Scripting Control Structures I

- Shell Scripts
- Shell Script Control Structures
- The *if* ... *then* Construct
- *test*
- The *test* operators
- Using *test* in scripts
- Checking the Arguments to a Script

Shell Scripts

- Most of the rest of this course will deal with *shell scripts*
- Shell scripts are a series of Unix commands placed in a file
  - You can run a shell script like any other program
  - Shell scripts allow you to automate certain routine operations
  - Much of the work in Unix system administration is done using shell scripts
- Shell script programming is **not** like other kinds of programming

Shell Scripts

- Some differences…
  - Unix commands are not uniform in the way they work because each was developed separately by different people
  - The **control structures** used in *shell scripts* are different from those in *programming languages*
- Some will advise you to only write shell scripts for *simple* tasks
  - If you need *if statements* or *loops* to write a script, then you may prefer to use another scripting language, like **Perl** or **Python**
  - Regardless, you should know how to *read* shell scripts

Shell Scripts

- When you run a shell script, your current shell creates a *sub-shell* to run the script
- You must have ***both read and execute permissions*** to run a script without using the **`bash`** command

Shell Script Control Structures

- **Control structures** allow commands in a script to be executed in a different order
- Without control structures, a shell script could only
  - start at the beginning…
  - …and go to the end once

  which would limit what it could do
- There are two basic types of control structures
  - **Conditionals** (Branching)
  - **Loops** (Repetition)

Shell Script Control Structures

- Conditionals are statements where different things happen…
  - based on some condition
  - which is either ***true*** or ***false***
- ***`if`*** statements are the conditional statements that you see most often
- Loops are constructs that ***repeat*** a number of statements until some condition is reached
- Shell scripts can have *both* conditionals and loops

The *if ... then* Construct

- The most basic conditional is the ***if ... then*** construction, which has the format

```
if COMMAND
then
     COMMAND_1
     COMMAND_2
     ...
fi
```

  - where **COMMAND** is **any** Unix command that returns an **exit status**
  - and **COMMAND_1** , **COMMAND_2** , ..., are a series of Unix commands

The *if* ... *then* Construct

- The most commonly used command following if is ***test***
- It is used to test the *truth* of some condition
- Let's look at an example...

```
$ cat if_1.sh
#! /bin/bash
##
## a shell script that demonstrates the Unix if
construct
   ....
```

if_1.sh

```
.....
echo -n "word 1: "
read word1
echo -n "word 2: "
read word2

if test "$word1" = "$word2"
then
    echo The two words match
fi
echo End of script

$ ./if_1.sh
word 1: foo
...
```

```
...
word 2: foo
The two words match
End of script

$ ./if_1.sh
word 1: foo
word 2: bar
End of script
```

The *if ... then* Construct

- ***read*** is a utility that
  - takes input from standard input…
  - …and *stores* that value in the variable given to it as an argument
- Notice that ***echo*** was used with the **-n** option
  - The **-n** option prevents echo from sending a newline character – which would move down to the next line
  - This allows ***echo*** print a prompt for input that will be read by ***read***

The *if* ... *then* Construct

- If the condition is true, then the statements that lie *between* the ***then*** and ***fi*** keywords are run
- ***then*** must either be
  - on a separate line from ***if***
  - or on the same line, but separated by a semi-colon
- Example:

```
$ cat if_2.sh
#! /bin/bash
##
## a shell script that demonstrates the Unix if construct
      ....
```

if_2.sh

```
.....
echo -n "word 1: "
read word1
echo -n "word 2: "
read word2

if test "$word1" = "$word2" ; then
    echo The two words match
fi
echo End of script

$ ./if_2.sh
word 1: foo
word 2: foo
The two words match
End of script
```

- The keyword ***fi*** <u>*must*</u> close the conditional statement
- If you don't, you will get an <u>*error*</u>
- ***fi*** is ***if*** spelled backwards

`test`

- ***`test`*** is a command that is often used in an ***`if`*** statement
- But, while ***`test`*** evaluates the expression that follows, it *does not return true or false* as you would expect
- In Unix, everything is text
  - unless it is enclosed in double parentheses **(( ))**
  - ...in which case the contents are treated as *numbers*
- Most programming languages have boolean variables, which can only have one of two values: **`True`** or **`False`**

`test`

- However, Unix *does not have* boolean values, so how can **`test`** return a value that can be used in an **`if`** statement?
- It returns a value through the *status code*
- Every program on Unix must return a status code before it finishes running
  - If the program runs without a hitch, then it returns a status code of **`0`**
  - If the program runs into a problem, then it returns a status code *greater than* **`0`**

`test`

- When you run ***`test`***
  - It evaluates an expression and…
  - Returns **`0`** if the expression is *true* and **`1`** if the expression is *false*
- In *most* scripting languages, **`0`** is *false* and any value greater than **`0`** is *true*
- But, this variation is useful when writing scripts because it means we *are not limited* to using ***`test`*** in an ***`if`*** statement

test

- **<u>Every</u>** Unix command returns a status code, so we can use **any** Unix command in an ***if*** statement:

```
$ cat if_3.sh
#! /bin/bash
##
## a shell script that demonstrates the Unix if construct

if cd ~ghoffmn
then
    echo was able to go to ~ghoffmn
fi
echo End of script

$ ./if_3.sh
was able to go to ~ghoffmn
End of script
```

test

- This means that a shell script could run a command that might fail – and then take appropriate action if it does
- In ***bash*** , ***test*** is a built-in, a part of the shell
- ***test*** is also a stand-alone program

```
$ which test
/usr/bin/test
```

- ***bash*** will always use the built-in version of ***test*** – unless you specify the *absolute pathname of the executable file*
- The two versions differ slightly

The *test* operators

- ***test*** understands a number of *operators*
  - The operators test for different conditions
  - When used with two arguments, the operators are placed *between* the arguments
- Some operators work only on numbers

Operator	Condition Tested
`-eq`	Two numbers are equal
`-ne`	Two numbers are not equal
`-ge`	The first number is greater than, or equal to, the second
`-gt`	The first number is greater than the second
`-le`	The first number is less than, or equal to, the second
`-lt`	The first number is less than the second

The *test* operators

- ***test*** uses **different** operators when comparing strings

Operator	Condition Tested
=	When placed between strings, are the two strings the same
!=	When placed between strings, are the two strings not the same

- Note that ***test*** uses symbols (**=**) when comparing **strings**
- But letters preceded by a dash (**-eq**) when comparing **numbers**

The *test* operators

- There are a couple of operators that apply only to a **single string**

Operator	Condition Tested
`-n`	Whether the string given as an argument has a length greater than 0
`-z`	Whether the string given as an argument has a length of 0

- In these cases, the operator comes before the string

The *test* operators

- Other operators apply to *files* and *directories*

Operator	Condition Tested
`-d`	Whether the argument is a directory
`-e`	Whether the argument exits as a file or directory
`-f`	Whether the argument is an ordinary file (not a directory)
`-r`	Whether the argument exists and is readable
`-s`	Whether the argument exists and has a size greater than 0
`-w`	Whether the argument exists and is writable
`-x`	Whether the argument exists and is executable

The *test* operators

- There are two additional operators that ***test*** uses when evaluating two test expressions
- They are placed between the two expressions

Operator	Condition Tested
-a	Logical AND meaning both expressions must be true
-o	Logical OR meaning either of the two expressions must be true

The *test* operators

- The exclamation mark **!** is a negation operator
- It inverts the value of the logical expression that follows it
  - It changes a *false* expression to **true**
  - And a *true* expression to **false**
- Some find it **very** hard to remember these operators
- This is why you may prefer **not** to write anything but the simplest shell scripts
- If you need to write a script that uses conditionals, you might consider doing it in a more programmer-friendly scripting language like *Perl* or *Python*

Using *test* in Scripts

- We can use ***test*** in an ***if*** statement

```
$ if test foo = foo
> then
> echo "The two strings are equal"
> fi
The two strings are equal
```

- But, this looks very different from an ***if*** statement in programming languages

Using *test* in Scripts

- To make the ***if*** statement look more like a "real" programming language, Bash provides a synonym for ***test*** a pair of square brackets: **[ ]**
- To test whether the value of **number1** is greater than the value of **number2** , you could write either

```
if test $number1 -gt $number2
```

- or

```
if [ $number1 -gt $number2 ]
```

Using *test* in Scripts

- Whenever you use **[ ]** instead of ***test*** , there **must** be a space before and after each square bracket
- If you don't, you will get an error message

```
$ [ 5 -ne 6]
-bash: [: missing `]'
```

- That's because Bash reads **6]** as a **single** token which it does not understand
- Putting a space between **6** and **]** makes it two tokens

Using *test* in Scripts

- The first thing to do when you get an error in a script using **[ ]** is make sure you have spaces surrounding all your square brackets
- ***test*** **does not return a value to standard output**
  - ***test*** returns *true* or *false* through the exit status
  - An exit status of **0** it means the condition was *true*
  - An exit status of **1** it means the condition was *false*

```
$ [ 5 -eq 4 ]; echo $?
1
```

```
$ [ 5 -ne 4 ]; echo $?
0
```

Checking the Arguments to a Script

- If a script **must** have a certain number of arguments, it should check to see that it has been given them on the command lines
- If a script doesn't get the right number of arguments, then it should print a usage message and exit
- A usage message has a standard form

```
Usage:  PROGRAM_NAME  ARG1  ARG2  ...
```

Checking the Arguments to a Script

- In a usage message, the strings that follow the program name should be a word or words that indicates
  - What kind information was required
  - What kinds of information could be provided
- So if you had a script `test_dr.sh` that needed the name of a directory as an argument it's usage message would be

```
Usage:   test_dr.sh  DIR_NAME
```

Checking the Arguments to a Script

- Let's look at an example

```
$ cat examples_it244/usage_1.sh
#! /bin/bash
# this program demonstrates checking for arguments
# and printing a usage message when
# the expected arguments are not supplied

if test $# -eq 0
then
    echo Usage:  $0  STRING
    exit 1
fi
echo Received argument $1
...
```

Checking the Arguments to a Script

```
...
$ examples_it244/usage_1.sh
Usage: examples_it244/usage_1.sh  STRING

$ examples_it244/usage_1.sh foo
Received argument foo
```

- The script first looks at the number of arguments it gets which is contained in **#**
  - If it receives zero arguments the script prints a usage message and then quits with an exit status of 1
  - Otherwise, it prints the argument it was given

Checking the Arguments to a Script

- The usage message uses the `0` positional parameter which contains the pathname that ran the script
  - The pathname that appears in this usage message is correct, but it is also *confusing*
  - What we *really* want in a usage message is the *filename* part of the pathname
- We can strip out everything from the pathname except the name of the file -- if we use Unix utility called ***basename***

Checking the Arguments to a Script

- ***basename*** takes a pathname as an argument and strips out *everything except for the filename*

```
$ basename examples_it244/usage_1.sh
usage_1.sh
```

- So, a better version of this script would be...

```
$ cat examples_it244/usage_2.sh
#! /bin/bash
# this program demonstrates checking for arguments
# and printing a usage message using basename
.....
```

Checking the Arguments to a Script

```
.....
if test $# -eq 0
then
    echo Usage: $(basename
      $0) STRING
    exit 1
fi
echo Received argument $1

$ examples_it244/usage_2.sh
Usage: usage_2.sh   STRING

$ examples_it244/usage_2.sh
foo
Received argument foo
```

- Here I used ***basename*** and **command substitution** to get the name of the file without the path
- You don't need a usage message if the script does not **require** arguments