

Lab 5

More loops, and reading code

1 October 2024

Part 1: finding the earliest letter

For the first part 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 codeboard assignment for Lab 5 part 1. 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, and submit it often?)
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:
 - Declare-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 preview, ask me about that (and while you're waiting for me to get to you, look at the next section). If you're not stuck but haven't finished the preview, work on that now. If you're done with the preview, continue on to the next section.

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). There's also a (minimal, unfinished) readme in that directory, which you can copy if you want—either way, you'll have to write the contents of the readme yourself.

Part 2a: 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! Don't forget to hand in when you've fixed it. This would also be a good time to edit the readme appropriately.

Part 2b: 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.

We’ve been designing algorithms in an increasingly formal way in this course, so let’s lay this out as a specific process that we follow:

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).

You *may*, but don’t have to, use the read-vector-from-input pattern we saw at the end of class yesterday, to build your test cases. If you do that, make sure the readme file tells me where to find them and how to use them. If you’re not comfortable with that yet (it’s really not the focus of this week’s lab) you can stick with declare-and-init the vector and give its expected results in comments.

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.

AI policy and frequent submission

(unchanged this week)

Some use of generative AI is fine, but you a) should not paste this assignment or type it verbatim into the AI prompt, and b) should not be asking the AI for the whole program all at once. (Just like you can ask for help from a human, but should not have them write the whole program for you!) If you get help from an AI chat OR from a person, you should note that in a program comment near whatever you got from them.

Relatedly, I expect that you'll submit—i.e. run the `handin` program—relatively often, and I *require* that you do so at least 2–3 times over the course of working on each part of the lab. As a rule of thumb, hand it in after completing each 1–2 points on the rubric. That gives me a version history and progress report on your work. Submissions that jump straight to a final, (near-)correct version with no intervening submissions along the way *will receive little or no credit* for that part. ← scoring note!

This is a new policy that I'm experimenting with; let me know if you have any feedback.

Handing in

It's due as usual on Monday at 4pm. The first part is already on codeboard 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 preview done or question written down
- 1 Appropriate documentation in each part
- Part 1 (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
- Part 2 (errormeasure)**
- 1 Program compiles and runs, does anything more than original
- 1 Part a has been debugged, prints correct answer
- 3 Part b:
 - $\frac{1}{2}$ test case (at least one) indicated in comments
 - $\frac{1}{2}$ good variable names in algorithm
 - 1 loop and accumulator
 - 1 computes and prints correct result

This document was written and prepared without the use of generative AI.