csce215 — UNIX/Linux Fundamentals
Fall 2021 — Lab Assignment 4

This assignment is intended to provide some practice and additional content for the material covered in lecture on Monday, September 20. You'll use a variety of special characters to specify command line arguments; compile, run, and debug a program using compound commands, and practice using command substitution. The assignment is meant to be completed in the lab sessions on Wednesday, September 22, and must be submitted by 11:59pm on Friday, September 24. A total of 40 points are available.

1 Get started

As usual, this for this assignment you'll submit an `asciinema` recording of your terminal session completing the tasks below. So fire up the recording —Perhaps `lab4.cast` would be a good filename?—, create a `lab4` directory, and navigate to it.

> Use `asciinema` to record your terminal session. Create new directory called `lab4` to use for the tasks below.

The exercises below will rely on a collection of files from the zip archive at this url:

```
https://cse.sc.edu/~jokane/215/lab4.zip
```

Use `wget` to download this file into your `lab4` directory, then use `unzip` to extract all of the zipped files.

> Download and unzip `lab4.zip`.

2 *Look at this mess!*

Many of the files extracted from `lab4.zip` are in a subdirectory called `mess`, which contains a very large number files whose names are, to put it mildly, quite strange. Navigate to that directory and have a look. Notice that some of the file names even contain spaces, including spaces at the beginning and/or at the end of the file name! Yuck!

Use `cd` to navigate to the `mess` directory and use `ls` to see its files. *1 point*

Fortunately, there is some structure: The name of each file has a single dot, splitting the full filename into a **name part** before the dot and an **extension part** after the dot. But note that, in these files, either of these elements can be empty.

Let's use this directory to practice looking for certain groups of files. As a reminder, remember that `ls -a` prints a list of all of the files the current directory, including those whose names start with a dot, i.e. hidden files. Also remember that `wc -l` (doubleyou sea dash ell) is a filter that outputs the number of lines in its standard input. So, for example, `ls -a | wc -l` will output the number of files (both hidden and non-hidden) in the current directory.

Give commands that output a list of files in `mess` in each of the following groups. For each one, you should find (a) a single command that outputs an exact list of the files asked for, and (b) a second command that outputs the number of files in the list.[1]

1. All files, including hidden files.
2. Files whose extension part is exactly a lowercase `c`.
3. Files whose name part starts with a lowercase `z`.
4. Files whose name part contains exactly two characters.
5. Files whose extension part contains exactly three characters.
6. Files whose names contain a Q or q in either the name part or the extension part.
7. Files whose name contains at least one space in either the name part or the extension part.
8. Files whose name contains a lowercase `x` followed eventually by a lowercase `y`, in either the name part or the extension part.

[1]Hint: For (a), use `ls` with appropriate wildcards. Check your notes for some explanation and examples of wildcards. For (b), pipe the output from (a) into `wc -l`.

To help you verify that you have correct commands, here are the counts, not necessarily in order.

11 13 64 344 532 633 1008 4954

For each of the 8 groups listed above, give a single command that outputs a list of files in mess in that group and another command that outputs the number of files in that group. *16 points*

3 *Fix this program*

Next, navigate to the 'pairs' directory. This directory contains a C++ program written by your instructor, whose job is to check whether the opening and closing parentheses and braces in its input are balanced correctly. (If you have not yet worked with C++, don't worry. Completing this part does not rely on knowing C++.)

Unfortunately, the instructor has been sloppy and the program has a few errors. Let's try to compile it to see them. Here's the command to compile a C++ program.

g++

Compile a C++ program.

-o *name* specify a name for the output executable

In this case, the source code is just one file called pairs.cpp and we want to create an executable called pairs. So this command should do the job:

```
g++ pairs.cpp -o pairs
```

Then, if that's successful, we can to run the program with this command:

```
./pairs
```

(The ./ part tells the shell to look for the program in the current directory, rather than its usual search locations. We'll learn more about this in the coming weeks.) Go ahead and try it, issuing both commands on the same line, with && between them so that the second command is executed only if the first is successful.

> Give a single command that attempts to compile `pairs.cpp` into an executable program called `pairs`, and then executes that program only if the compiling step is successful. *2 points*

You should see quite a few compile errors. Let's fix them. For each of the errors listed below, use `vim` to correct the problem in `pairs.cpp`, and then use the same compile-and-if-successful-then-run command as above to try it.

1. The first error complains about `cin` not being declared, and suggests doing something like `#include <iostream>`. To fix this, edit the file using `vim` and change the first line of the file to include `iostream` instead of `ostream`.

   Then quit `vim` and try the compile-and-if-successful-then-run command again. (You might be tempted to try to fix the other errors as well, but that's often a bad idea, because one problem can sometimes cause many different, seemingly unrelated compile errors. So after fixing one problem, it can be helpful to compile again.)

   > Use `vim` to fix the typo on line 1 of `pairs.cpp`. Then save and quit, and repeat the line that compiles and runs only if the compile is successful. *3 points*

2. The list of errors should be much shorter now. The next one complains about line 26. This is supposed to be an `if` statement, but somehow the `f` is missing. Fix this and then save, quit, and try to compile/run again.

   > Use `vim` to fix the typo on line 26 of `pairs.cpp`. Then save and quit, and repeat the line that compiles and runs only if the compile is successful. *3 points*

3. Now the first error is a problem with line 43. This line is missing a colon and has a < where it should have <<. Fix these two problems. The result should look similar to line 44. Then save, quit, and try to compile/run again.

Use `vim` to fix the typos on line 43 of `pairs.cpp`. Then save and quit, and repeat the line that compiles and runs only if the compile is successful. *3 points*

If you've made these corrections, the program should compile cleanly now and begin to run. The program reads from standard input (i.e. your keyboard), after compiling is finished, so it may look at first as though the compiling is simply taking a long time.

Try it out by typing a line or two containing parentheses, square brackets, or curly brackets along with any other characters you like. If each `(` matches eventually with a `)`, each `[` matches with a `]`, and each { matches with a }, then the program should output `ok` at the end of standard input (i.e. when you type `Ctrl-D`). If any of these pairs of grouping symbols matches with the wrong partner or with nothing at all, you should see an error message.

So, does the program work correctly? (Spoiler alert: No. No, it does not.) There are two files here that can help us check. One is called `a.in`, which has all of the parens and braces correctly matched, so `pairs` should report success. The other is called `b.in`, which has mismatches, so `pairs` should generate an error message about a mismatched symbol report failure.

Let's try `a.in` first. Add an input redirection to your previous command, so that it uses `a.in` as the input to the `pairs` program. Then try the same with `b.in`.

Use input redirection at the end of the previous command, instructing the shell to use `a.in` as the input for `pairs`. Repeat for `b.in`. *4 points*

Based on those outputs, it looks like something is wrong with the program: in `b.in`, there are `[` characters that match with `)`, but the program does not catch this. What could be going wrong?

Aha! There's a logic error in the program. On line 28, we have || where we should have &&. Use `vim` to correct that mistake. Repeat the two commands that run the program against the `a.in` and `b.in` input files, confirming this time that `a.in` still generates an `ok` message, and that `b.in` now correctly causes an error about mismatched symbols on line 1. Then use `cat pairs.cpp` to show the fully-corrected program into your recording.

Use `vim` to correct the logic error on line 28 of pairs.cpp. Repeat the two commands above: one that compiles and runs with `a.in` as input, and another that compiles and runs with `b.in` as input, showing the correct behavior for each one. Use `cat` to show the now fully-corrected `pairs.cpp`. *4 points*

4 A very cool command

One last series of tasks: Let's use command substitution —the technique of using the output of one command to form arguments to another command— to find and copy a file from one of the system directories.

The file we're looking for is an image in scalable vector graphics (SVG) format of a smiling face with sunglasses. We know that the filename has the word 'cool' in it and that it has an `svg` extension, but we don't know anything else about what the file is called nor where the file might be.

Fortunately, there's a program called `locate` for just this sort of situation.

`locate`
List files on the system matching a pattern.

For example, to find files related to the java programming language, a command like

```
locate java
```

will list all of the files whose names or directories contain the word `java`, of which there are quite a few.

In this case, we are interested in an SVG file containing 'cool' in its name. So first, use `locate` to find the file that we're looking for, that is, a file with an `svg` extension, whose name contains the word `cool`. We just want to copy one single image file, so you'll need a command that outputs a single line containing the full path to the cool SVG file we are looking for. If you want, you can use pipes to process the output of `locate`. (Your system probably has multiple copies of this file; any of them are OK.)

Use `locate`, optionally piped through one or more other commands, to generate a single line of output. That line should contain the path name of a file whose name contains the word "cool" and whose extension is `svg`. *2 points*

Now we can use that `locate` command in a command substitution —check the notes if you need to remind yourself of the syntax— as part of another command that copies this file to the current directory. So this should be a `cp` command where the source, i.e. the first argument, comes from a command substitution of our `locate` command, and whose destination is the current directory, i.e. a single period (.). To see the details of what is being copied —that is, where we're copying from and where we're copying to— let's use the `-v` option to `cp`.

If this has worked correctly, you should be able to see the cool new file using `ls`. Optionally, if you'd like to see the file itself —It is an image, after all— you can use the `eog` command to display it in a separate window.

Use `cp -v`, with a command substitution for the first argument, to copy exactly one file, whose name contains the word "cool" and whose extension is `svg`, to the current directory. *2 points*

5 *Mission accomplished*

That's all for this time. As always, use `Ctrl-D` or `exit` to end your recording. Then consider replaying it to ensure that all of your work has been captured correctly. Finally, upload to the Dropbox site and use the submit button to complete the process.