useful Environment Variables

- PS1 - primary prompt
- PWD - present working directory
- HOME - absolute path to user's home directory, similar to ~
- HOST - name of host
- USER - name of current user

Let's review a previous example:

```
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'."
```

Recall that we either run `Action 1` or `Action 2` but not both. However, we will always reach the echo command on line 14.

Compare that example with this one:

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

```
 1  #!/bin/bash
 2
 3  echo -n "Is everything alright? (y/n) "
 4
 5  read response
 6
 7  if [ $response = "y" ]
 8  then
 9          echo "Good to hear!"
10          exit 0
11  else
12          echo "Sounds like an error!"
13          exit 1
14  fi
15
16  echo "We will never reach this line."
```

Try running to verify whether or not you will reach line 16. You shouldn't.

When you try running this, you will see output like this:

```
 Is everything alright? (y/n) n
Sounds like an error!
```

- The first observation to make is this: when your script reaches an `exit` statement, it halts immediately. It will no longer continue.
- The second observation is that `exit` statements have a number associated with them. This number is invisible, but we can read it with another builtin variable: `$?` . Try running this script again, but immediately after running it, type in `echo $?` and see what gets printed.

Exit Codes

`$?` will return the *exit code* of whatever command just completed. We've just seen exit codes in our script, but exit codes are returned by every Linux command that you've learned. By convention, an exit code of **zero** means no errors were encountered and a **non-zero** exit code means that something went wrong, or didn't behave the way we expected it to. The exit codes and what they mean are chosen by the programmer. Often **man** pages will indicate different error codes.

Again, Summarising Exit codes: - 0: Everything's alright - not 0: Somethings's wrong - All Linux commands return exit codes - The meaning of exit codes is something chosen by programmers, and they often document exit codes in `man` pages - `$?` will return the exit code of *the most recent command*.

Application Of Exit Codes: Grep

`grep` indicates to us that *nothing* was returned with a *1*, and a *0* will indicate that there was a positive match.

```
grep "ford" cars
```

```
ford    mustang 65      45      17000
ford    ltd     83      15      10500
```

```
echo $?
```

```
0
```

```
grep ferrari cars
```

```
echo $?
```

```
1
```

We can use exit codes in our scripts to make sure that commands are succeeding. This is an important concept to understand for real-world situations.

Here's an example of an **IF** statement using grep:

```
grep -i "$1" cars # Bonus: what happens if you replace "" with ''?
if [ $? != 0 ]
then
    echo "Sorry, $1 is not in the cars file" >&2
    exit 1
fi
```

In this example, we run a grep command using the first argument passed to our script. *Anything other than success* will get caught by our **if** statement, and it will print out the "Sorry, nothing found" statement. The exit code that our script will return is 1, which is bad.

File Checking

Bash has many file-checking operators that are very useful for the types of jobs that we perform with scripts. For example, `-f` will return TRUE if a file exists.

Note: using `!=` reverses the logic, and means "does not equal."

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

```
     1  #!/bin/bash
     2
     3  fileInQuestion=$1
     4
     5  if [ $# != 1 ]
     6  then
     7                  echo "Please enter only one argument."
     8                  exit 1
     9  fi
    10
```

```
11  if [ -f $fileInQuestion ]  # check if the file exists yet.
12  then
13                  echo "$fileInQuestion exists."
14  else
15                  echo "$fileInQuestion does not exist."
16                  touch $fileInQuestion
17                  echo "...but now it does."
18  fi
19  exit 0
```

Notice that the filename we are checking is being passed in as an argument from the user. If the number of arguments does not equal 1, then the script encounters an `exit 1` statement. Remember that a non-zero exit code indicates an error. Otherwise, we are assigning the variable `fileInQuestion` with the value from the first argument.

At line 11, we run a file check on `$fileInQuestion`. That check is either true or false, and observe what happens when the condition is false.

other test options:

- f - check if file is an ordinary file
- d - check if directory exists
- s - check if file exists, with file size greater than 0
- r - check if file exists, and you have read permission
- w - check if file exists, and you have write permission
- x - check if file exists, and you have execute permission
- z - check if string has zero length

Shell Arithemtic and Comparators

Unlike C/C++ for example, Bash scripts aren't usually used to do math. There are better languages for this purpose.

There are many ways to do math from the shell, but probably the easiest is to put your expression inside of a `$(( ))`.

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

```
 1  #!/bin/bash
 2
 3  x=10
 4  y=5
 5
 6  result=$(( $x + $y ))
 7
 8  echo $result
```

Similarly, doing math comparisons is different from other programming languages. `=` and `!=` work fine for strings, but to test for greater or less than, we use:

- `-eq` : equal to
- `-ne` : not equal to
- `-gt` : greater than
- `-ge` : greater than or equal to

- `-lt` : less than
- `-le` : less than or equal to

```
#!/bin/bash

x=10
y=5

result=$(( $x + $y ))
if [ $result -gt 10 ]
then
    echo "The sum is greater than 10."
fi
```

Summary

Be sure to remember the syntax for:

- exit codes
- math

Variables:

- $?: exit code of previous command
- $PATH: allowed paths for scripts

File checks

- -f: check if file exists and is an ordinary file
- -d: check if directory exists
- !: reverses the check: (`[ ! -d "$path" ]` : if directory `$path` doesn't exist...)

Comparisons (Numeric):

- lt: less than
- le: less than or equal to
- eq: equal to
- gt: greater than
- ge: greater than or equal to
- ne: not equal to