Imperative Problem Solving and Data Structures

Bash Scripts

Goals

In this laboratory you will gain experience writing Bash scripts.

Prerequisites

Basic knowledge of Bash shell commands.

Acknowledgement

This laboratory exercise was written by Marge Coahran, with editing by Henry M. Walker for consistency in formatting for CSC 161.

Index

Part A: Getting Started

A bash script is a file containing bash commands. To make the file executable, we must first take the following two steps. First, the script file should contain the following code as its very first line.

```
#!/bin/bash
```

This specifies which program (bash) should be used to interpret the commands in the script. Next, the file permissions for the script should be set such that is it executable. Recall that this is typically done as follows.

```
chmod 755 scriptname
```

The script can then be run like any other executable file:

```
./scriptname
```

Actually, we can also run a script as follows (even without making the script executable), but this is less frequently done.

```
bash scriptname
```

1. Please take a few minutes to review the bash commands, and the information on file permissions, that you learned in the GNU/Linux labs we did earlier in the semester.

2. Write a bash script called `greeting` that prints a greeting to the user -- perhaps "Good morning" or something similar. Remember to insert the proper incantation on the first line of the script, and to set the file permissions to allow execution.

Note that commands in the script file do not need to end with semi-colons. (They are bash commands, not C statements.)

3. Now modify your script `greeting` so that it

    - greets the user by (user)name,
    - prints the current date, and
    - prints a list of users currently logged onto the computer.

    For example, your output might look similar to the following:

```
Good morning, mcoahran.
Date is Sun Apr 20 12:22:47 CDT 2008
Users on Leah:
mcoahran :0           2008-04-20 11:45
mcoahran pts/0        2008-04-20 11:45 (:0.0)
mcoahran pts/1        2008-04-20 11:49 (:0.0)
```

    Be sure to add comments to your script! Comments in bash scrpts begin with #, except on the very first line.

Part B: Control Structures: IF

Variables and Arithmetic Expressions

To complete this exercise you will probably need to use variables and arithmetic comparisons.

Variables can be created and initialized as shown below. Note that there should NOT be a space between the variable name and the assignment operator. (If the variable name has a space after it, bash will recognize it as a token, but since the variable does not yet exist, bash will assume the token is command name. It will then complain about not finding such a command.)

```
varname=12
```

To access the value in a variable, prepend a $ to the variable name. For example,

```
echo Value in varname is $varname.
```

Note that variables in a bash script are by default interpreted as text. To get bash to interpret a value as a number, in order to do arithmetic operations including arithmetic comparisons, you must wrap the expression in double-parentheses. For example, you might say:

```
if [ hour < 7 ]; then
```

The arithmetic comparison operators are the same in bash as in C:

```
==  !=  <  <=  >  >=  &&  ||
```

Bash's other arithmetic operators are also reminiscent of C, with the addition of an operator for exponentiation (`**`).

```
+  -  ++  --  *  /  %  **
```

If statements

Bash scripts also provide for a variety of program control structures, including conditionals, loops, and functions.

The syntax for an if-statement follows. As you would probably expect, the `elif` and `else` clauses are optional, and the conditions and commands should be replaced with meaningful expressions. Note that the spaces that separate the square brackets from the conditions ARE required.

```
if [ condition ]; then
  command(s)
elif [ condition ]; then
  command(s)
else
```

```
  command(s)
fi
```

4. Modify your script `greeting` to present a different greeting based on the time of day. For example, your greeting could be "good morning", "good afternoon", "good evening", or "YOU SHOULD BE SLEEPING!" based on the current hour. Note that the following command can be used to return the (24-hour) hour of the current time:

   ```
   date +%k
   ```

Part C: Control Structures: WHILE

The syntax for a while loop follows.

```
while test
do
  command(s)
done
```

5. Write a script called `countdown` that prints output similar to the following:

   ```
   10
   9
   8
   7
   6
   5
   4
   3
   2
   1
   GO!
   ```

   Don't forget that bash uses the same arithmetic operators as C, but arithmetic expressions need to be wrapped in double-parentheses.

Part D: Getting User Input

Getting Input From stdin

The command `"read varname"` will read a value from stdin and assign it to a variable called `varname`. If the variable does not yet exist, it will be created.

Note that the option `"-n"` can be used with `echo` to cause it to not output a newline character. (This is nice when printing user prompts, so the user can enter a reponse on the same line as the prompt).

Getting Input From Command-line Arguments

The mechanism for accessing command-line arguments in a bash script differs substantially from C. In bash, there are several variables that are automatically defined and loaded with information from the command-line as shown below.

variable	stores...
`$#`	number of arguments given
`$*`	list of all arguments given
`$1`	the first argument
`$2`	the second argument
`etc`	

For example, you might say:

```
if [ $# == 3 ]
```

Note that, unlike in C, the script name itself is NOT counted or included in the list of arguments given.

6. Modify `countdown` such that it prompts user for a starting value, and counts down from there.

7. Write a script called `countdown2` that accepts the initial value as a command-line argument. For example, the command and its output might look like the following.

```
$ ./countdown2 12
12
11
10
9
8
7
6
5
4
3
2
1
GO!
```

8. Modify the script `countdown2` to check for correct usage. The script should print a usage message and exit if it does not receive exactly one argument. An example session might look like this:

```
$  ./countdown2
  Usage: countdown2 initial-value
```

   Note that you can exit a bash script with:

```
  exit returnvalue
```

Part E: Control Structures: FOR

The syntax for a for-loop follows.

```
for varname in value1 value2 ...
do
  command(s)
done
```

The variable `varname` will take on each value in the list of values in turn, with one iteration of the loop occuring for each value.

9. For example, consider the following script. Predict what it will do, and then copy it and run it to confirm your prediction.

   ```
   #!/bin/bash

   for var in 1 2 3 4 5
   do
     echo $var chimpanzee...
   done
   echo DONE!
   ```

10. Recall that your bash script is run by the very same bash that responds to your commands in the terminal window. Thus, all functionality available in the terminal window is also available to your script.

    For example, we could write a for-loop that uses filename expansion (globbing) to print a list of all files in the current directory as follows:

    ```
    for file in *
    do
      echo $file
    done
    ```

11. Write a script that uses this idea to print output similar to the following:

    ```
    Files in this directory that match *~ :
    bash-scripts.html~
    ```

```
chimpanzee~
countdown~
countdown2~
hello~
TODO~
```

Part F: Some Useful Scripts

Conditionals Regarding Files and Directories

Bash scripts are frequently written for tasks involving the creation and maintenance of files and directories because it can be much faster and easier to write a script for these tasks than to write the corresponding C programs.

There are many operators that can be used for testing conditions that involve files: whether they exist, what kind of file they are, etc. Here are a few. Note that you could also place the NOT operator (`!`) before any of these tests to test whether the stated condition is false.

condition	checks whether...
`-f` *`file`*	file exists and is a regular file
`-d` *`dir`*	dir exists and is a directory
`-x` *`dir`*	file exists and is executable

For example, you might write:

```
clist=classlist.txt
if [ -f $clist ]; then
  echo File $clist exists.
else
  echo File $clist does not exist.
fi
```

12. For this exercise I wanted to give you some real-world examples of useful bash scripts. Therefore, you will write a couple of scripts that I actually use in my teaching to automate clerical tasks. To begin, please copy this (ficitious) classlist to your directory, and note that it contains a list of usernames.

    Write a script called `addnames` that is to be called as follows, where *`classlist`* is the name of the classlist file, and *`username`* is a particular student's username.

    ```
    ./addnames classlist username
    ```

    The script should

    - check that the correct number of arguments was received and print an usage message if not,
    - check whether the classlist file exists and print an error message if not,
    - check whether the username is already in the file, and then either
        - print a message stating that the name already existed, or
        - add the name to the end of the list.

    Hint: Use a for-loop to process each line in the file. To create a list of lines in the file, remember that you can use any bash construct inside a script that you can use in the terminal window. How would you list the lines of the file in the terminal window? Similarly, how would you append a particular item to the end of an existing file from the terminal window?

    If you have not been doing so all along, you should now add some comments to your script and thoroughly test it!

13. Write a script called `submit-dirs` that is to be called as follows.

    ```
    ./submit-dirs classlist
    ```

    The script should:

    - check whether the correct number of arguments was received and print a usage message if not,

- check whether the classlist file exists and print an error message if not,
- create a directory named `submit` within the current directory (but only if one does not already exist),
- create a directory within the directory `submit` for each student in the class (but only if one does not already exist). These student directories should be named according to the students' usernames.

14. Write a script called `trash` that takes a single argument, which should be the name of an existing file. The script should move the given file, if it exists, to a directory named `trash` that is located within your home directory. If the `trash` directory does not exist, the script should create it. If the given file does not exist, an appropriate error message should be printed.

    Hint: There are several "environment" variables that automaticaly exist within a bash script. These include the following. You may want to explore what values these hold by echoing them to stdout.

    - $HOME
    - $USER
    - $PWD
    - $PATH

15. Once your `trash` script is working well, modify it so that it accepts a list of files and moves all of them to the trash. The script should print an error message for each file that does not exist, and it should print a usage message if no file names are given.

    When testing your script, try invoking it with a command like the following:

    ```
    ./trash *.o
    ```

16. If you would like to, you could begin a directory of bash scripts you would like to use regularly (such as `trash`). You can then add the name of that directory to your $PATH by following the instructions given below. Doing so will allow the scripts therein to be available to you regardless of which directory you are currently working in.

    To add a directory to your path, look for a file named `.bash_profile` in your home directory. It is a bash script

that is run automatically when you log in. It should contain a line that defines the variable PATH. Note that the value assigned to this variable is a list of directories, delimited by colons. You can add your new script directory to your path by adding it to the *end* of this list. (That will cause bash to check your scripts directory after checking the others when search for programs to run.)

After modifying your `.bash_profile`, you should also run `./bash_profile` with the following command to update the contents of your current path variable.

```
source ~/.bash_profile
```

Work to be Turned in

- Bash scripts for steps 2-4 (may be one script), 5, 6, 7-8 (may be one script), 11, 12, 13, 14, 15
- *Extra Credit:* notes on work for step 16 (e.g., listing of revisions made to `.bashrc`

This document is available on the World Wide Web as

```
http://www.cs.grinnell.edu/~walker/courses/161.sp09/labs/lab-bash-scripts.shtml
```

---

created 20 April 2008 by Marge Coahran
revised 25 April 2008 by Henry M. Walker
revised 24 January 2009 by Henry M. Walker *last revised 8 May 2009* by Henry M. Walker

For more information, please contact Henry M. Walker at walker@cs.grinnell.edu.