Scripting Control Structures III

- ***while*** Loops
- ***until*** Loops
- ***continue***
- ***break***
- ***case*** Statement
- ***select*** Statement

***while* Loops**

- The ***for*** loops we saw earlier keep running until all values supplied to them have been used in the loop
- A ***while*** loop will continue running as long as the test command returns a status code of **0**
- ***while*** loops have the form

```
while COMMAND
do
     COMMAND_1
     COMMAND_2
     ...
done
```

***while* Loops**

- As long as the COMMAND returns a status code of ***0***, the commands between ***do*** and ***done*** will be run
- Let's look at an example
- The line after ***echo -n*** (the one with the double parentheses) tells Bash to interpret the expression as a *number*, not a string

```
$ cat count_to_nine.sh
#! /bin/bash
#
# counts to 9, then stops

number=0
while [ $number -lt 10 ]
do
     echo -n $number
     (( number += 1))
done

$ ./count_to_nine.sh
0123456789
```

***while* Loops**

- The **+=** operator tells Bash to add the number that follows (in this case, **1**) to the current value of **number**

```
(( number += 1))
```

- The ***echo*** command in the ***while*** loop uses the ***-n*** option and does not print a newline

```
echo -n $number
```

- This allows the script to print the numbers one right after the other, all on the same line
- To end the output and get a new line, we need the final ***echo*** command

while Loops

- The condition that the ***while*** loop tests must change, or the ***while*** loop will run forever
- The value of **number** never changes, so the expression given to ***test*** never becomes false, and the script loops forever (until it is aborted)

```
$ cat forever.sh
#! /bin/bash
#
# this loop runs forever

number=0
while [ $number -lt 10 ]
do
  echo  $number
done

$ ./forever.sh
0
0
0
0
^C
```

until Loops

- The ***until*** loop is similar the ***while*** loop, except that the ***until*** loop ends when the test condition becomes **true**
- Whereas the ***while*** loop stops when the text condition becomes **false**
- The ***until*** loop has the form

```
until COMMAND
do
     COMMAND_1
     COMMAND_2
     ...
done
```

until Loops

- Here is an example:

```
$ cat count_until.sh
#! /bin/bash
#
# counts from 1 to its argument, then stops

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

- ***while*** loops are used much more often than ***until*** loops

```
...
number=1
until [ $number -gt $1 ]
do
  echo  $number
  (( number += 1))
done

$ ./count_until.sh 6
1
2
3
4
5
6

$
```

continue

- Normally, a loop will run through all the commands between ***do*** and ***done*** for *each* pass through the loop
- Sometimes, however, you want to skip all or part of the loop commands – for a specific pass through the loop
- Let's say you are calculating interest for a group of savings accounts
- In doing this, you perform a series of operations on all savings accounts
- But, if you get to an account that has been closed, you don't want to perform these operations

continue

- In other words, you want to stop working on a *specific* account – but continue looping through the other accounts
- It is for situations like this that ***continue*** was created
- When the shell comes to ***continue*** inside a loop, it stops running the loop code and jumps to the *top* of the loop to begin another pass through the loop
- Let's look at an example….

continue

```
$ cat continue.sh
#! /bin/bash
#
# demonstrates how continue works

total=0
for number in 1 2 3 4 5
do
    if [ $number -eq 2 -o $number -eq 4  ]
    then
        continue
    fi
    echo Adding $number to $total
    (( total += $number ))
done
echo
echo total: $total
```

```
$ ./continue.sh
Adding 1 to 0
Adding 3 to 1
Adding 5 to 4

total: 9
```

- Whenever the variable **number** is 2 or 4, execution of the rest of the code stops, and the next pass through the loop begins
- ***continue*** does not cause the script to break out of the loop; it merely stops execution of the loop code *for **one** iteration*

break

- Every time you start a loop, you specify what will cause the loop to end
- With ***for ... in*** and simple ***for*** loops, the code exits the loop when every value in the argument list has been used
- In the ***while***, ***until***, and three-expression ***for*** loops, the code exits the loop *when a logical condition is met*
- In all cases, the terminating condition is specified at the *top* of the loop

break

- But, what if you encountered some unusual condition and wanted to break out of the loop *entirely*?
- To do this, you would have to use ***break***
- When ***bash*** comes across the ***break*** in the code inside a loop, it jumps out of the loop *completely* – and proceeds with the commands *following* the loop
- Let's look at an example...

break

```
$ cat break.sh
#! /bin/bash
#
# demonstrates how break works

for filename in *
do
   if  [ -x $filename ]
   then
        echo First executable file: $filename
        break
   fi
done

$ ./break.sh
First executable file: bother.sh
```

- This script looks at *each* file in the current directory
- When it finds a file that is executable, it prints that filename and *stops*
- The loop will end either
  - When it *finds* an executable file
  - Or when it has examined *every* file and found *no* executable

break

- Notice how we got the values for the variable **`filename`** in the ***`for ... in`*** loop.
- We used **`*`**
- When the script is run, the value of **`*`** on this line of the script is replaced with the name of ***every*** *file or directory* in the current directory

case Statement

- Sometimes a script needs to take a specific path, depending on the value of a **single** variable
- You could do this with an ***if*** ... ***then*** ... ***elif*** ... statement
- But, there is a simpler structure for such situations – the ***case*** statement – which has the following format...

```
case TEST_VARIABLE in
    PATTERN_1)
        COMMAND_1A
        COMMAND_1B
        COMMAND_1C
        ...
        ;;
    PATTERN_2)
        COMMAND_2A
        COMMAND_2B
        COMMAND_2C
        ...
        ;;
    PATTERN_3)
        COMMAND_3A
        COMMAND_3B
        COMMAND_3C
        ...
        ;;
    ...
esac
```

***case* Statement**

- When Bash encounters a ***case*** statement, it
  - Finds the *first* pattern that matches the test variable
  - *Runs the statements for that pattern*
  - Leaves the *case* statement
- Notice
  - There is a right parenthesis **)** after each pattern
  - The statements for each pattern end with two semi-colons **;;**
  - *esac* marks the end of the *case* statement
- ***esac*** is ***case*** spelled backwards
- Let's look at an example...

case Statement

```
$ cat case_1.sh
#! /bin/bash
#
# demonstrates how the case statement
works

echo -n "Enter A, B, or C: "
read letter
case $letter in
    A)
        echo You entered A
        ;;
    B)
        echo You entered B
        ;;
    C)
        echo You entered C
        ;;
    *)
        echo You did not enter A, B, or C
        ;;
esac
echo Exiting program
```

```
$ ./case_1.sh
Enter A, B, or C: A
You entered A
Exiting program

$ ./case_1.sh
Enter A, B, or C: B
You entered B
Exiting program

$ ./case_1.sh
Enter A, B, or C: d
You did not enter A, B, or C
Exiting program
```

- Notice the last pattern *****
- This pattern is a *catchall* that will match *any* input

`case` Statement

- You should use this as the *final pattern* in a **`case`** statement
- This pattern will match *anything that has not matched a previous pattern*
  - The code for this pattern should print an *error message* because the value of the variable was not expected
  - You must put the **`*`** at the *end* of the pattern list because Bash will never see any patterns that follow it since **`*`** matches everything
  - If you don't use the asterisk, and a matching pattern is not found, Bash will simply execute the code following **`esac`**

`case` Statement

- When creating patterns, you can use the *metacharacters* and the *logical OR*

*	Matches any string of characters
?	Matches any single character
[ ]	Every character within the brackets can match a single character in the test string
|	Logical OR separates alternative patterns

- The last symbol is a vertical line | which is the symbol for a logical OR
- This symbol allows us to put *many* possible matches on the same line
- We can use | to accept letters of either case

case Statement

```
$ cat case_2.sh
#! /bin/bash
#
# demonstrates the use of the | (logical or)
# operator in patterns within a case statement

echo -n "Enter A, B, or C: "
read letter
case $letter in
    a|A)
        echo You entered A
        ;;
    b|B)
        echo You entered B
        ;;
    c|C)
        echo You entered C
        ;;
    *)
        echo You did not enter A, B, or C
    ;;
esac
echo Exiting program
```

```
$ ./case_2.sh
Enter A, B, or C: A
You entered A
Exiting program

$ ./case_2.sh
Enter A, B, or C: a
You entered A
Exiting program
```

***select* Statement**

- The ***select*** statement is used to create a menu inside a shell script
- It needs a list of values, which it turns into *numbered menu choices*
- When the user enters a number, the variable with that number is assigned to a loop variable

- A ***select*** statement has following form

```
select LOOP_VARIABLE [in LIST_OF_VALUES]
do
    COMMAND_1
    COMMAND_2
    COMMAND_3
    ...
done
```

  - Notice that

    ```
    in LIST_OF_VALUES
    ```

  - is *optional*

***select* Statement**

- The ***select*** statement needs a list of values
  - These values can be **hard coded** into the script following the ***in*** keyword
  - Or, they can be supplied as *arguments* at the command line
- In other words, the loop variable can get its values the same way a ***for*** loop can

- When Bash comes upon a ***select*** statement, it
  - Prints a *menu* on the screen
  - Creates *menu items* for each of the values, assigning each value a number
  - Prints a *prompt* asking the user for input
  - *Reads* a number from user input
  - *Assigns the value* for that number to the select variable
  - *Runs the statements* between *do* and *done* with that value
  - Prints *another* prompt

***select* Statement**

- The ***select*** statement is a *loop* construct that will run *forever*, unless you do something to stop it
- Let's look at an example:

```
$ cat select_1.sh
#! /bin/bash
#
# demonstrates how the select statement works

PS3="Choose your fruit: "
select fruit in apple banana blueberry orange
do
    echo You chose $fruit
    echo That is choice number $REPLY
done
```

- Here, we have *hard coded* the values into the script itself

select Statement

```
$ ./select_1.sh
1) apple
2) banana
3) blueberry
4) orange
Choose your fruit: 1
You chose apple
That is choice number 1
Choose your fruit: 2
You chose banana
That is choice number 2
...
```

```
...
Choose your fruit: 3
You chose blueberry
That is choice number 3
Choose your fruit: 4
You chose orange
That is choice number 4
Choose your fruit: ^C
```

select Statement

- We can also supply the values *from the command line*

```
$ cat select_2.sh
#! /bin/bash
#
# demonstrates how the select structure works
# taking argument from the command line

PS3="Choose your fruit: "
select fruit
do
    echo You chose $fruit
    echo That is choice number $REPLY
done
```

select Statement

```
$ ./select_2.sh peaches pears watermelons
1) peaches
2) pears
3) watermelons

Choose your fruit: 1
You chose peaches
That is choice number 1

Choose your fruit: 2
You chose pears
That is choice number 2

Choose your fruit: 3
You chose watermelons
That is choice number 3

Choose your fruit: ^C
```

- The only difference between these two scripts is that
  - the first has hard coded values …
  - while the second takes the values from the command line
- In both scripts, Bash assigned a value to the select variable **`fruit`** based on the number *chosen by the user*

***select* Statement**

- The ***select*** statement uses a number of **keyword shell variables**
- The variable **PS3** contains a string that the shell will use to prompt for input
- If we had not given **PS3** a value, the default value **#?** would be used
- Let's look at an example...

select Statement

```
$ cat select_3.sh
#! /bin/bash
#
# demonstrates the select statement where
# PS3 has the default value

select fruit in apple banana blueberry orange
do
    echo You chose $fruit
    echo That is choice number $REPLY
done

$ ./select_3.sh
1) apple
2) banana
3) blueberry
4) orange
#? 3
You chose blueberry
That is choice number 3
#? 4
You chose orange
That is choice number 4
#? ^C
```

- Another keyword shell variable used by ***select*** is **REPLY**
- When the user enters a number at the keyboard, the value associated with that number is assigned to the loop variable, but the **number** entered is assigned to **REPLY**

select Statement

- Unless you include a menu choice to jump out of the loop, the loop will go on forever
- The list of values should include something that can be used to *break out of* the loop

```
$ cat select_4.sh
#! /bin/bash
#
# demonstrates a select menu with a stop value

PS3="Choose your fruit: "
select fruit in apple banana blueberry orange STOP
do
    if [ $fruit = STOP ]
    then
        echo About to leave
        break
    fi
    echo You chose $fruit
    echo That is choice number $REPLY
done
echo Exiting program
```

select Statement

```
$ ./select_4.sh
1) apple
2) banana
3) blueberry
4) orange
5) STOP
Choose your fruit: 2
You chose banana
That is choice number 2
Choose your fruit: 5
About to leave
Exiting program
```