

Lab 5

More loops, and reading code

20 February 2025

v20250220-1315

Part 1a: 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 1a. 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 among the breakout rooms to answer questions (also you can talk to the other people in your breakout room, and you can message me on slack). 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.

Part 1b: Input loops

The task for part 1b is *almost* the same as for part 1a: it should read multiple numbers, which represent measurements, and prints out the smallest recorded measurement. You can assume that there will be at least one number.

You should use the "read until you run out of input" pattern that we discussed in class (and the book); recall that in codeboard you can terminate the input on a sequence of numbers by typing a non-number (the test cases use "xxx"). I recommend using that loop to read the values into a vector, and then process the vector (it makes the accumulator work more cleanly).

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/* .
```

(don't forget the dot at the end, which says to copy it into the current directory). This copies *all* the files from that directory into the current directory. ← Updated!

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 two groups 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. Add your additional test cases as `.in/.expect` file pairs as we've done before.

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 Wednesday at 4pm. The first parts are 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 Appropriate documentation in each part
- Part 1 (earliest letter, lowest number)**
- 1 Programs compile and runs, read something, print something
- 1 Loops over entire string/vector and does accumulator checklist
- 1 Accumulator init and update
- 1 Reads all numbers
- 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 additional) built and in readme
 - $\frac{1}{2}$ 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.