Lab 3

28 January 2021

Start a fresh directory for this week's lab (if you haven't already) and copy in my maze code from `/home/shared/162/lab3/` *and also* your `Location` code from Lab 2 (including tests). Today we'll continue our study of classes, the maze project, and test-driven development. But first...

Feature of the day: Saving you some typing

Tab completion on the command line

Do the copying mentioned above and `cd` into your directory for this lab. Then, at your command line prompt, type

```
cat Loca
```

(no space at the end) and hit Tab. Since all of the files in this directory that start with "`Loca`" follow that with "`tion`", the command line is able to partially *tab-complete* the filename for you. Helpful! Even better: hit Tab again, and it should list all of the files that start that way, so that you know what you could type next.

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 `Location.h`.

Go to a blank line in the file, enter insert mode (by pressing '`i`') and type "`isE`" (minus the quotes), then hit Ctrl-N (while still in insert mode). Assuming you

worked on the method `isEqualTo` last week, it should complete the name for you.

Delete that and type "`ge`" (minus the quotes), then hit Ctrl-N repeatedly (while still in insert mode). Since there are multiple identifiers that start that way, it will cycle through all of them, which should at least include "`getX`" and "`getY`" or their equivalents. 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 keywords (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.

Back to the maze

In the shared course directory, I've put my own implementation of a a program that reads in a maze and writes it back out again (with a little extra info). You should have already looked at that and made some notes about what parts seem less familiar. I'll talk through a few of those in lab today, and more in class tomorrow.

This week's lab is primarily focussed on using that maze code to help you write a `Maze` class. Since you will be using both my code and your own, make sure that both of us are listed as `@author`s in the documentation comments above the code.

Note, by the way, that while you'll need to have a `Location` class that compiles, I don't require (this week) that all the tests actually pass. If you didn't get to all of them, or if you didn't get all of them working, just make sure there's a stub method in there so that the test case will compile—and make a note of it in your readme. The only `Location` methods that need to work correctly *for now* are the constructor and the accessors (`getX`, `getY`).

The file format

Maze files look like this:

```
7 4
#######
#...#o#
#*#...#
#######
```

The first line contains two numbers (the width and height of the maze); subsequent lines contain a map of the maze itself, with each different type of maze content represented by a different character:

walls	`#`	(hash mark)
open spaces	`.`	(period)
start	`o`	(lowercase 'O')
finish	`*`	(asterisk)

Each maze will have exactly one start and exactly one finish; though note that not all open spaces need be reachable from the start, and the finish may also be unreachable.

Building a maze class

As I've said before, our eventual goal will be to write a maze solver, but for this week we'll focus on the following behaviours:

- Constructor sets up this `Maze` to be a model of the maze it reads from the text of a given `istream&`
- `elementAt` looks up the element in this `Maze` that lies at a given valid `Location`
- `inBounds` determines whether a given `Location` is valid for this `Maze`
- `getStart` returns a `Location` indicating the start position of this `Maze`
- `getFinish` returns a `Location` indicating the finish position of this `Maze`
- `print` prints a view of this `Maze` to a given `ostream&`

Some of those descriptions include terms you haven't seen. I'll explain as we go along, but I expect you to follow the design process laid out in Lab 2 (and I won't reiterate the whole thing here, so refer back to it).

Steps 1–4 of the data design (see pp. 3–7 of Lab 2) should be mostly straightforward up to the constructor definition; Note that the values you assign to each instance variable when you declare them will be literals, like `0` or `Location{0,0}`.

Actually do steps 1–4 before you move on. If you would like to type the step 1 stuff, put it in a comment in `Maze.h`. For step 2, draw your example mazes in comments in your `.u` file (i.e. before fully encoding them as examples, which I'll talk about below; but having them in the file will make it easier for me to see them if you screen share). Now, for real, work on steps 1–4. Ask me questions and/or chat with your breakout-room-mate if you're not sure how. SERIOUSLY DO THOSE STEPS NOW—DO NOT MOVE ON UNTIL YOU'VE GOT THEM. It's ok for now if the constructor is only a stub or only partially implemented.

For step 5 (encoding the example), I need to call attention to two new(ish) things:

Then, read through both new things before you try to encode the examples.

Important thing #1: When you have data in string form but would like to treat it as if it came from the keyboard or from a file, you can use something called an `istringstream`. You can see illustrations of its use in the `strdemo` code. Any method that accepts an `istream&` can happily accept your `istringstream` (as well as `ifstream`s, `cin`, and a few more exotic things).

Important thing #2: C++ doesn't let you start a double-quoted string literal on one line and then finish it on the next. But it *does* let you have a string literal extend over multiple lines: if you have two (or more) double-quoted string literals separated by nothing but whitespace, C++ smooshes them all together into a single string literal. This is super-convenient for long strings that you want to visually line up in your code. Like mazes.

So here we go: You've already set up your `.u` file. In the fixture section, type:

```
fixture:
  istringstream sample1str = istringstream {
                      "7 4\n"
                      "#######\n"
                      "#...#o#\n"
                      "#*#...#\n"
                      "#######\n" };
  Maze sample1 = Maze{sample1str};
```

You'll also need to `#include <sstream>` at the top of the test file, and you should have already written the header for the constructor in the `.h` file and a stub for the constructor in the `.cpp`.

As always, verify that it compiles and runs before you move on.

Add the examples you wrote out by hand for step 2 to the fixture. As with the one I gave you, each will set up an `istringstream` with the text view of the maze, then actually make the `Maze` object from it.

Once you've got them in, compile and run the test suite (which still has zero tests, so as long as it doesn't crash we're in good shape). The constructor isn't really fully tested yet, but that's ok for the moment.

Finishing the constructor

The actual job of the constructor will involve a bit more processing than simply assigning a variable or two (like in `Location` or `Card`). That processing should be tested, somehow, but since we haven't done any other methods yet, there's no way to do so! The next method you work on, `elementAt`, will let you access the main contents of the maze, and so the test cases you write for that method will, practically speaking, be testing *both* the constructor and the accessor. In fact, you could jump ahead slightly and do the first parts of the `elementAt` process—up through the test cases—before continuing work on implementing the constructor. Then you'll be able to test the constructor as you go.

Whenever you do get to implementing the constructor, remember that you don't have to do so completely from scratch: you can adapt the code I've given you into the form you'll need for the lab. You'll definitely want to refer back to the code in `mazerw.cpp` as you develop this. Reading other people's code and figuring out how to glue it into your own—modified to fit the task at hand—is a difficult but incredibly important skill that you're developing here.

Method design

For the other methods, you can follow the Lab 2 method design process more explicitly. I've given the descriptions; think carefully about the questions in step 2 of the design process (Lab 2 p. 10) when you're trying to figure out what the methods' headers should be.

Writing your test cases will be much like for Lab 2. For `print`, I'll point out that there are `ostringstream`s as well, which you can see in action in the tests for `strdemo`.

RUBRIC

General

1 Present in lab with comments/questions about preview

1 Readme with required stuff

Maze data

1 Instance variables

1 Additional valid maze examples (≥ 2 that I didn't give you)

Maze methods

1 Constructor correct header and reads in maze♣

1 `elementAt` correct header and good test cases♣

½ `elementAt` definition

1 `inBounds` correct header and good test cases♣

½ `inBounds` definition

½ `getStart` and `getFinish` correct header and test cases♣

½ `getStart` and `getFinish` implementation

½ `print` correct header and good test cases♣

½ `print` definition

♣ indicates point is only available if the code compiles, with at least a stub for the relevant method(s).

Handing in

This week there are a ton of files to hand in, so you should definitely plan to hand in the whole directory. Use the `handin` script and assignment name `lab3`; you should hand in what you have on *next* Wednesday, the 3rd, so I can look it over, but its final due date is 4pm the following Wednesday, the 10th.

(Those of you still needing to work on Lab 1 should find time *soon* to do so—if you haven't gotten to 10 problems working you will begin to fall behind. Ask for help if you need it!)

Don't forget your readme!