P1

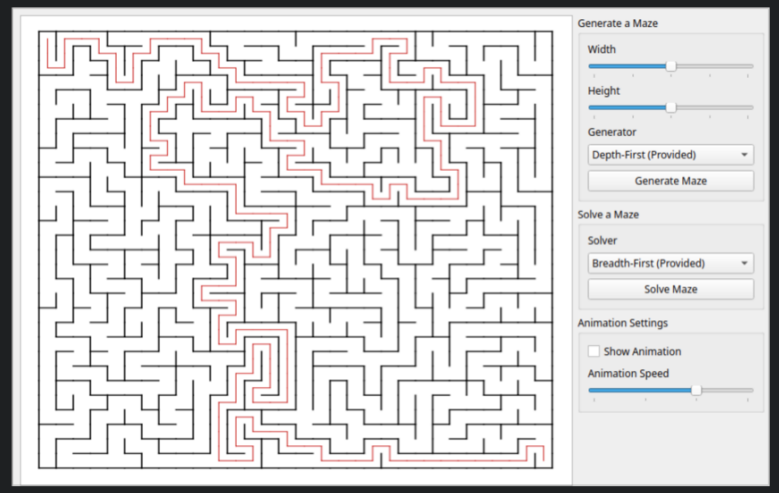
- This project asks you to implement one or more classes in C++ that are capable of generating two-dimensional mazes of arbitrary size, along with one or more classes in C++ that are capable of solving them. The goal is to provide you with more practice and a fuller understanding of how to use recursion to solve real problems, as at least one of your generators and at least one of your solvers is required to use a recursive /*depth-first*/ algorithm. It will also provide you with an opportunity to make heavy use of pre-existing classes for which you have no source code, and for which only part of it will have value to you; understanding how existing code works and determining what parts of it can be applied to solve your own problems are important real-world programming skills that you'll need to employ as you move from "sanitized" coursework to real-world work, so I'd like to help you to develop those skills here.

- More specifically, here's what you'll find in your project directory:
- A directory called *lib*, in which there are two precompiled libraries that make up the part of the project that you won't be implementing yourself.
  - *libdarkmaze.so* contains implementations of maze-related concepts like mazes, maze solutions, verifiers, and so on.
  - *libdarkui.so* contains the implementation of the application's graphical user interface. You will not have to implement the GUI yourself; it is being provided, in its entirety, in this library.
- Note that these libraries are being provided in a compiled form without source code, though you'll find declarations of some of their classes elsewhere (see below). There are separate versions of the two different VMs. Note, also, that these libraries are likely only to be compatible with the VM, so you are unlikely to be able to do your work anywhere except using the VM.
- A directory called *include*, in which you'll find three directories:
  - *darkmaze*, which contains declarations of classes exported by *libdarkmaze.so*. You will need to include some of these files in your own header and source files, though you will not need all of them; it's up to you to decide which of these is relevant to your work.
  - *darkui*, which contains declarations of classes exported by *libdarkui.so*. This is not something you're likely to need, as the only place this is likely to be useful is in the application's *main()* function, which has already been written.

- A directory called *app*, in which the application's *main()* function resides. This has already been completed; you shouldn't have to modify it.
- A directory called *core*, in which you'll write your maze generators, maze solvers, and any additional code for this project.
- A directory called *exp*, in which you'll write any experiments that you'd like to write outside of the context of the GUI. The file *expmain.cpp* is the entry point for the experiments (which you run with the command *./run exp*).
- A directory called *gtest*, in which you'll write any unit tests that you'd like to write. The file *gtestmain.cpp* is the entry point and has already been written; all you need to do is create new source files and place them in the *gtest* directory, then rebuild, and they will be executed automatically when you issue the command *./run gtest*.

The application

- Your work on this project begins with an already-existing, already-working application with a graphical user interface (GUI) that can display a maze and its solution, and can also animate the process of generating and solving a maze. The GUI window looks like this:



- The large area with a white background is where a maze and its solution are drawn. Initially, this will be an empty area with a white background; when you generate or a solve a maze, the result will appear within that area. In the example above, both a maze and its solution are displayed.
- Along the right-hand side of the window are a set of controls, allowing you to:
- Choose the width and height of the maze you'd like to generate. The range, not shown numerically, is from 10-50 cells wide and 10-50 cells tall.
- Choose what algorithm you'd like to use to generate a maze. There are three algorithms provided (and you'll write one or more additional algorithms):
  - *__Depth-First (Provided)__*, which is a recursive, depth-first maze generator like one that you'll be building.
  - *__Kruskal's Algorithm (Provided)__*, which uses a well-known algorithm called Kruskal's algorithm to randomly remove walls so long as they do not cause the maze to become imperfect (i.e., introduce two separate paths connecting any two cells).
  - *__Boo's Algorithm (Provided)__*, which uses a newly-invented algorithm called Boo's algorithm to generate a maze that, while not technically perfect, looks pretty good nonetheless.
- Generate a new maze, which will clear out any existing maze and its solution.
- Choose what algorithm you'd like to use to solve a maze. There are two algorithms provided (and you'll write one or more additional algorithms):
  - *__Breadth-First (Provided)__*, which uses a breadth-first approach to solving a maze.
  - *__Depth-First (Provided)__*, which uses a recursive, depth-first algorithm for solving a maze like one that you'll be building.

- Control whether or not the process of generating and solving mazes will be animated (i.e., each step will be shown individually, as opposed to only seeing the final result) and, if so, at what speed the animation will progress.
- Just below the display of the maze and its solution is a line of text that displays various messages. When you first start the program, it says *Welcome!*. When a maze generator or maze solver finishes, this message will tell you about the result – in particular, whether a generated maze is perfect, and whether a solution is complete and correct – which is a good way to verify that your algorithms are doing what you expect them to do.
- Note, also, that you can stop a maze generator or maze solver while it's running by clicking the *X* at the top-right corner of the window. Normally, that *X* closes the window, but if a maze generator or maze solver is running, it simply cancels the operation in progress instead.

The requirements

- This project requires you to complete two tasks:

1. Write a maze generator that uses a recursive, depth-first algorithm to randomly generate a maze of arbitrary size, with the result required to be a /*perfect maze*/.
    - You can also optionally write as many additional maze generators as you'd like, with no limitations on what algorithms or techniques you use, and with no limitation that the result be a perfect maze. Feel free to do anything you'd like and let your creativity run wild.
2. Write a maze solver that uses a recursive, depth-first algorithm to traverse and solve a maze of arbitrary size, with the solution extending from the maze's starting cell to its ending cell without crossing any walls.
    - You can also optionally write as many additional maze solvers as you'd like, with no limitations on what algorithms or techniques you use, and with no limitation that the result be a correct, complete maze solution. Feel free to do anything you'd like and let your creativity run wild.

- Note that you are not required to write any code in *exp* or *gtest*, but you are welcome to write anything you'd like that helps you to isolate issues and test your work.

A quick note about extra credit

- While we encourage you to explore as many maze generators and maze solvers as you'd like, be aware that we are not offering extra credit for generators or solvers beyond the one of each that you are required to implement. You can receive a perfect score on this project while implementing only a single maze generator and a single maze solver, and writing additional ones will not improve your score, but they can be a lot of fun to build!

Generating a maze

- Each maze generator needs to be written in its own class. So, to write a maze generator, create a new class in the *__core__* directory of your project directory, declaring the class in a header file and defining its member functions (and other source code) in a corresponding source file.
- The GUI automatically displays all of the maze generators that are compiled into the program, but only if you follow a couple of rules to help the GUI find and create them:
- You must derive your class from the abstract base class *__MazeGenerator__*, which is declared in a file *__MazeGenerator.hpp__* in *__include__/*darkmaze*. *(You can include this file by simply saying *#include "MazeGenerator.hpp"*, since the compiler has already been configured to look in the *include/__darkmaze__* directory for header files.)* Deriving from this class obligates you to provide an override for this virtual member function (which is declared as a pure virtual function in *__MazeGenerator__*):
  void generateMaze(Maze& maze) override;

- where *__Maze__* is declared in the file *__Maze.hpp__*, also in the *__include/darkmaze__* directory.
- You must be sure that your class has a default constructor (i.e., a constructor that takes no parameters). Most likely, you won't implement a constructor in your class at all, since the default is probably going to be fine; if you do implement your own constructor, though, be sure you implement at least one that takes no parameters.
- In the source file – not in the header file, as it's important that this only be executed once – you'll need to do two things:
  - Write this include directive near the top:#include <factory/DynamicFactory.hpp>

- Write this line of code somewhere after that include directive (and somewhere after you've included the header file corresponding to your source file, so that the declaration of your class will have been seen already):

IDYNAMIC_FACTORY_REGISTER(MazeGenerator, *name of your class*, "*display name*");

The required algorithm

- The required algorithm must generate a *perfect maze*. Viewing a maze as a two-dimensional matrix of square cells, a perfect maze is one in which any two cells are connected by a single unique path. An important consequence of a maze being perfect is that all cells in a perfect maze are reachable from the starting point by some unique path, meaning that perfect mazes are guaranteed to have a solution. They're also guaranteed to have a unique solution, which makes them more interesting to solve.
- To generate a perfect maze, you'll use a recursive algorithm to "dig tunnels" of various lengths. It starts with a maze in which all of the possible walls exist (i.e., a wall exists on every side of every cell), then continues removing walls until a perfect maze has been constructed. Naturally, it requires some care not to remove walls that would cause the maze to be imperfect; in our tunnel-digging algorithm, we have to be sure we stop digging before we knock out walls that would lead to places we've already been.
- The algorithm works, then, by starting at a particular cell (and it doesn't matter, ultimately, which cell you start from), and does the following:
- Mark the current cell as "visited."
- While the current cell has any adjacent cells that have not yet been visited…
  - Choose one of the unvisited adjacent cells at random. Randomness is important here, or your algorithm will always generate the same maze.
  - Remove the wall between the current cell and the cell you just chose.
  - Recursively call this algorithm, with the chosen cell becoming the current cell.
- As you generate your maze, you'll need to call member functions on the *Maze* object that was provided as a parameter. Don't assume anything in particular about that *Maze* object, other than it has the correct width and height; make any changes you need to make in order to achieve the correct result, and make sure it works regardless of what walls are in place (or not in place) when your algorithm is called.

- The animation in the GUI is automatic; if animation is selected in the GUI, any change you make to your maze will result in the GUI window being redrawn, so you won't need to do anything special to accommodate that feature. (Among other things, this will help you to visualize your own algorithm's progress, which might help you to determine whether it's correct, and also to debug it.)
- You can write as many other maze generators as you'd like, by following the same steps (i.e., creating a separate class that derives from *MazeGenerator*, registering it with the *DynamicFactory*, giving it a display name, etc.). All of the maze generators you write should show up in the GUI if you set them up right.

Naming your required maze generator so we can find it

Each of your maze generators has a /*display name*/, given as a string literal as the third parameter in the call to the ***DYNAMIC_FACTORY_REGISTER*** macro. So we know which one of your maze generators is the required one (and, thus, the one we should grade), you /*must*/ choose a display name for your required maze generator that has the parenthesized word ***(Required)*** on the end of it (similar to how the provided generators have the parenthesized word ***(Provided)*** on the end of their names); capitalization and the parentheses are important here.

- Otherwise, you can name your required generator anything you'd like, and you can name any other generators in any way you'd like, except they should /*not*/ have the word ***(Required)*** on the end of them. And, of course, /*none*/ of your generators should have the word ***(Provided)*** on the end of their names, since none of yours were provided to you by us.

Solving a maze

- Each maze solver needs to be written in its own class. So, to write a maze solver, create a new class in the ***core*** directory of your project directory, declaring the class in a header file and defining its member functions (and other source code) in a corresponding source file.

- The GUI automatically displays all of the maze solvers that are compiled into the program, but only if you follow similar rules to those you followed for your generators:
- You must derive your class from the abstract base class *MazeSolver*, which is declared in a file *MazeSolver.hpp* in *include/darkmaze*. *(You can include this file by simply saying *#include "MazeSolver.hpp"*, since the compiler has already been configured to look in the *include/darkmaze* directory for header files.)* Deriving from this class obligates you to provide an override for this virtual member function (which is declared as a pure virtual function in *MazeSolver*):
  void solveMaze(const Maze& maze, MazeSolution& mazeSolution) override;

- where *Maze* is declared in the file *Maze.hpp* and *MazeSolution* is declared in the file *MazeSolution.hpp*, also in the *include/darkmaze* directory.
- You must be sure that your class has a default constructor (i.e., a constructor that takes no parameters). Most likely, you won't implement a constructor in your class at all, since the default is probably going to be fine; if you do implement your own constructor, though, be sure you implement at least one that takes no parameters.
- Register your class with the DynamicFactory, just as you did with your maze solver:
  DYNAMIC_FACTORY_REGISTER(MazeSolver, ***name of your class***, "***display name***");

The required algorithm

- The required algorithm must solve the maze using a recursive algorithm with /*backtracking*/. A backtracking algorithm is one that recursively investigates all of the possibilities by moving down a path that hopefully leads to a solution and then, if that path fails, backing up to the nearest place where some untried alternative is available and trying another path. While you could potentially implement an algorithm like this iteratively, it turns out to be a lot less work to do so recursively, as the process of recursion will naturally and automatically manage details that you would otherwise have to manage yourself.
- I'll leave the details of this algorithm as an exercise for you to figure out. If you understand the maze-generating algorithm above, it should not be a big step to design the maze-solving algorithm.

- As your algorithm seeks a solution, you'll need to call member functions on the *__Maze__* and *__MazeSolution__* objects that were provided as parameters, though note that the *__Maze__* has been passed as a constant (because you shouldn't have to change a maze in order to solve it).
- The animation in the GUI is automatic; if animation is selected in the GUI, any change you make to your maze solution will result in the GUI window being redrawn, so you won't need to do anything special to accommodate that feature. (Among other things, this will help you to visualize your own algorithm's progress, which might help you to determine whether it's correct, and also to debug it.)
- You can write as many other maze solvers as you'd like, by following the same steps (i.e., creating a separate class that derives from *__MazeSolver__*, registering it with the *__DynamicFactory__*, giving it a display name, etc.). All of the maze solvers you write should show up in the GUI if you set them up right.

Naming your required maze solver so we can find it

Each of your maze generators has a /*display name*/, given as a string literal as the third parameter in the call to the *__DYNAMIC_FACTORY_REGISTER__* macro. So we know which one of your maze solvers is the required one, you /*must*/ choose a display name for your required maze solver that has the parenthesized word *__(Required)__* on the end of it (similar to how the provided solvers have the parenthesized word *__(Provided)__* on the end of their names); capitalization and the parentheses are important here.

- Otherwise, you can name your required solver anything you'd like, and you can name any other solvers in any way you'd like, except they should /*not*/ have the word *__(Required)__* on the end of them. And, of course, /*none*/ of your solvers should have the word *__(Provided)__* on the end of their names, since none of yours were provided to you by us.