Lab 5
More loops, and reading code

26 September 2019

Drill: finding the earliest letter

For the drill this week, you'll write a program that reads a single word and prints out the alphabetically-earliest letter in that word. You can assume that the word is letters only and will be either all-caps or all-lowercase (but it could be either one!).

1. Open the repl.it assignment for Lab 5 drill. Enter the boilerplate code (this should be feeling a bit more automatic, but you can still look it up if you need to!)
2. Add code to read in a single word. Don't prompt the user for it (it'll break my test cases).
3. Write a loop that accesses each character of the word.
4. (Are you remembering to check after every step that your program compiles and runs?)
5. Add three lines of code to build an accumulator. In this case, the accumulator will be tracking the earliest letter seen so far (which at the end will thus be the earliest letter overall). Remember the three items on the checklist:
   - Define and init the accumulator (before the loop)
   - Update the accumulator (inside the loop)
   - Use (e.g. print) the accumulator (after the loop)

   If you're not yet sure what you should use to init the accumulator, put *some* arbitrary value in there (and fix it later). If you're not sure how to update, put a comment inside the loop reminding you to update the accumulator (and fix it later).
6. Fix the body of the loop, if you haven't already. When and how is it appropriate to update the earliest-letter-seen-so-far value (which is what we're storing in the accumulator)?

7. Fix the initial value of the accumulator, if you haven't already. What is a "safe" initial value for this variable that won't break the rest of the algorithm? (There are at least two correct answers to this question, in the context of the current problem.)

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 dotfiles). If you're not stuck but haven't finished the drill, work on that now. If you're done with the drill, continue on to the next section.

Vim FOTD: "ex" mode

Open two terminal windows, and arrange them side-by-side. The one on the left I will designate the "edit" window, and the one on the right the "other" window. In the edit window, edit a file named `dummy.txt`, type a few lines into it, and save and quit. (The content doesn't matter at all.)

In the other window, `cat` the file, that is, type

```
cat dummy.txt
```

You should see the exact contents you just typed in. If not, make sure both windows are in the same directory (your home directory is fine) and try again.

Return to the edit window, again edit the `dummy.txt` file, and add a couple more lines. This time, don't save and quit. Back in the other window, `cat` the file; since you haven't saved yet, what should you see?

Return to the edit window again. Make sure that you are in command mode (hit escape), and this time, instead of save-and-quit (with `:wq`), just save: if you type `:w` by itself and hit enter, this writes the file without quitting. Now in the other window, `cat` the file again—you should now see the updated version.

What you are seeing here is (more of) a third mode of Vim besides insert and command mode, called "Ex mode". Ex was an editor back in the day, a precursor to vi (which was itself the basis for vim). All interactions with the ex editor were done through commands typed on a line that started with a colon, and vestiges of this survive in the modern Vim editor; from ex mode

you control file access operations, among other things. The `:w` command that you have just seen simply saves ("writes") the current file.

Another command is to save to a different file. After you issued the `:w` command a moment ago you were returned to command mode, so press the colon key again to return to ex mode and type

```
w dummy2.txt
```

This creates a brand new file, named `dummy2.txt`, with the current contents of the edit buffer. But unlike "Save As..." in a typical modern word processor, it doesn't change the default name of the file, so that if you type `:w` again, it will save it under the original name (`dummy.txt`).

You can use ex mode to edit a different file. Type

```
:e newfile.txt
```

and the `dummy.txt` file will disappear from the window, to be replaced by an empty buffer. If you type text in here and then hit `:w`, it will be saved under the name `newfile.txt` (as you can verify by using `ls` and `cat` to look at the file from the other window).

Ex mode can be used to quit Vim by typing `:q` and hitting enter. Note that this does *not* include writing out the file first. If you type `:q` before saving (and you can tell that the file has been edited by the "`[+]`" after the filename in the status bar), Vim will warn you that you haven't saved the file. You can then type

```
:q!
```

to say, no really, I mean it, just quit (don't save).

There are other ex mode commands that we'll learn eventually, but these file-related commands enable a particularly useful interaction style: you can now leave your source code open in one window, save it, and run the compiler in the other window. This is useful so that you can keep any compiler errors on the screen while you scan the code for the problem; in fact, from now on you should get in the habit of having at least two windows open when you're programming: one for editing, and one for compiling and testing. (Some of you have already been doing this, but it's even more streamlined now.)

Reading and modifying code

For the remainder of the lab, you'll be working on a program that I already started: reading it, understanding it, fixing it, and adding to it. To start, on the department server, make a directory for this lab (if you haven't already) and change into that directory. Then, type this:

```
cp /home/shared/160/lab5/errormeasure.cpp .
```

(don't forget the dot at the end, which says to copy it into the current directory).

Part a: the existing stuff

Read the code and see what it says it's doing and what it's trying to do. Trace it and run it, to see what it's actually doing.

Like the code we worked on in class yesterday, this code has a bug. However, it's a different bug. Debug it!

Part b: root mean squared error

The data in the provided vector is meant to represent error measurements, i.e. how far from an ideal curve a bunch of measurements fell. One way that such measurements are processed is called "root mean squared error" (or RMSE). It is: the square root, of the average (mean), of the squares, of each value. (It's used because it tends to give greater weight to outliers.)

You're going to add to the program some code to compute and print the RMSE for whatever is in the vector.

Recall the steps of algorithm design:

1. Understand the problem
2. Work through examples by hand and write them as test cases
3. Explain algorithm (pseudocode) including nameable values
4. Set up the boilerplate and type in the test cases
5. Encode algorithm in C++
6. Test

In this case, you already have one vector of values you could use as the input half of a test case (but you'll have to compute the result by hand). You also could comment out that line and try a different group of values for `errors` that you might find easier to compute with. Indicate your test case's expected output in a comment (and if you have more than one test case, you can just leave the extra TCs commented out).

The algorithm to do this will combine a few of the techniques we've used recently (and one or two things we haven't looked at in a while), but it does not require any particularly exotic corners of C++ that we haven't covered yet.

Handing in

It's due as usual on Wednesday at 4pm. The drill is already on repl.it and should be submitted there; the errormeasure portion needs to be handed in on the server. Hand in using the usual command, this time with assignment name `lab5`.

Rubric

RUBRIC (Tentative)

1 Attendance at lab with drill done or question written down
1 Appropriate documentation in each part
Drill (earliest letter)
1 Program compiles and runs, reads string, prints something
1 Loops over entire string and does accumulator checklist
1 Accumulator init and update
Rest of lab (errormeasure)
1 Program compiles and runs, does anything more than original
1 Part a has been debugged, prints correct answer
3 Part b:
- **½** test case (at least one) indicated in comments
- **½** good variable names in algorithm
- **1** loop and accumulator
- **1** computes and prints correct result