Lecture 12B: Some More Bash Script Examples

Example 1: *Say Hi To* version 2.0

Consider this example:

```
 #!/bin/bash
# This is a test script! :)

if [ "$#" -lt 2 ] # If the number of arguments is less than two...
then
    echo "Hello $1!" # Say hi to one person. Otherwise...
else
    echo "Hello $1 and $2!" # Say hi to two people.
fi
echo "Number of arguments: $#" # print the number of names that were given.
```

This script has a lot of limitations. If you give more than two names, it will say hello to the first two and ignore the rest. It also doesn't handle a case where the user gives *zero* arguments. We can improve this script by using a better *if.. elif.. else* format.

```
 if [ condition ]
then
    action
elif [ second condition ]
then
    second action
else
    default action
fi
```

In this next example, we don't need to think about numerical comparisons such as *less than* `-lt` or *greater than* `-gt`. We can treat the value of `$#` like a string. So let's specify a condition for *zero* arguments, a condition for *one* argument, and a default case for everything else:

```
 #!/bin/bash
# Always comment your code.

if [ $# == 0 ]
then
    echo Usage: please include at least one person to greet. 1>&2
    exit 1
elif [ $# == 1 ]
then
    echo "Hello $1!"
else
    echo "Wow, there are a lot of people in here..."
fi
exit 0
```

So one thing to note about our exit codes: in the first condition we specify an exit code of 1. Once we hit an exit, the program is finished. If there are no arguments, we take the first set of actions and never get to the bottom of the script. We never encounter `exit 0`. However, if either one argument or many arguments are encountered, we *do* reach `exit 0`. This is good: *0* means everything was working properly. *1* means there was an error. We consider the script to be running normally if one or more names have been entered.

The next thing to think about is how to deal with the case where we have many names? How can we handle this case in a way that looks nice? Well, let's try a loop. We want to deal with each name in the arguments list `$*`.

```
 for variable in list
do
    repeated actions
done
```

So now our script might look like this:

```
 #!/bin/bash

# say hi to version 2

if [ $# == 0 ]
then
    echo Usage: please include at least one person to greet. 1>&2
    exit 1
elif [ $# == 1 ]
then
    echo "Hello $1!"
else
    for name in $*
    do
        echo $name
    done
fi
exit 0
```

Please note that you can also use `$@` and you get roughly the same output. For the scope of this course, we can say that both `$*` and `$@` will return all arguments as an *array*. Please note though that there are some differences between the two.

This script now handles 0 arguments by complaining to the user, it handles 1 argument by saying "Hello User!" and it handles 2 or more arguments by simply listing those names, each on a new line:

```
 Eric
Sarah
Yoko
Nate
Paul
...
```

This isn't what we want. We want our script to respond to us by saying hello in a way that is somewhat closer to real communication.

First, let's make it so that before we enter the loop, we say Hello, and using `-n` to suppress the newline...

```
echo -n "Hello $1"
```

...then inside our loop we insert "and" and a name...

```
echo -n " and $name"
```

...and after iterating through each name in our argument list, we finish with an "!" to make it match up with our output before. This time, we *do* want to finish with a newline, because this is the end of our script.

```
echo '!' # Notice the single quotes here!
# We don't want this ! to be interpreted as anything other than punctuation.
```

(If you don't understand why we want a newline, go ahead and try this script with a `-n` flag and see how it looks!)

There's one last piece of the puzzle: If I run this script with arguments like this:

```
./say-hi-to2 Eric Sarah Yoko Nate Paul
```

...My output looks like this:

```
Hello Eric and Eric and Sarah and Yoko and Nate and Paul!
```

...So we can conclude that the value of `$1` and the *first* value of `$*` are the same. What we want to do is discard the first value of our argument list and to *shift* all of our arguments up one:

Before Shift:

$1	$2	$3	$4	$5
Eric	Sarah	Yoko	Nate	Paul

After Shift:

$1	$2	$3	$4	$5
Sarah	Yoko	Nate	Paul	–

Conveniently enough, the name of this command is `shift`. We will call it after saying "Hello":

```
#!/bin/bash

# say hi to version 2

if [ $# == 0 ]
then
    echo Usage: please include at least one person to greet. 1>&2
    exit 1
```

```
elif [ $# == 1 ]
then
    echo "Hello $1!"
else
    echo -n "Hello $1"
    shift # discard the value $1, and shift all arguments up
    for name in $*
    do
        echo -n " and $name"
    done
    echo '!'
fi
exit 0
```

Bonus: If you are looking for practice, I challenge you to create *say-hi-to3*. Instead of separating names with the word 'and', separate them with commas, until you reach the last name, which should have an 'and' preceding it.

```
./say-hi-to3 Eric Sarah Yoko Nate
```

```
Hello Eric, Sarah, Yoko and Nate!
```

Example: How Much Reading Have You Done?

Here's another example that will output the average words per lecture for each of our twelve weeks.

Start with a shebang and comment:

```
#!/bin/bash

# return statistics about the lecture notes: total words and average words per
  lecture.
```

To solve this problem, you'd need to know something about how I have named my files. I have been using Markdown to create lecture notes. Here's how it looks on my computer:

```
ls /home/eric.brauer/uli101/LECTURE_NOTES | grep 'lecture[0-9][0-9ab]*.md'
```

```
lecture10.md
lecture11a.md
lecture11b.md
lecture11.md
lecture12.md
lecture2a.md
lecture2b.md
lecture3a.md
lecture3b.md
lecture4a.md
lecture4b.md
lecture5a.md
lecture5b.md
```

```
lecture6.md
lecture7.md
lecture8.md
lecture9.md
```

So we have one tool which will return number of words in a file, if you'll remember: `wc` with the `-w` option. We will need to run that command for each file.

```
wc -w lecture10.md
```

```
2245 lecture10.md
```

We also need to parse this result. We will want to sum these numbers, so we need to strip the filename from the result. Let's use `awk` to do this. We can then store this in a variable.

```
sum=$(wc -w lecture10.md | awk '{print $1}')

echo $sum
```

```
2245
```

Here is our script so far...

```
#!/bin/bash

# return statistics about the lecture notes: total words and average words per
  lecture.

sum=0
count=0

for file in $(ls /home/eric.brauer/uli101/LECTURE_NOTES | grep 'lecture[0-9][0-
  9ab]*.md')
do
    sum=$(( $(wc -w $files | awk '{print $1}') + $sum ))
    count=$(( $count + 1 ))
done
```

Finally, we do some basic error checking and output the results. The final product:

```
#!/bin/bash

# return statistics about the lecture notes: total words and average words per
  lecture.

sum=0
count=0

for file in $(ls /home/eric.brauer/uli101/LECTURE_NOTES | grep 'lecture[0-9][0-
  9ab]*.md')
do
    sum=$(( $(wc -w $files | awk '{print $1}') + $sum ))
    count=$(( $count + 1 ))
done

echo "Total words: $sum"
```

```
if [ $count -gt 0 ]
then
    echo "Average length of a lecture: $(( $sum / $count ))"
fi
```

And here is the result of the script...

```
 Total words: 29123
Average length of a lecture: 1456
```

That's a lot of words! I hope it's been worth it... :smile: That's all folks!