Scripting Control Structures II

- The ***if ... then ... else ...*** Construct
- The ***if ... then ... elif ...*** Construct
- Debugging Scripts
- Indenting When Writing Scripts
- ***for ... in ...*** Loops
- ***for*** Loops
- Three-Expression ***for*** loops

The *if...then...else...* Construct

- Another Unix conditional is the ***if ... then ... else ...*** statement, which has the following format

```
if COMMAND
then
     COMMAND_1
     COMMAND_2
     ....
else
     COMMAND_A
     COMMAND_B
     ...
fi
```

The *if...then...else...* Construct

- If **COMMAND** returns an exit status of **0**
  - **COMMAND_1**, **COMMAND_2**, ... will be executed
- Otherwise, **COMMAND_A**, **COMMAND_B**, ... will be run
- Let's look at an example

```
$ cat cat_file.sh
#! /bin/bash
#
# demonstrates the use of the if ... then ... else ... construct

if [ $# -eq 0 ]
then
    echo Usage: $(basename $0) filename
    exit 1
fi
    ......
```

The *if...then...else...* Construct

```
....
if [ -f $1 ]
then
    cat $1
else
    echo $1 is not a file
fi

$ ./cat_file.sh
Usage: cat_file.sh filename

$ ./cat_file.sh lines.txt
line 1
line 2
line 3
line 4
line 5
```

```
$ ./cat_file.sh foo
foo is not a file
```

- The second ***if*** statement does one thing if the first argument *is* a file – and another thing if it *is not* a file
- The test command

  [-f $1]
- returns a status code of **0** if the first argument is the name of a file
- Otherwise, it returns **1**

The *if...then...elif...*Construct

- The ***if ... then ... elif ...*** construct lets you create *nested* conditionals
- ***elif*** stands for "else if"
- Notice that ***elif*** must be followed by ***then***
- The ***then*** must either be on the next line or on the same line, separated by a semi-colon **;**

```
if COMMAND
then
     COMMAND_1
     COMMAND_2
     ...
elif OTHER_COMMAND
then
     COMMAND_A
     COMMAND_B
     ...
...
else
     COMMAND_N1
     COMMAND_N2
     ...
fi
```

The *if...then...elif...*Construct

- While each ***else*** statement must be ended by a ***fi*** ...
- ... ***elif*** only requires a single ***fi*** at the end
- Let's look at an example

```
$ cat if_4.sh
#! /bin/bash
#
# demonstrates the if ... then ... elif ...  construction

echo -n "word 1: "
read word1
echo -n "word 2: "
read word2
echo -n "word 3: "
read word3
...
```

The *if...then...elif...*Construct

```
...
if [ $word1 = $word2 -a $word2 = $word3 ]
then
    echo "Match: words 1, 2 & 3"
elif [ $word1 = $word2 ]
then
    echo "Match: words 1 & 2"
elif [ $word1 = $word3 ]
then
    echo  "Match: words 1 & 3"
elif [ $word2 = $word3 ]
then
    echo  "Match: words 2 & 3"
else
    echo No match
fi
```

```
$ ./if_4.sh
word 1: foo
word 2: bar
word 3: bletch
No match

$ ./if_4.sh
word 1: foo
word 2: foo
word 3: boo
Match: words 1 & 2
```

Debugging Scripts

- It is easy to make a mistake when writing a script
- Thus, it is a good idea to practice *incremental development* when writing a script
- This means...
  - Writing a *few* lines of the script
  - Testing it and correcting errors
  - Writing *a few more* lines
- To help you find bugs in your Bash script, you can run ***bash*** with the `-x` option

Debugging Scripts

- The **-x** option causes ***bash*** to print each command *before* it executes the commands on that line

```
$ cat match_three.sh
#! /bin/bash
#
# takes three stings as
# input and compares them

if [ $# -lt 3 ]
then
    echo Usage: $(basename $0)  STRING1  STRING2  STRING3
    exit 1
fi
....
...

if [ $1 = $2 -a $2 = $3 ]
then
    echo All arguments match
elif [ $1 = $2 ]
then
    echo Arguments 1 and 2 match
elif [ $1 = $3 ]
then
    echo  Arguments 1 and 3 match
elif [ $2 = $3 ]
then
    echo  Arguments 2 and 3 match
else
    echo No arguments match
fi
```

Debugging Scripts

```
$ bash -x match_three.sh foo bar foo
+ '[' 3 -ne 3 ']'
+ '[' foo = bar -a bar = foo ']'
+ '[' foo = bar ']'
+ '[' foo = foo ']'
+ echo Arguments 1 and 3 match
Arguments 1 and 3 match
```

- Before ***bash*** prints a line from the script, it prints a plus sign **+** to let you know that the line is not output from the script
- Notice that the script keeps running ***test*** commands represented by **[ ]** until it finds one condition that evaluates to *true*

Debugging Scripts

- Using ***bash*** with the **`-x`** option, you can trace the path ***bash*** takes through your script
- This can help you find errors

Indenting When Writing Scripts

- Control structures work by marking off certain parts of the script that are executed *differently* from the rest of the script
- Most commands in a script are executed only *once* in the order they appear in the script
- In ***if*** constructions, certain blocks of commands are only executed under certain conditions
- In loop constructs, a block of commands is run more than once

Indenting When Writing Scripts

- Special Unix keywords set off these blocks of commands
- Keywords like ***`then`*** , ***`else`*** , ***`elif`*** , and ***`fi`***
- When writing scripts with control structures, it is a good idea to *indent* all lines in a block of commands so it is clear that they are treated *differently* from the rest of the script
- Let's look at the **`arg_test.sh`** script – which you wrote for Class Exercise 22...

Indenting When Writing Scripts

```
#! /bin/bash
#
# responds with the number of the arguments given
# to this script

if test $# -eq 0
then
    echo "You entered no arguments"
fi
if test $# -eq 1
then
    echo "You entered 1 argument"
fi
if test $# -eq 2
then
    echo "You entered 2 arguments"
fi
if test $# -gt 2
then
    echo "You entered more than 2 arguments"
fi
```

Indenting When Writing Scripts

- Each block of commands contained in each ***if ... then*** statement is clearly set off by the indent
- If we did not indent, we would get this →
- This is *much harder to read* than the indented version
- Indenting blocks of commands in a control structure is a good habit to get into

```
#! /bin/bash
#
# responds with the number of the
arguments given
# to this script

if test $# -eq 0
then
echo "You entered no arguments"
fi
if test $# -eq 1
then
echo "You entered 1 argument"
fi
if test $# -eq 2
then
echo "You entered 2 arguments"
fi
if test $# -gt 2
then
echo "You entered more than 2 arguments"
fi
```

for ... in ... Loops

- The most common programming construct, after the ***if*** statement, is the loop
- Looping can also be called *repetition*
- Bash provides many kinds of loops, but we'll start with the ***for ... in*** loop, which has the following format

```
for LOOP_VARIABLE  in  LIST_OF_VALUES
do
     COMMAND_1
     COMMAND_2
     ...
done
```

for ... in ... Loops

- ***do*** must be on a different line from ***for*** – unless you place a semicolon **;** before the ***do*** (just like ***then*** in an ***if*** statement)
- The commands between ***do*** and ***done*** are repeated *each time* through the loop
- In a ***for ... in*** loop, Bash
  - Assigns the first value in the **LIST_OF_VALUES** to the variable specified by **LOOP_VARIABLE**
  - Executes the commands between *do* and *done*
  - Assigns the next value in the **LIST_OF_VALUES** to the **LOOP_VARIABLE**
  - Executes the commands between *do* and the *done* again
  - And so on until each value in **LIST_OF_VALUES** has been used

for ... in ... Loops

- Here is an example

```
$ cat fruit.sh
#! /bin/bash
#
# demonstrates the for in loop

for fruit in apples oranges pears bananas
do
    echo $fruit
done
echo Task complete.

$ ./fruit.sh
apples
oranges
pears
bananas
Task complete.
```

for ... in ... Loops

- Notice that the variable **`fruit`** does not have a dollar sign in front of it when it appears after ***for***
- That's because here we are dealing with *the variable itself*, not its value
- We are telling Bash *which variable to use* when storing the values in the list
- The list of values can come from a number of different sources, including but not limited to, these:
  - A variable containing a list of values
  - Pathname expansion
  - Command substitution

for ... in ... Loops

- For example:

```
#!/bin/bash

#
# Performs a long listing of all files
# ending in .sh and then prints them
# and changes their permissions to 755
#

for file in *.sh
do
    ls -l $file
    echo
    cat $file
    chmod 755 $file
    echo
done
```

***for* Loops**

- The ***for*** loop is simpler than the ***for ... in ...*** loop
- and has the following format

```
for LOOP_VARIABLE
do
        COMMAND_1
        COMMAND_2
        ...
done
```

- The difference between the two ***for*** loops is *where* they get the values assigned to the loop variable
- The ***for ... in ...*** loop gets values from the list that follows ***in***

***for* Loops**

- These values are "**hard coded**" into the script
- They never change
- The plain ***for*** loop gets its values from *the command line*
- The plain ***for*** loop can have different values each time it is run
- Here is an example…

```
$ cat for_test.sh
#! /bin/bash
#
# demonstrates the simple for loop

for arg
do
    echo $arg
done

$ ./for_test.sh foo bar bletch
foo
bar
bletch

$ ./for_test.sh bing bang boom
bing
bang
boom
```

Three-Expression *for* Loops

- The ***for*** loops above are very different from the for loops in programming languages
- In programming languages, the ***for*** statement
  - *Initializes* a loop variable
  - *Tests* the value of the loop variable to decide whether to run the loop one more time
  - *Changes* the loop variable at the end of the loop code
- ***for*** statements in programming languages **create** the values used in the loop

Three-Expression *for* Loops

- But, the ***for*** loops above **must be *given*** the values used in the loop
- In the ***for ... in ...*** statement, the values come after ***in*** in the script itself
- In the plain ***for*** statement, the values are given at the command line
- But, there is a third form of ***for*** loop in Bash
- This form *creates* the values for the loop variable – the same way as the for loop in programming languages

Three-Expression *for* Loops

- It has the following form

```
for (( EXP1; EXP2; EXP3 ))
do
    COMMAND_1
    COMMAND_2
    ...
done
```

- Notice that the three expressions are inside ***double*** **parentheses**
- That means that anything inside will be treated as *numbers* not text

Three-Expression *for* Loops

- The three expressions are the *loop control*:
- The *first* expression sets the value of the loop variable
- The *second* is a logical expression. As long as it is **true**, the loop will continue
- The *third* expression changes the value of the loop variable **after** each pass through of the loop
- Let's look at an example. The key is the variable `count`

Three-Expression *for* Loops

```
$ cat count_to_five.sh
#! /bin/bash
#
# this script demonstrates the
# three expression for loop

for (( count=1; count<=5; count++ ))
do
    echo $count
done

$ ./count_to_five.sh
1
2
3
4
5
```

- The expression `count++` increases the value of `count` by one
- Without this third expression, the loop would never end, and we would have an *infinite* loop