Chapter 11 Notes

I. Creating a Script File
   A. When creating a shell script file, you must specify the shell you are using in the first line of the file:
      1. #!/bin/bash
         a) In a normal shell script line, the pound sign (#) is used as a comment line
   B. You can use the semicolon and put both commands on the same line if you want to, but in a shell script, you can list commands on separate lines.
   C. The PATH environment variable is set to look for commands only in a handful of directories. To get the shell to find the script, we need to do one of two things:
      1. Add the directory where our shell script file is located to the PATH environment variable
      2. Use an absolute or relative file path to reference our shell script file in the prompt
         a) Some Linux distributions add the $HOME/bin directory to the PATH environment variable. This creates a place in every user's HOME directory to place files where the shell can find them to execute
   D. Ensure that the file has execute permissions for the owner

II. Displaying Messages
   A. The echo command can display a simple text string if you add the string following the command:
      1.
```
$ echo This is a test
This is a test
$
```
   B. The echo command uses either double or single quotes to delineate text strings. If you use them within your string, you need to use one type of quote within the text and the other type to delineate the string:
      1.
```
$ echo "This is a test to see if you're paying attention"
This is a test to see if you're paying attention
$ echo 'Rich says "scripting is easy".'
Rich says "scripting is easy".
$
```

C. Use the -n parameter if you want to echo a text string on the same line as a command output:

    1. echo -n "The time and date are: "

III. Using Variables

A. Variables allow you to temporarily store information within the shell script for use with other commands in the script

B. Environment variables

    1. The shell maintains environment variables that track specific system information, such as the name of the system, the name of the user logged in to the system, the user's system ID (called UID), the default home directory of the user, and the search path used by the shell to find programs

        a) You can display a complete list of active environment variables available by using the set command:

    2. You can tap into these environment variables from within your scripts by using the environment variable's name preceded by a dollar sign

        a)
```
$ cat test2
#!/bin/bash
# display user information from the system.
echo "User info for userid: $USER"
echo UID: $UID
echo HOME: $HOME
$
```

            (1) The $USER, $UID, and $HOME environment variables are used to display the pertinent information about the logged-in user

        b) Whenever the script sees a dollar sign within quotes, it assumes you're referencing a variable

            (1) To display an actual dollar sign, you must precede it with a backslash character:

                (a)
```
$ echo "The cost of the item is \$15"
The cost of the item is $15
```

C. User variables

1. Setting variables allows you to temporarily store data and use it throughout the script, making the shell script more like a real computer program
    a) User variables can be any text string of up to 20 letters, digits, or an underscore character
    b) User variables are case sensitive, so the variable Var1 is different from the variable var1
2. Values are assigned to user variables using an equal sign
    a) No spaces can appear between the variable, the equal sign, and the value
        (1)
        ```
        var1=10
        var2=-57
        var3=testing
        var4="still more testing"
        ```
3. The shell script automatically determines the data type used for the variable value
4. Variables defined within the shell script maintain their values throughout the life of the shell script but are deleted when the shell script completes
5. Just like system variables, user variables can be referenced using the dollar sign
6. When referencing a variable value you use the dollar sign, but when referencing the variable to assign a value to it, you do not use the dollar sign:
    a)
    ```
    $ cat test4
    #!/bin/bash
    # assigning a variable value to another variable
    value1=10
    value2=$value1
    echo The resulting value is $value2
    $
    ```
    b) Output:
        (1)
        ```
        $ chmod u+x test4
        $ ./test4The resulting value is 10
        ```

```
$
```

D. Command substitution

1. There are two ways to assign the output of a command to a variable:
   a) The backtick character (`)
      (1) testing=`date`
   b) The $() format
      (1) testing=$(date)
         (a)
```
#!/bin/bash
# copy the /usr/bin directory listing to a log file
today=$(date +%y%m%d)
ls /usr/bin -al > log.$today
```

E. Redirecting Input and Output

1. The bash shell provides a few different operators that allow you to redirect the output of a command to an alternative location (such as a file)
2. Output redirection
   a) The most basic type of redirection is sending output from a command to a file (output redirection). The bash shell uses the greater-than symbol (>) for this:
      (1)
```
$ date > test6
$ ls -l test6
-rw-r--r--      1 user  user           29 Feb 10 17:56 test6
$ cat test6Thu Feb 10 17:56:58 EDT 2014
$
```
   b) Anything that would appear on the monitor from the command instead is stored in the output file specified
   c) Sometimes, instead of overwriting the file's contents, you may need to append output from a command to an existing file. In this situation, you can use the double greater-than symbol (>>) to append data:
      (1)
```
$ date >> test6
$ cat test6
user    pts/0    Feb 10 17:55
Thu Feb 10 18:02:14 EDT 2014
```

3. Input redirection
    a) Input redirection takes the content of a file and redirects it to a command. The input redirection symbol is the less-than symbol (<):
        (1)
```
$ wc < test6
        2       11      60
$
```
            (a) The wc command provides a count of text in the data. By default, it produces three values:
                (i) The number of lines in the text
                (ii) The number of words in the text
                (iii) The number of bytes in the text
        (2) Another method of input redirection is called inline input redirection. This method allows you to specify the data for input redirection on the command line instead of in a file. The inline input redirection symbol is the double less-than symbol (<<). Besides this symbol, you must specify a text marker that delineates the beginning and end of the data used for input. You can use any string value for the text marker, but it must be the same at the beginning of the data and the end of the data:
            (a)
```
command << marker
data
marker
```
                (b)
```
$ wc << EOF
> test string 1
> test string 2
> test string 3
> EOF
        3       9       42
$
```

F. Pipes

1. Instead of redirecting the output of a command to a file, you can redirect the output to another command. This process is called piping. The pipe (|) is put between the commands to redirect the output from one to the other:
   a) command1 | command2
   b) $ rpm -qa | sort | more
      (1) This command sequence runs the rpm command, pipes the output to the sort command, and then pipes that output to the more command to display the data, stopping after every screen of information

G. Performing Math

1. The bash shell mathematical operators support only integer arithmetic. This is a huge limitation if you're trying to do any sort of real-world mathematical calculations
2. The expr command
   a) Originally, the Bourne shell provided a special command that was used for processing math-ematical equations. The expr command allowed the processing of equations from the com-mand line, but it is extremely clunky:
      (1) $ expr 1 + 5
      6
   b) The expr command recognizes a few different mathematical and string operators:
      (1) **ARG1 | ARG2**: Returns ARG1 if neither argument is null or zero; otherwise, returns ARG2
      (2) **ARG1 & ARG2**: Returns ARG1 if neither argument is null or zero; otherwise, returns 0
      (3) **ARG1 < ARG2**: Returns 1 if ARG1 is less than ARG2; otherwise, returns 0
      (4) **ARG1 <= ARG2**: Returns 1 if ARG1 is less than or equal to ARG2; otherwise, returns 0
      (5) **ARG1 = ARG2**: Returns 1 if ARG1 is equal to ARG2; otherwise, returns 0

(6) **ARG1 != ARG2**: Returns 1 if ARG1 is not equal to ARG2; otherwise, returns 0

(7) **ARG1 >= ARG2**: Returns 1 if ARG1 is greater than or equal to ARG2; otherwise, returns 0

(8) **ARG1 > ARG2**: Returns 1 if ARG1 is greater than ARG2; otherwise, returns 0

(9) **ARG1 + ARG2**: Returns the arithmetic sum of ARG1 and ARG2

(10) **ARG1 - ARG2**: Returns the arithmetic difference of ARG1 and ARG2

(11) **ARG1 * ARG2**: Returns the arithmetic product of ARG1 and ARG2

(12) **ARG1 / ARG2**: Returns the arithmetic quotient of ARG1 divided by ARG2

(13) **ARG1 % ARG2**: Returns the arithmetic remainder of ARG1 divided by ARG2

(14) **STRING : REGEXP**: Returns the pattern match if REGEXP matches a pattern in STRING

(15) **match STRING REGEXP**: Returns the pattern match if REGEXP matches a pattern in STRING

(16) **substr STRING POS LENGTH**: Returns the substring LENGTH characters in length, starting at position POS (starting at 1)

(17) **index STRING CHARS**: Returns position in STRING where CHARS is found; otherwise, returns 0

(18) **length STRING**: Returns the numeric length of the string STRING

(19) **+ TOKEN**: Interprets TOKEN as a string, even if it's a keyword

(20) **(EXPRESSION)**: Returns the value of EXPRESSION

3. Many of the expr command operators have other meanings in the shell (such as the asterisk). Using them in the expr command produces odd results:
   a) $ expr 5 * 2

```
expr: syntax error
$
```

b) To solve this problem, you need to use the shell escape character (the backslash) to identify any characters that may be misinterpreted by the shell before being passed to the expr command:

(1)
```
$ expr 5 \* 2
10
$
```

H. Using brackets

1. In bash, when assigning a mathematical value to a variable, you can enclose the mathematical equation using a dollar sign and square brackets ($[operation]):

   a)
   ```
   $ var1=$[1 + 5]
   $ echo $var1
   6
   $ var2=$[$var1 * 2]
   $ echo $var2
   12
   $
   ```

2. This same technique also works in shell scripts

IV. A floating-point solution

A. The basics of bc

1. The bash calculator (bc) is actually a programming language that allows you to enter floating-point expressions at a command line and then interprets the expressions, calculates them, and returns the result. The bash calculator recognizes these:

   a) Numbers (both integer and floating point)

   b) Variables (both simple variables and arrays)

   c) Comments (lines starting with a pound sign or the C language /* */ pair)

   d) Expressions

   e) Programming statements (such as if-then statements)

   f) Functions

2. To exit the bash calculator, you must enter quit
3. The floating-point arithmetic is controlled by a built-in variable called scale. You must set this value to the desired number of decimal places you want in your answers, or you won't get what you were looking for:
    a)
    ```
    $ bc -q
    3.44 / 5
    0
    scale=4
    3.44 / 5
    .6880
    quit
    $
    ```
        (1) The default value for the scale variable is zero
        (2) The -q command line parameter suppresses the lengthy welcome banner from the bash calculator
4. The print statement allows you to print variables and numbers

B. Using bc in scripts

1. You can use the command substitution character to run a bc command and assign the output to a variable:
    a) variable=$(echo "options; expression" | bc)
        (1) The first portion, options, allows you to set variables. If you need to set more than one variable, separate them using the semicolon. The expression parameter defines the math-ematical expression to evaluate using bc
        (2)
        ```
        $ cat test9
        #!/bin/bash
        var1=$(echo "scale=4; 3.44 / 5" | bc)
        echo The answer is $var1
        $
        ```
        (3) Output:
            (a)
            ```
            $ chmod u+x test9
            $ ./test9
            The answer is .6880
            $
            ```

2. The bc command recognizes input redirection, allowing you to redirect a file to the bc command for processing:
    a) variable=$(bc << EOF
```
options
statements
expressions
EOF
)
```
        b)
```
$ cat test12
#!/bin/bash
var1=10.46
var2=43.67
var3=33.2
var4=71
var5=$(bc << EOF
scale = 4
a1 = ( $var1 * $var2)
b1 = ($var3 * $var4)
a1 + b1
EOF
)

echo The final answer for this mess is $var5
$
```

V. Exiting the Script

A. Every command that runs in the shell uses an exit status to indicate to the shell that it's finished processing. The exit status is an integer value between 0 and 255 that's passed by the command to the shell when the command finishes running. You can capture this value and use it in your scripts

B. Checking the exit status
1. Linux provides the $? special variable that holds the exit status value from the last command that executed.
    a) You must view or use the $? variable immediately after the command you want to check

b) $ date

```
Sat Jan 15 10:01:30 EDT 2014
$ echo $?
0
$
```

2. By convention, the exit status of a command that successfully completes is zero
3. If a command completes with an error, then a positive integer value is placed in the exit status
4. Linux Exit Status Codes
   a) **0**: Successful completion of the command
   b) **1**: General unknown error
      (1) Will show if you supply an invalid parameter to a command
   c) **2**: Misuse of shell command
   d) **126**: The command can't execute
      (1) Indicates that the user didn't have the proper permissions set to execute the command
   e) **127**: Command not found
   f) **128**: Invalid exit argument
   g) **128+x**: Fatal error with Linux signal x
   h) **130**: Command terminated with Ctrl+C
   i) **255**: Exit status out of range

C. The exit command

1. The exit command allows you to specify an exit status when your script ends:
   a) $ cat test13

```
#!/bin/bash
# testing the exit status
var1=10
var2=30
var3=$[$var1 + $var2]
echo The answer is $var3
exit 5
```

```
$
```

b) Output:

(1)
```
$ chmod u+x test13
$ ./test13
The answer is 40
$ echo $?
5
$
```

2. You can also use variables in the exit command parameter
3. You should be careful with this feature, however, because the exit status codes can only go up to 255
    a) If an exit status code is out of range, it is reduced to fit in the 0 to 255 range
        (1) The shell does this by using modulo arithmetic. The modulo of a value is the remainder after a division. The resulting number is the remainder of the specified number divided by 256