

Lab 3

Expressions and design

17 September 2024

Tinkerblock preview

Make a directory for this lab. In your directory for this lab, you'll start the early part of the design process, and move towards building a working program.

To recap: I'm running a Tinkerblock factory; the Tinkerblock connectors are long sticks of wood with square cross-sections (the fancy term for this is a right rectangular prism, but you don't need to know that). My core question, which this program should answer, is what is the total surface area of one run of these sticks? We decided in class that we would need three inputs (the width of the square cross-section, the length of the piece, and the number of pieces).

1. In the README.txt file (or in your notebook, but it will eventually go in the readme file), write down at least a brief summary of the problem statement.
2. In your notebook, work out at least two test cases for the problem. Use actual numbers that are real-ish (if you measure in millimetres they could all be integers :). At this stage all your computation can be with the actual numbers of your test cases.
3. In your notebook, write out using generic variable names the computations you just did when you "did it by hand". Pick good names for the nameable intermediate values. This is your pseudocode.
4. Put your test cases into `.in` and `.expect` files and write a `.cpp` file with the boilerplate stuff (`#include`, `main`, etc), and check that it compiles and runs before you start adding more.
5. (And don't forget to hand it in occasionally, this week as `lab3`.)

6. Add the pseudocode to your `.cpp` file piece by piece, writing code to read in data, compute the required values according to your algorithm, and print a result. Try to compile and test your code after every meaningful chunk that you add.

I'll be circulating around the lab to answer questions. If you're stuck on some part of the first problem, ask me about that (and while you're waiting for me to get to you, start looking at and thinking about the next section. If you're not stuck but haven't finished the Tinkerblock problem, work on that now. If you're done with the preview, continue on to the next section.

Another problem

Consider the following scenario:

Imagine the owner of a movie theater who has complete freedom in setting ticket prices. The more they charge, the fewer the people who can afford tickets. In a recent experiment the owner determined a precise relationship between the price of a ticket and average attendance: at a price of \$12.50 per ticket, 120 people attend a performance. Decreasing the price by a quarter (\$0.25) increases attendance by 15. Unfortunately, the increased attendance also comes at an increased cost. Every performance costs the owner a base amount of \$450. Each attendee costs another ten cents (\$0.10). The owner would like to know the exact relationship between profit and ticket price so that they can determine the price at which they can make the highest profit.¹

As with the tinkerblock problem, the program you write will not quite answer the ultimate question (here, what price makes the highest profit), but it will be a tool that lets someone inform such a decision by trying certain input values and seeing what output values they result in.

Work through the problem-solving process again, this time on the theater profit problem. These are the six steps of the design process we've followed for these problems:

¹Adapted from Felleisen et al, "How to design programs," §3.1

1. Understand the problem: input(s)? output(s)? description?
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

There should be some written thing for each step in the process; some will be reflected directly in the final program, others in other files such as the readme or in test case files (including both the `.in` and its corresponding `.expect` for each test case). The readme file *should definitely* include at least the description and other documentation for the lab (see below).

Your test cases for this second one will need filenames that look different from the earlier test cases! Devise an appropriate naming scheme for them.

When it comes time to write pseudocode, don't forget that you need to be using several intermediate values, each expressing one piece of the computation, rather than trying to cram all the computation in a monolithic (and incomprehensible) one-liner.

Organising the readme

In your file named `README.txt`, there's starting to be a few different things that need to be in there. The name and date and assignment (here, "Lab 3") that we've been putting at the top of the `.cpp` file should go in the readme too (or instead). This file is also the place you should put instructions on how to compile the program, how to test it, and how to run it as a regular user, and any instructions (such as these) that are meant to be typed at the command line should be put on a line by themselves, to stand out visually and for easy copy-pasting.

Part of the need for the explicit instructions is that there are now multiple unrelated `.cpp` files in the directory, and two unrelated sets of test cases. There should be *just one* readme for the whole lab directory, and it will give clear instructions for what to do with all the files. Here's a checklist for the stuff that should go in the readme this week:

1. Name, date, assignment

2. Description of tinkerbloc program
3. How to compile it and run it
4. How to test it
5. Description of theatre profit program
6. How to compile it and run it
7. How to test it
8. Notes about tests that are failing or anything else you need to tell me

If you've decided not to print a prompt in your program, that's fine, but then your instructions for how to run it, in the readme file, should certainly say what the user is expected to type!

AI policy and frequent submission

(minor edits to the second paragraph since last 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

As before, the lab is due 4pm on Monday. Submit it as `lab3`; put both parts (tinkerbloc and theatre profit) in the same directory and hand the whole thing in with each submission.

Rubric (tentative)

RUBRIC

1 Attendance at lab with preview done or question written down

Documentation (readme)

1 File exists, includes descriptions

1 Instructions for compile/run/test

1 Tests pass *exactly* OR are mentioned as not passing

Tinkerblock factory

1 Compiles, runs, reads correct required inputs

$\frac{1}{2}$ Test cases (worked out examples)

$\frac{1}{2}$ Appropriately named intermediate values

1 Prints correctly-computed result

Theater profit

1 Compiles, runs, reads correct required inputs

$\frac{1}{2}$ Test cases (worked out examples)

$\frac{1}{2}$ Appropriately named intermediate values

1 Prints correctly-computed result