Lab 2
Mad Libs

8 September 2015

The drill for this lab is the **Chapter 3 drill**. Come to lab on Tuesday either with it completed or with a specific written question in your notebook identifying which drill step you got to and what about it you're stuck on.

Notes and adjustments:

- In your readme for this week's lab, follow the pattern for documentation (from last week's lab): say what that program does, how to compile it, and how to run it.
- Make note of any test cases—plausible inputs and the corresponding expected outputs—that you think of while writing the drill. For now, write them either in your notebook or in a file in the directory; during the lab I'll give more specific instructions on how I want them formatted and turned in.

I'll be circulating around the lab to answer questions. If you're stuck on some part of the drill, ask me about that (and while you're waiting for me to get to you, look at the next section about paths). If you're not stuck but haven't finished the drill, work on that now. If you're done with the drill, go on to the next section.

Command Line Feature of the Day: paths

When you specify a filename on the command line, it doesn't have to be in the current working directory; but if it isn't, you have to specify a "path" of how to get there.

Open a fresh PuTTY connection to a department machine—remember that when it first opens, you're in your home directory. Run the "`ls`" command to list the directory, which should include a couple directories from the last two labs, but not really any of the individual files from the labs (which should be inside the directories). If your directory from last week was named `lab1`, you could type "`ls lab1`" to list its contents. (For the following paragraphs,

I'll assume the directory was `lab1` and one of the files was `hello_world.cpp`, but if you used other names for them, substitute accordingly!) If you type

```
cat hello_world.cpp
```

you will get an error message (try it), because there isn't any file of that name in the current working directory.[1] But if you type

```
cat lab1/hello_world.cpp
```

you are telling the `cat` command to look *inside* the `lab1` directory for a file named `hello_world.cpp`—which is where it should be.

Now, use `cd` to change to the directory you made for this week's lab work.

If you run the `ls` command now, you'll see the contents just of this new directory. How could you look at `hello_world.cpp` now? If you again type

```
cat lab1/hello_world.cpp
```

you'll get an error message (try it), because there isn't any `lab1` *here*; you'd have to go "back one" or "up one" first. As I've mentioned before, there is a special shortcut name for the "up one" directory, from wherever you currently are: two periods ("`..`") is a reserved special symbol for the "up one" (or "parent") directory. So type

```
cat ../lab1/hello_world.cpp
```

and you should again see the listing. Here, starting from the current directory (which is your directory for Lab 2), the path says to go up into the parent directory, then down into the `lab1` directory, and there it will find `hello_world.cpp`.

Note also that it's important that in the paths described above we want to start from wherever you currently are—this is called a "relative path"—and so they don't *start* with a slash. If they did, that would be an "absolute path", and would mean to start at the top-level "root" directory of the filesystem. An example of that is the directory where I've put files we write in class, `/home/shared/160-3`. You can type

[1] Unless you left a copy in your home directory as well, of course.

```
ls /home/shared/160-3
```

to see what's in there, and (for instance)

```
ls /home/shared/160-3/0904
```

to see the directory we made on Friday. You can also `cd` to that directory and `cat` the files in there, or even open them with `vim`, to review what we've done.

Testing a program

If you've been compiling and running and testing your code as often as I (and the book) have encouraged you to do, you're probably getting a little sick of typing the same sample input over and over again. (If not, you're probably not running and testing enough!)

The normal response of a computer scientist faced with something annoyingly repetitive is to figure out how to automate it, so that's what we'll work on now.

Command-line redirection

We often pretend that `cin` always reads from the keyboard and `cout` always writes to the screen, but that's not quite true: that's just what they do by default. You can tell a program to read from/write to other places—files—instead.

Let's try it out. Edit a file called `drilltest1.in`, and in it type the things that you would normally give as input to the program from the Chapter 3 drill. (That should be: two friends' names, the sex (`m` or `f`) of the second one, and an integer age.) Then, at the command line, run your program as follows:

```
./a.out  < drilltest1.in
```

The arrow—a relative of the `<<` operator we use in C++ programs—tells the system to get input for the program from that file instead of from the

keyboard. Notice that it still prints the prompts, even though they turn out to be unnecessary. Is its output correct?

Later on, we'll see other convenient forms of redirection, but for now, this will be a good timesaver. It's also a way for you to be methodical about testing your code (and for me to verify that you've done so).

Completeness

I've said in class that running a program once isn't generally enough to test it. In this drill, we start to see why—a single test would not be able to demonstrate all the different parts of the code. Indeed, you'd need at least five or six different sets of input, which we'll call "test cases", to make sure that all the age conditions are verified.

Write more test cases, in files named `drilltest2.in`, `drilltest3.in`, and so on, that (collectively) cause all the different messages from the drill program to be printed. Note that each one still needs to follow the full input format (name, name, sex, age)!

Documentation

Remember that your readme file (`README.TXT`) is your way of explaining to me what you've done and how you've met the requirements—and making sure I know how to run your program. This extends to telling me how to test it. You already should have lines in there that say

```
To run:
    ./a.out
```

Immediately after them, you should now add lines that say:

```
To test:
    ./a.out  < drilltest1.in
```

You should have one of these for each test case you've written, and for the ones designed to test particular things, you can say so, e.g.:

```
To test (general):
    ./a.out  < drilltest1.in
```

```
Testing age 17, male:
    ./a.out  < drilltest2.in
Testing under 12, female:
    ./a.out  < drilltest3.in
```

and so on. It's convenient if the running/testing commands are on a line by themselves, which makes them easier to quickly highlight and copy and paste. In PuTTY, highlighting something automatically copies it to the clipboard; then right-clicking pastes it—for instance, at the command line. Saves typing.

Mad Libs

A "Mad Lib" is a story constructed by prompting someone for words in certain categories (adverbs, proper names, numbers, articles of clothing) and using them to fill in blanks in a template.[2] You now have the tools you need to write a program that does exactly that—it will be a lot like the drill, but you get to choose what direction it takes.

Your program should prompt for at least six things. At least one of the things you have to use twice (or more) in the story; and at least one of the things has to be a number (on which you do math) in the story. Your story doesn't have to be particularly long, and will probably be fairly surreal. That's ok! Have fun with it.

Before you get in too deep, write at least one of your test cases (maybe even do that first), so that you can automate your compile-and-test technique. Also make sure to add a new section to your readme to reflect this second program for the week.

[2] I didn't invent this idea or the name. Google it.

Handing in

If all your files are in a directory named `lab2`, you should be able to go to your home directory (by calling `cd` by itself) and then running

```
handin cmsc160-3 lab2 lab2
```

The final version (i.e. including the Mad Libs part) is due Monday afternoon (14 September) at 4:00pm.

Rubric

Don't forget to clearly indicate which rubric points you think you have or haven't gotten. You can put this information directly in the readme.

RUBRIC

1 Attendance at lab with drill done or question written down
1 Readme file exists, includes description
Drill
1 Readme instructions for compile/run/test of drill
1 Compiles, runs, steps 1–3 and 7 (read/write)
½ Step 4 (ask sex, use pronoun)
½ Step 5 (error on bad age)
½ Step 6 (age responses)
1 Inputs (`.in` files) cover needed test cases
Mad Lib
1 Compiles, runs, reads/writes, basic documentation
1 Read and used ≥ 6 inputs
½ Used an input twice in story
½ Did math on an input
½ Appropriate test cases provided