Lab 3

11 September 2014

Start a fresh directory for this week's lab. Copy in your maze code from Lab 1 and your location code from Lab 2. 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). As-

suming 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 the program from Lab 1. Edit that directly by typing

```
vim /home/shared/162/lab1maze.cpp
```

Read through the code, and make note of the interesting parts: in the space below, write one column of things that I did differently from your code but you understand it, and one column of things you don't understand or haven't seen before (if any). Include line numbers.

This week's lab is primarily focussed on turning your maze code into a `Maze` class. If your Lab 1 implementation worked (or almost worked), you should use that as your starting point; if not, though, you're welcome to use my implementation instead. If you do so, make sure to include both your name and mine as `@author`s in the documentation comments above the class.

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`).

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 five behaviours:

- EDIT: We'll rely on the implied default constructor: make sure to init your instance variables when you declare them
- `read` modifies this `Maze` to be a representation of the text it reads from a given `istream&`
- `print` prints this `Maze` to a given `ostream&`
- `contains` determines whether a given `Location` is valid for this `Maze`
- `elementAt` looks up the element that this `Maze` that lies at a given valid `Location`

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; EDIT: 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. Use your notebook to write down anything you need to write out by hand (notably step 2, possibly also step 1).

For step 5 (encoding the example), I need to introduce three new things, but before I do that, create a test file for `Maze` and put an empty example into it:

```
Maze sample = Maze();
```

and verify that it compiles before you continue.

Then, read through all three new things before you try to implement them.

New thing #1: Sometimes, when we want to create data examples for a fixture, we can't do the whole setup in the constructor call. A more complete layout of the fixture part of an Unci file is this:

```
fixture:
    // example declarations and constructor calls

    setup
    {
        // additional code to set up examples
    }
```

We'll need this for the mazes, because the constructor doesn't fully set up the example—we'll need to use `read` for that.

New thing #2: 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`. At the top of the file you must `#include <sstream>` and then when you declare the `istringstream` variable, you provide its constructor with the string you want to read from, just like you can provide an `ifstream` with the name of the file you want to read from. Then, any method that accepts an `istream&` can happily accept your `istringstream` (as well as `ifstream`s, `cin`, and a few more exotic things).

New thing #3: 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 declared the variable `sample` in your test file. Immediately below that line, type:

```
setup
{
  string samplestr = "7 4\n"
                     "#######\n"
                     "#...#o#\n"
                     "#*#...#\n"
                     "#######\n";
  istringstream samplein(samplestr);
  sample.read(samplein);
}
```

You'll also need to `#include <sstream>` at the top of the test file; and add the header for `read` to the `.h` and a stub definition for `read` to 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 test file. Each example will have a declaration before the `setup` block, and then a `string`, an `istringstream`, and a call to `read`, just like the code I had you type in.

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 `read` method

I want you to do the `read` method first, because of the fact we need it to fully set up our examples. I gave you a description, and you've already declared it and stubbed it; and because of its constructor-like nature, there's really no separate test cases for it either. So we can proceed straight to defining it, and for that we can reuse a lot of the code we wrote for Lab 1. Inside `Location.cpp`, if you go to the body of the `read` method and type `:r mazeread.cpp` (or whatever you called your Lab 1 file), Vim will read in that file, and you can press `dd` a bunch of times to get rid of the extra lines you don't want.

The main changes you'll be making are:

- Using `Location` objects to store the start and finish

- Switching the data from a 2D array to a 2D vector

I'll let you think about what to do with the `Location` objects. As for 2D vectors:

2D vectors

A 2D vector is just a vector of 1D vectors. In this section I'll give you a few code examples that are *not quite* what you need to type into your own files—**you will have to think about how to adapt them.**

Since the data is a vector of vectors, its declaration would look something like this:

```
vector<vector<int>> data;
```

(although our grid doesn't hold `int` values). If you didn't already set up your maze's stored data to be a 2D vector, you can change that now.

Notice, though, that we have still not specified the size of the grid—vectors are by default initially created with size zero.

Once you know the maze's size (i.e. in the `read` method, after reading the first line of the input), you'll call `resize`. That will look something like this:

```
data.resize(8);
for (int a = 0; a < 8; ++a)
  data[a].resize(6);
```

Of course, you won't want to always create an 8×6 grid—what variables should you use there?

Once all this resizing is done, you can treat your 2D vector exactly like a 2D array. In particular, you can access it as

```
data[a][b]
```

where `a` and `b` are your coordinates (x, y, row, column, whatever).

Mostly, though, `read` will look a lot like the Lab 1 code.

Method design

For the other methods, you can follow the Lab 2 method design process more explicitly. I've given you the descriptions, and they are good descriptions: think carefully about the questions in step 2 (p. 11) when you're trying to figure out what the methods' headers should be.

Writing your test cases for `contains` and `elementAt` will be much like for Lab 2. For `print`, I'll point out that there are `ostringstream`s as well, which I can treat sort of like an output file:

```
ostringstream testout;
testout << "Testing\n";
check(testout.str()) expect == "Testing\n";
```

Of course, you won't want to explicitly use `<<` here, but rather make a call to `sample.print(testout)`. Think about how to make a mental connection between what you did with the `istringstream` earlier, and the test cases you've written before, to figure out how to make this test case work.

RUBRIC

General

1 Present in lab

1 Documentation in readme

Maze data

1 Instance variables

1 Constructor ♣

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

Maze methods

1 `read` correct header and definition♣

½ `print` correct header and good test cases♣

½ `print` definition

1 `contains` correct header and good test cases♣

½ `contains` definition

1 `elementAt` correct header and good test cases♣

½ `elementAt` 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`; this lab is due at 4pm next Wednesday the 17th.

Don't forget your readme!