Lab 3
Expressions and design

12 September 2019

Tinkerblock drill

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. 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 drill, ask me about that (and while you're waiting for me to get to you, look at the next section about completion). 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. This week we have not one but *two* features of the day!

Feature of the day: Saving you some typing

Tab completion on the command line

At your command line prompt, from your home directory, type

```
echo "This is a test"  > quaffle-160-long-name
```

This will create what is presumably a new file named "`quaffle-160-long-name`" in your home directory. Boy, that's a long name. So type this:

```
cat qua
```

(no space at the end) and hit Tab. Since this is presumably the only file in your directory that starts with "`qua`", the command line will *tab-complete* the filename for you. Helpful!

Sometimes, it can't give you a complete filename, but it might still be helpful. Type

```
cat la
```

and hit Tab. By now you shuold have *multiple* directories that start with "`la`"; it should complete as far as it can (adding a `b` to make "`lab`"), and then let you type the rest, and if you hit Tab again it will give you a list of valid completions.

Tab completion is a feature of all modern command line shells. It has even made its way into Windows's Command Prompt. If you type enough to uniquely identify a file, it will complete the filename for you, followed by a space, so you can type the next argument or hit enter. If there are multiple choices, it'll fill in as much as it can, and then wait for you to finish.

I trust that a well-cultivated sense of laziness will addict you to this feature fairly quickly. Hitting Tab will become part of your typing muscle memory within days—if not hours.

Completion in Vim

Open vim to edit a `.cpp` file you've already got lying around, such as `inorder.cpp` from last week's lab.

Now, go to a blank line in the file, enter insert mode (by pressing '`i`') and type "`in`" (minus the quotes), then hit Ctrl-N repeatedly (still in insert mode). You will cycle through everything starting with those two letters, which should at least include "`include`" and "`int`", and possibly also "`input`" or some other variable name depending on the program you're in, eventually cycling back to just plain old "`in`". Add a "`c`" after the "`in`" and the Ctrl-N will only give you "`include`" since you've ruled out the others. If you use Ctrl-P ("previous") instead, it cycles in the reverse order.

I have encouraged you to use descriptive variable names, and this makes doing so a lot more feasible. Basically, as you edit a file, vim will keep track of all the words (variable names, function names, reserved words like "`else`" and "`double`") in that file. When you type part of a keyword, Vim knows what other keywords in that file could match what you've typed; and Ctrl-N and Ctrl-P will let you use these potentially long names without typing them all out each time.

Go ahead and undo the changes you've made to this file (in command mode, press '`u`' a couple times until you run out of changes), and exit.

Another problem

Consider the following scenario:

> Imagine the owner of a movie theater who has complete freedom in setting ticket prices. The more he charges, 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 $12.50 per ticket, 120 people attend a performance. Decreasing the price by a quarter ($.25) increases attendance by 15. Unfortunately, the increased

> attendance also comes at an increased cost. Every performance costs the owner $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 he can determine the price at which he 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. Recall that these are the steps of the design process I laid out in class yesterday:

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.

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

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). There should be *just one* readme for the whole lab directory, but since the directory now contains files for two different programs, it's now becoming important for the readme to also contain instructions for how to build the program as well as testing it. Here's a checklist for the stuff that should go in the readme this week:

1. Name, date, assignment
2. Description of tinkerblock 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

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!

Handing in

As before, the lab is due 4pm on Wednesday. Submit it as `lab3`.

Rubric (tentative)

RUBRIC

1 Attendance at lab with drill 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

½ Test cases (worked out examples)

½ Appropriately named intermediate values

1 Prints correctly-computed result

Theater profit

1 Compiles, runs, reads correct required inputs

½ Test cases (worked out examples)

½ Appropriately named intermediate values

1 Prints correctly-computed result