csce215 — UNIX/Linux Fundamentals
Spring 2022 — Assignment 9

This assignment is intended to provide some practice and additional content for the material covered in lecture on Monday, April 18. You'll practice creating a somewhat more complex shell script, including loops and conditionals. The assignment is meant to be started in the lab sessions on Wednesday, April 20 and Thursday, April 21. It must be submitted by 11:59pm on Sunday, April 24. A total of 92 points are available.

1 Get started

One last time, you should create and navigate to a directory for this lab, begin an `recbash` recording, create a directory for this assignment, make that your current directory, and then copy over some provided files:

```
cp -rv /class/215/assignment9/* .
```

And, as always, if you are working in the lab during the scheduled time, use the QR code to mark yourself present.

Record your attendance using the QR code. Record your terminal session using `recbash`. In a directory called `assignment9`, copy the provided files as shown above.

2 Your mission, should you choose to accept it

You should see a single subdirectory in `assignment9`, called `otto`. Navigate to that directory and have a look.

In some contexts —including projects for a number of CSCE courses that you may perhaps take in the future— you'll be asked to write a program that reads input from standard input in a certain format, performs some computations on that input, and generates standard output in a certain format to show the results. In cases like that, it's common to be given

examples of correctly-formatted input files, along with other files that contain the correct output for each sample input. If a program produces the exactly correct output for each of the sample inputs, then you have some good evidence that the program is right.

However, it can be extremely tedious to re-enter those inputs and compare the outputs manually, so this is a perfect opportunity to put the philosophy of this course (i.e. 'Make the computer work for us, instead of the other way around') to work. **Your goal in this lab is to create a shell script that automates the process of testing whether a program produces the correct output for each of the given sample inputs.**

In this lab, we'll work with a real example of this type: a project that was assigned for CSCE 350, a junior-level course about algorithms and data structures, in the recent past. **You won't need to complete or even understand that project today**; instead, you'll use your skills from CSCE 215 to create a shell script called `check` that uses the sample inputs and outputs to help determine whether a program written for that project is correct or not.

To give you a sense of what you're aiming for, here are some examples of how the completed script should work.

```
$ ./check prog1.cpp
Found source file prog1.cpp.
Program prog1 is up-to-date. No need to recompile.
1.in: Correct output
2.in: Incorrect output
3.in: Incorrect output
4.in: Incorrect output
```

```
$ ./check prog2.cpp
Found source file prog2.cpp.
Recompiling prog2.cpp.
1.in: Correct output
2.in: Correct output
3.in: Correct output
4.in: Correct output
```

```
$ ./check
No program name given on command line.
```

```
$ ./check nonexistent.cpp
Source file nonexistent.cpp does not exist.
```

Before getting started, you should take a look at the provided files. Here's a description of each one.

- The **project description**, `project4.pdf`. This is the assignment that describes the input and output formats. You can ignore this file if you like, but you might be curious to understand what the programs given here are trying to do.
- Four **sample input** files, called `1.in`, `2.in`, `3.in`, and `4.in`. These are inputs, in the format described in the project description, that we can use to test programs written to complete the project.
- Four **sample output** files, called `1.out`, `2.out`, `3.out`, and `4.out`. Each one of these goes with one of the sample input files. For example, `1.out` goes with `1.in`; when a correct program gets the contents of `1.in` as its standard input, it will produce the contents of `1.out` as its standard output. Likewise for the other three samples.
- Two **example programs**, written in C++, that are attempts to complete this project, called `prog1.cpp` and `prog2.cpp`. Think of these as examples of the program you might write to complete the assignment given in `project4.pdf`. In this case, `prog1.cpp` has a subtle bug and `prog2.cpp` is a correct solution — notice in the examples above that `prog1.cpp` produces incorrect outputs for several of the sample inputs. These C++ programs are provided to you, and you don't need to modify them; you don't need to know anything about C++ for this lab.

So your mission is this: Create the shell script called `check` that uses these files to test the given programs, as shown in the examples above. The next section will walk you through the steps of creating this script.

3 Building the script

Let's work through the process of creating the `check` script one step at a time.

1. First, we need to create the script itself. Use `vim` to create a file called `check` and add an appropriate interpreter directive for a `bash` shell script at the top. Enable execute permission on this file. Test to ensure that it can be executed with the command

   ```
   ./check
   ```

   Since you have (so far), no commands in the script, running it should do nothing, but it should execute without any error messages.

2. Now let's start adding features to the script. Edit `check`, adding an assignment statement that creates a shell variable called `cpp` with the same value as the **first command line argument** to the script. Make sure your assignment works correctly even when this first argument contains spaces. (We learned how to do this in Chapter 8.) Run the script to ensure that it's working as expected.

3. Add a conditional that checks whether `cpp` is an empty string, which should happen if we run `check` without any command line arguments. Your conditional should use a `-z` primary expression. If the variable `cpp` is indeed empty, print the exact error message

   ```
   No program name given on command line.
   ```

   and `exit`. Save the script, and then run it to ensure that it's working as expected.

4. Add a conditional that checks whether the file named by `cpp` exists. Your conditional should use a `-a` primary expression. If the file doesn't exist, print the exact error message

   ```
   Source file $cpp does not exist.
   ```

   and `exit`. If the file does exist, print the status update

   ```
   Found source file $cpp.
   ```

   Save the script, and then run it to ensure that it's working as expected.

5. Add an assignment statement that creates a shell variable called `exe` that holds the name of the executable that we'll create by compiling the given C++ source file. One customary way of naming this sort of executable file is the same as the name of the source code file, but without the `.cpp` extension. Here's one way to assign the variable `exe` to that shortened file name:

   ```
   exe="${cpp%.cpp}"
   ```

   This assignment statement, which you should use in your script exactly as shown, uses one of the many additional shell features that we didn't have time for this semester, called *parameter substitution*. The basic idea is something like 'Get the value of the shell variable `cpp`, but remove `.cpp` from it.' So if `cpp` has the value `prog1.cpp`, this will assign `exe` to contain just `prog1`. There are a number of other forms of parameter substitution that you may be curious to learn about as well. After adding this assignment, save the script and then run it to ensure that it's working as expected.

6. Now we have variables `cpp` and `exe` that hold the names of the program source code and the executable that would be created from that source code, respectively. We need to add code that compiles the source code, but only if its modification date is more recent than the modification date of the executable file, or if the executable file does yet not exist. This is exactly what the `-nt` primary expression is for.

   So add a conditional that checks whether the file named by `$cpp` is newer than the file named by `$exe`. If so, print the message

   ```
   Recompiling $cpp.
   ```

   and run the command

   ```
   g++ "$cpp" -o "$exe"
   ```

   to (re-)compile the program. If not, just print the message

   ```
   Program $exe is up-to-date. No need to recompile.
   ```

   Either way, the program should continue, so no `exit` is needed here. After adding this check, save the script and then run it to ensure that it's working as expected.

7. The main goal of our script is to run the program named by `$exe` several times, once for each of the input files. To achieve that, we need a loop.

   So edit `check` again, adding a `for` loop. Use a new variable called `input` as the iteration variable. Make the loop iterate over every file in the current directory whose name ends with `.in`. Inside the loop, temporarily put an `echo` statement that prints the value of `input`, so you can see that the loop is working correctly. After adding this loop, save the script and then run it to ensure that it's working as expected.

8. Next, let's create shell variables for the names of the correct output file, the output file that will be produced by the program we're checking, and the differences between those two files. Do that by adding lines like these inside your `for` loop:

   ```
   correct_output=${input%.in}.out
   program_output=${input%.in}.$exe.out
   differences=${input%.in}.$exe.diff
   ```

   This is another example of the parameter substitution pattern that we saw above. Add these assignments to the body of the loop. Then save the script and then run it to ensure that it's working as expected.

9. Now we are finally ready to run the program named by `$exe`. We want to redirect standard input from `$input` and redirect standard output to `$program_output`, so the command should look like this:

```
./$exe < $input > $program_output
```

   Add this command inside the loop, and then save the script and then run it to ensure that it's working as expected.

10. Next, we need to check and see if the output produced by `$exe` is the same as the correct output in the file named by `$correct_output`. To perform that check, we need a command called `diff`, which is an important command for finding differences between files (but which, somehow, has not come up earlier in the semester).

`diff` 
Look for differences between files. Return code is 0 if the files are identical or non-zero if the files differ.

   So, for example, if two files `a.txt` and `b.txt` are identical but `c.txt` differs from both, it works like this:

```
$ diff a.txt b.txt
$ echo $?
0
```

```
$ diff b.txt c.txt
1c1
< aaa
---
> ccc
$ echo $?
1
```

   (Remember that `$?` refers to the return code of the previous command.) In the first example, there is no output and the return code is 0, because the files are identical. In the second example, differences between the files are shown and the return code is 1.

   Let's use `diff` to compare the file named by `$correct_output` to the file named by `$program_output`. If the files do differ, we want to capture the differences in a file named by `$differences`. So the command to check for differences will be:

```
diff $correct_output $program_output > $differences
```

Add this line to the end of the loop body in your script. Save the script and then run it to ensure that it's working as expected.

11. The last step is to print an appropriate message: If the program's output is the same as the correct output, we know that the program worked correctly. At the end of the loop body, add a conditional that checks the return code of the `diff` command from the previous step. You may want to check the lecture notes about the 'simplest form if statement' to remind yourself of how to check the return code of a command. If `diff` had a 0 return code, print

    ```
    $input: Correct output
    ```

    Otherwise, print

    ```
    $input: Incorrect output
    ```

At this point, your `check` script should be complete, with all of the features shown in those examples back on page 2. Run your script a few times to show that it can reproduce each of the four examples exactly as they are shown there.

Create the `check` shell script described above. Execute it at least four times, producing outputs **identical** to the four examples on page 2. (Hint: You might need to do `touch prog2.cpp` or `rm prog2` to force the script to recompile, to get the `Recompiling prog2.cpp` message in the second example.) *24 points*

When you've verified that your script works as shown in those examples and is complete, use `cat` to display its full contents into your recording. **If you do not `cat` the completed script into your recording, we will not be able to see it, and thus will not be able to award credit for your work on it.**

The green box below summarizes the features we'll check for when grading your script. It's strongly recommended to use this as a checklist to verify that you've got things right before submitting.

Create a shell script called `check` within the `otto` directory. Use `cat` to display it in the recording. The script should:

- Contain an appropriate interpreter directive on its first line. Be executable within the `otto` directory using the command `./check` .
- Assign a variable called `cpp` to the first command line argument.
- Use a `-z` primary expression to check whether `cpp` is an empty string. If `$cpp` is an empty string, print the appropriate error message and quit.
- Use a `-a` primary expression to determine whether `cpp` refers to a file that exists. If `$cpp` refers to a file that does not exist, print the appropriate error message and quit. If `$cpp` refers to a file that does exist, print the appropriate happy message and continue.
- Assign a variable called `exe` to be the same as `$cpp`, but with `.cpp` removed.
- Use a `-nt` primary expression to determine whether `cpp` refers to a newer file than `exe`. If `$cpp` is newer than `$exe` or `$exe` does not exist, use the appropriate `g++` command to create `$exe`. In either case, print the appropriate message.
- Use a loop, with `input` as its iteration variable, over `*.in`.
- Within the loop, assign values to the shell variables `correct_output`, `program_output`, and `differences`, as shown above.
- Within the loop, execute the program named by `$exe`, with input and output redirected as shown above.
- Within the loop, use `diff` as shown above to compare the sample output to the output produced by the program and capture the differences in a file named by `$differences`.
- Within the loop, use the return code from `diff` to print an appropriate message, showing whether the program worked correctly or incorrectly on `$input`.

68 points

4 *Done*

Whew! That's the end of the last assignment! Congratulations on learning quite a bit about how to use UNIX- and Linux-like operating systems (and on setting the stage to learn lots more in the future)!

As always, you should finish the `recbash` recording, check that your work is shown correctly in the transcript, and submit.

Upload your recording(s) and then submit.

And here's one last batch of instructions for after you've submitted.

Prepare for the final exam by studying the practice exam questions on the course website. Ace the final exam. This summer, make good choices and have some fun!