Lab 1: Maze setup

28 August 2014

Our first long-running project in this course will be a maze solver. Eventually, your program will be able to analyse a maze and animate its solution using different strategies; but we need to cover a bit more material before it can manage all that.

What we can do today is learn or remember how to use a command line, set up the basic file format, and shake some of the rust off our programming skills.

So here's the entire content of today's lab/this week's project: read something that looks like this:

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

from a file named at the command line; store it in a 2D array; and then print it back out again, followed by the coordinates of the start and the coordinates of the finish.

That's it! The entire rest of this handout just gives more detail, some warnings, and a few suggestions on how to approach the problem.

First: the command line

I will demo how to use PuTTY to access torvalds, but for your future reference, the remote login instructions are linked from the course webpage.

Once you are logged in: you're now in your "home directory", which you can list by typing `ls` (that's a lowercase L) and hitting enter. To keep everything tidy in your account, first you probably want to

```
mkdir 162
cd 162
```

to create a directory named 162 and, for now, change to that as your working directory. In the future you won't have to `mkdir`, but you'll `cd 162` every time you log in to work on this course. (You'll want to keep other directories for other courses that you're taking now or in the future.)

These commands and several others are laid out in the "Command line starter kit" page stapled to the back of the lab.

To open the editor for a file called `hello.cpp`, you can now type

```
vim hello.cpp
```

When the vim editor first opens, you are in "command mode". To switch to "insert mode", which is the mode that lets you actually type things in, press `i` once. Then, type in a C++ hello world program.[1] When you are done, or at least ready to compile, press the Escape key to exit from insert mode to command mode, then type `:wq` to write the file and quit the editor.

These commands and several others are laid out in the "vim starter kit" page stapled to the back of the lab.

Finally, at the command line, type

```
compile hello.cpp
```

to compile the file. This is a script I've installed locally to run the C++ compiler with some useful settings; it will show you the exact command that it runs (the line starting with `g++`), followed by either a success message or a compiler error.[2] If there are errors, edit the file again and fix them; if not, the compiler has just created an executable named `a.out`. To run it, type

```
./a.out
```

and verify that it runs correctly.

You're now ready to move on to the actual body of the lab and write the maze reader.

[1] That is, a program that prints "Hello, world!" to the screen and does nothing else. Hello world programs serve to let you check that you understand how editors and compilers work before you worry about writing more complicated programs.

[2] You may be more used to running `g++` directly. It's still possible to do so, but if you use `compile` it runs it using an updated C++ standard, prepares the output for debugging, detects a number of mistakes that novice C++ programmers often make, and limits the output to the first error message—none of which are done by default if you just type `g++`.

Setting up the program

Create a directory inside `162` called `lab1` to hold your work for this lab. You'll need to use `mkdir` to do this (see the front page or the command line reference if you don't remember how). Don't forget to use `cd` to go into the `lab1` directory once you make it!

Before you start writing code, it's generally a good idea to at least start your documentation file. Edit a file that will be called `README.txt` to contain this documentation; at this point you will be able to put your name, the name of the assignment, and a brief description of what it does. Eventually, before handing in, it should (at a minimum) contain instructions on how to compile, run, and test your program.

The input format

Every maze file that your system can handle will have the same format: the first line will contain 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.

The output format

The output of the program should start with a repeat of the input, followed by lines that identify the coordinates of the start and finish. For the example on the first page, that would be something like

```
The start is at (5,1).
The finish is at (1,2).
```

Testing

Here is the example from the first page side-by-side with the expected corresponding output:

CONTENTS OF INPUT FILE

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

OUTPUT

```
7 4
#######
#...#o#
#*#...#
#######
The start is at (5,1).
The finish is at (1,2).
```

Type the maze on the left into a file named `test1.in` . You would now be able to test your program by running

```
./a.out test1.in
```

and then carefully visually inspecting the output to compare it with the contents of the OUTPUT box above. But that's tedious and prone to human error. Instead, I want you to put the OUTPUT box into its own file named `test1.out` (the easiest way is to copy `test1.in` to `test1.out` and then add the extra two lines at the end), and then run

```
./a.out test1.in | diff test1.out -
```

If this command prints nothing, the program output was identical to what was expected—so it was correct. If there were any differences, though, they will be printed.

As usual you should also come up with a few test cases of your own; right now make at least a `test2.in` and matching `test2.out`, and if you think of other interesting examples, add them later. Note that the input file format does not require that the file *end* after the last line of the maze, and leaves the further content of the file unspecified—a good use of this fact would be to write a little comment at the end of the file explaining what interesting case that particular maze is there to test.

Some likely gotchas

Don't forget your `#include` lines, and don't forget about `std`.

At the start of the file, you're reading numbers (`int`s, presumably), but subsequent lines you'll want to read line-by-line using `getline`. What happens to the newline after the second number?

Because of some quirks of 2D array representation in C and C++, you actually won't be able to write functions that pass the 2D array as an argument. We'll see a way around that problem later this week. For now, all the array manipulation (and possibly the entire program) will have to go in `main`.

Tips

Start by creating the smallest program that compiles and runs, and keep adding little pieces that you test by compiling and running. This is general advice for any programming work you ever do!

You could write an algorithm to scan the 2D array in order to find the start and finish, but it'd be easier to just check for them when you're reading in the maze in the first place.

If you don't remember how to do file I/O, the thing you want to look up is `ifstream`.

Never learned how to get strings from the command line (as you'll have to do for the filename here)? If you declare your main function as

```
int main (int argc, char *argv[])
```

then you can refer to the first command-line argument as `argv[1]`, which is a C string (a `char *`).

Handing in

There is a program called `handin` to submit your work. In a window where you are in the directory with your work for Lab 1, you can type, for instance,

```
handin cmsc162 lab1 maze.cpp README.txt test1.in test1.out
```

This line tells the program what course you're handing in for, what assignment you're handing in for, and finally what files to include—use the

filenames you actually created, and make sure you hand in *all* the necessary files.

The lab handin is due Wednesday the 3rd at 4pm. (Labs will, in general, be due the following Wednesday afternoon.)

Rubric notes

Lab grades, including this one, will typically include a small portion (10–20%) for lab attendance and performance and a small portion (10–20%) for documentation and testing; with the rest for correctness.

If your code does not compile, or if it compiles but immediately crashes before producing any output, you will in general receive very few of the remaining points! It is always better to submit a program that compiles and runs, but is incomplete, than to submit a program that "has everything" but doesn't run at all.

Specifically, here is the plan for this week's rubric:

RUBRIC

2 Present in lab

General

1 Handin includes readme file with appropriate documentation

1 Correct file headers and `main`

1 Program compiles and runs

Maze

1 Command line argument and file access

1 Declares 2D array of appropriate size

1 Reads/stores maze

1 Prints maze from array

1 Identifies start/finish and prints coordinates

Extras

When you store the coordinates for the start and finish location, use a `struct` or `class` object to bundle the x and y coordinates together.