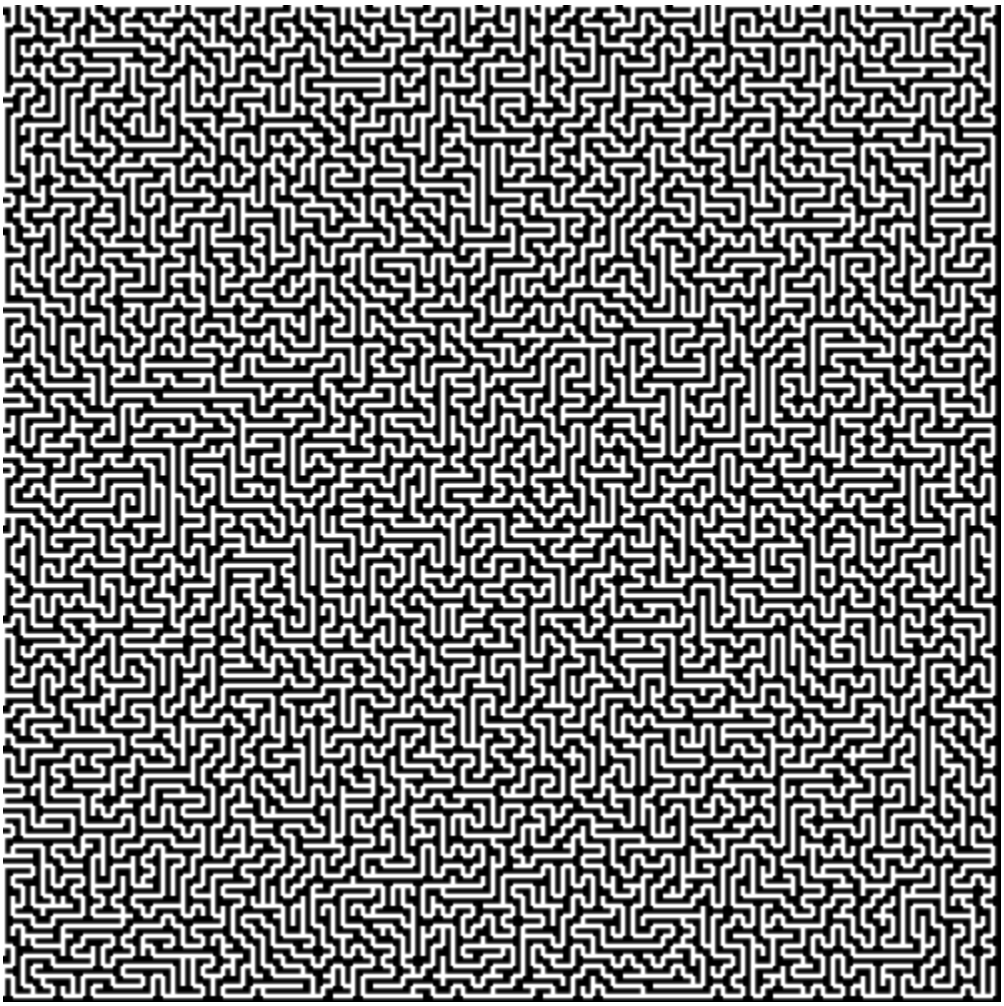
COSC1252/1254 Programming Using C++

Assignment 1: Mazes (part 1)

Due: end week 6, Sunday 26th August 11:59pm



At a minimum you should use the following compile flags for the assignment:
-Wall -pedantic -std=c++14

You must use either c++14 for this assignment.

In order to get these compiler flags to work you must run the following command on titan/jupiter/saturn:

```
source /opt/rh/devtoolset-4/enable
```

If you leave out the word “source” it will appear that the script has not run as it will make no permanent changes to your environment. You must run this command each time you log in before compiling. It would be advisable to put the command in your .bashrc file.

You may do the assignment in pairs or individually. If you complete this assignment in pairs, please only submit the assignment as one student but specify in your readme who the students who contributed were. If you decide to do this assignment in pairs then you would be well advised to use source control (such as git) so it is clear who worked on what components in case there is a dispute.

Overview

This assignment will require you to explore the basics of C++ classes and file I/O by implementing a basic maze loader/generator. The main requirements are:

- Load a maze from a binary file and output it to SVG
- Create a randomly generated maze and save it as a binary, or an SVG file

Details

The assignment shall be done in C++, and must compile on the jupiter/saturn/titan servers using C++14. A large part of the assignment can be completed in the labs, and the exercises are directly related to the assignments.

You may work in pairs or individually.

Maze

The program should store a maze as a grid (2D array or vector of vectors) of Cells, where each Cell stores a list of Cells that it is connected to, as well as its x,y position in the grid. You may also store a list of Edges if you need to, where each Edge references the two Cells that it connects. **The maze path is the white part in the output.**

Maze Loading

The program should be able to load a maze from a binary maze file, which has the following format:

- int: width of grid

- int: height of grid
- int: no. of edges in maze
- edges

Where each edge stored has the format:

- int: x location of Cell A
- int: y location of Cell A
- int: x location of Cell B
- int: y location of Cell B

Maze Saving (Binary)

Your program should be able to save any maze it has stored as a binary file with the same format as given above. A binary file is a file where the data is stored as it would be stored in memory rather than as strings. Eg the number 5 is written to disk as a 32 bit integer as 00000000 0000000 0000000 00000101 rather than as “5”. **Mazes stored as ascii will get no marks for this requirement.**

Maze Saving (SVG)

Your program should be able to save any maze it has stored as an SVG file. The SVG should have a black background, and should render a white line for every edge in the maze. An example 2x2 maze could look like:

```
<svg viewBox='0 0 1 1' width='500' height='500' xmlns='http://www.w3.org/2000/svg'>
        <rect width='1' height='1' style='fill: black' />
        <line stroke='white' stroke-width='0.005' x1='0' y1='0' x2='0.5' y2='0'/>
        <line stroke=white stroke-width='0.005' x1='0.5' y1='0' x2='0.5' y2='0.5'/>
        <line stroke='white' stroke-width='0.005' x1='0.5' y1='0.5' x2='0' y2='0.5'/>
</svg>
```

Maze Generation

Your program should also be able to generate a maze using the Hunt-and-Kill method. In this method, your program will start an arbitrary location in the grid. Then it performs a random walk, carving passages to unvisited neighbors until the current cell has no unvisited neighbors. Enter “hunt” mode, scanning the grid for an unvisited cell adjacent to a visited cell. If found, carve a passage between these two cells, and let the formerly unvisited cell be the new starting location. The process repeats until all the cells in the grid have been visited.

Details available at:
http://weblog.jamisbuck.org/2011/1/24/maze-generation-hunt-and-kill-algorithm

All random choices should use the mersenne twister algorithm available in the standard library from the class std::mt19937.

Controls

Your program should be able to run from the command line, with arguments to specify how the program should run, ie:

./mazer --lb filename.maze --sv filename.svg # load binary file and save svg file
./mazer --g seed --sb filename.maze # generate with seed value, save binary file
./mazer --g seed --sv filename.svg # generate with seed value, save svg file
./mazer --g seed --sb filename.maze --sv filename.svg # gen with seed, save binary, save svg

You should also be able to specify the width/height of the maze when generating (but not when loading), ie:

./mazer --g seed width height …

When generating a maze, the seed is a number that determines what the sequence of psuedorandom numbers to be generated will be. When using the –g flag, the seed is optional. If it is not provided you should base it on the current time and output it so that on a future run, a user can replicate the results.

Note that, as above, command line args will be given in the order they should be processed, and an appropriate error message should be given if inappropriate args are entered. You should reject commands that do not make sense such as:

./mazer --sb filename.maze --g 1230465

should be rejected as when the request to save the file is given there has been no request to load or generate a maze before saving it.

Please note that you are required to also validate all mazes loaded into memory – if a binary file has vertices outside the range of the maze for example, it should be rejected.

Class Diagrams

Prior to submission you will draw a class diagram representing your implementation. In this diagram you should represent any datastructures and fields as well as the major public methods of the class.

At this point there is no requirement to represent inheritance relationships in the class diagram as that is not a requirement of assignment 1. This class diagram will represent 10% of your mark for this assignment.

Make or CMake

You are required in this assignment to use either make or cmake (generating makefiles in a portable way) to separately build each source file and then link them together into an executable. The executable created must be called mazer. You should already be familiar with how to write a makefile. Cmake requires you to write a CMakeLists.txt file which details how to build the program. You can use cmake to create the unix makefiles for the project as well as to generate build files for a range of different IDEs. Using CMake is a preferred approach over the use of Makefiles but for this iteration of the course, we are giving you a choice.

README

Your readme file should outline how to use the program, how to compile the program and how to run the program at a minimum. If there are bugs or unresolved issues, these should also be details in this file. Your readme should also outline the process you went through to develop the program.

Submission

Submit assignments under assessment tasks on canvas via MyRMIT. Assignments can be submitted prior to the assignment due date.

Submit all of your source code, Makefile or CMakeLists.txt file and README file. Create a zip, gzip or tar archive and submit that as a single file. Do not include object files or executables. Assignments will be compiled by markers on the jupiter/saturn/titan servers so make sure you test your assignments on those servers. If you are working in pairs only one person needs to submit, but make sure to include both student numbers in the README file. In the case of both students submitting, the latest assignment submission will be used for marking.

A few notes:

- Markers can miss functionality if the default controls are changed. Make sure alterations and additions are listed in the readme. Document any variations but please discuss such variations with me before you implement them as this will avoid misunderstandings.

- If there is a bug which can be avoided in certain circumstances, describe how, as well as any workarounds that you have used - if it appears to us that you have not implemented a requirement correctly, we will mark accordingly.
- If you have any code that breaks your program, please comment it out. However, if a feature is partially working, leave it in and add a note in the readme as to what is working and what is not working.
- When commenting code focus on complex areas. Not only does this improve readability, but it shows you know what you've coded. It is advisable to carefully name identifiers so that they are self documenting as well.
- You should be writing code that is modular and easily maintainable. Code where this is not the case will receive a mark deduction.
- No lines of code should be longer than 80 characters. Properly structured and indented code has less bugs and receives more marks. Don't mix tabs and spaces.
- Submitting early is always a good idea. You can resubmit, the last submission will be the one used for assessments.
- You are encouraged to use other libraries provided they are installed on the server. In particular, you are encouraged to use the boost library but please ensure you use the same version as that on the server and only use the components available there - version 1.53.
- If you're having trouble submitting close to the due date, contact Emily.
- **If you consistently use modern features of c++14 as outlined in the tutorials you will gain an additional 5 marks towards your assignment mark.**

Marking Guidelines

A focus on good quality coding standards is expected as well as implementation of the algorithms. A basic implementation of the algorithms without focus on coding standards or software design / validation should expect a pass mark but not much more than that.

Some hallmarks of a high distinction implementation include:

- Separate classes for entities that are logically separate. You will be marked on your choices here.
- Good use of appropriate C++ containers
- An understanding of the C++ language as taught in this course – procedural C implementations of this assignment will attain a poor mark.
- Minimising wasteful or inefficient code
- No memory leaks
- Good commenting – each function should be commented as should the internals of more complex functions. In particular, any algorithmic complexity should be commented upon.
- Good makefile/CMakeLists.txt files – that is, compile each class separately then link them together. Have a clean target that removes all files created as part of compilation.
- Please note that roughly 50% of the marks will be allocated to your program performing the tasks correctly. 40% of the marks will assess the quality of your implementation, good software design and object orientation and good coding standards. Finally, 10% of the marks will be allocated to your class diagrams.

Marks are deducted for poor coding conventions/practices such as:

- use of global variables.
- use of goto statements.
- inconsistent use of spaces or tabs for indentation. We recommend a minimum of 3 spaces for every level of indentation. Be careful to not mix tabs and spaces. Each “block” of code should be indented one level.
- having line lengths of code that are beyond a reasonable maximum such that they fit in the default xterm screen width (80 characters wide).
- Not identifying yourself in some files – this is your work, you don't want others to steal it.
- Use of inappropriate identifier names.
- Use of magic numbers.
- Use of platform specific code with system() or system specific header files such as windows.h or unistd.h.
- No modern C++ features.