Project 2: A-maze-ing Race

Due: Sunday, 3/21/2025

You are allowed to work on this project in teams of two. The main goal of this project is to use data structures to implement algorithms. Concepts you will be familiar with after this project include:

- Stacks, Queues, and Dictionary
- Depth-First Search
- Breadth-First Search
- debugging (expect a fair amount of that!)

You can download all the files containing the starter code for this project.

In this project, you will implement two search algorithms—depth-first search and breadth-first search—for finding a path through a maze.

Search Algorithms

Below is the pseudo-code for the search algorithm:

```
Algorithm MazeSearch:
   mark Start as visited // add the start to the dictionary as its own previous
   add Start to DataStructure // either stack (for DFS) or queue (for BFS)
   while (DataStructure is not empty):
      Current = remove location from DataStructure
      if (Current == Destination): // found a path!
         build the Path found by tracing back previous pointers in the dictionary
         return Path
      else
         get the list of neighbors of current
         for each Neighbor of Current:
         if Neighbor not visited: // you can tell by checking if Neighbor can be found in the dictionary
            record that Neighbor has Current as its previous
            add Neighbor to DataStructure
   return no Path exists
```

The way this algorithm traverses through locations depends on which data structure is used to store visited locations. When we use a stack, the algorithm is called **Depth-First Search (DFS)**. When a queue is used, it is commonly called **Breadth-First Search (BFS)**.

In this project, you will implement both **DFS** and **BFS** to find paths through a 2-D maze and explore how the two search algorithms produce different paths.

Starting Code

Below is an overview of the files required for submitting this project. Those highlighted in **blue** will require implementation on your part. Those highlighted in **black** are complete and should not be modified except for comments. In addition, you need to include a README in your submission.

- For this project, you must use the Standard Template Library's implementation of **vectors**, **stacks**, **queues**, and **unordered_map**. Click on the links to get the web page describing all their methods. **You will not have to implement these data structures for this project.**
- **position.h**, **position.cpp**: The Position class, which keeps track of the contents of a position in a maze.
- **maze.h, maze.cpp**: The Maze class, which is used to represent a maze and provide solutions to it.
- **mazeUtils.h, mazeUtils.cpp**: definitions for Maze helper functions that load a maze from a file and render an answer.
- **main.cpp**: the main program that loads a maze from a file, solves it, and prints out the result.
- **Makefile**: The build instructions for your project.
- **README**: description of the project, list of files, dependencies, how to compile & run, sample runs, etc.
- **labyrinths.zip**: a .zip file containing several maze files. Their format is described below. Copy all these files in the *cmake-build-debug* folder of your project.

Programming Requirements

For each .cpp file, read the corresponding .h file very carefully. The comments that precede each function and method give you precise information about what the function or method is supposed to do.

The Maze Layout

Each maze is a rectangular grid. Each space in the grid is assumed to be connected to the adjacent spaces (left, right, up, and down, but **not** the diagonally adjacent spaces. The starting point of each maze is in the upper left corner; the exit is in the bottom right corner.

The layout of a particular maze will be stored in a file with the extension **.map**; you can find several examples in your **labyrinths.zip**. A map file contains the following information:

- The width of the maze in squares (i.e., the number of columns in the map)
- The height of the column in squares (i.e., the number of rows in the map)
- For each row of the map, a string describing all of the spaces in that row.
    - Each **#** character is a wall.
    - Each **.** character is an open space.

For instance, the following is a valid map with 5 columns and 3 rows:

```
5 3
.#.#.
.#...
...#.
```

We have given you the code that loads files in this format; this explanation is just for reference.

The Position Class

We will represent a maze as a two-dimensional grid of pointers to **Position** objects. Each **Position** object contains relevant information for one place in the maze: the (X, Y) coordinates, where (0,0) is the upper-left and **X increases as you move right, Y increases as you move down**) and whether that position is a wall. The constructor of the **Position** class should take the X and Y coordinate values and initialize the other fields to suitable default values.

You will have to write a method **to_string()** in this class. The function should unambiguously convert the coordinates of the Position object into a string. For example, the Position object at coordinates (3,51) should be converted to a different string from the Position object at coordinates (35,1). So make certain that you add at least one character (e.g., comma) between the X and Y coordinates.

The Maze Class

A **maze** object represents our knowledge of a particular maze. We can use this information to find a path through the maze using various search algorithms. You will write two methods—**solveDepthFirstSearch()** and **solveBreadthFirstSearch()**—which are very similar. These methods will return a **vector<Position*>**: a vector of **Position** pointers, which constitute a correct path through the maze **from the start (upper-left corner) to the finish (bottom-right corner)**. If no such path exists, the method should return an empty vector.

The 2-D grid that you create for your maze should take the form of a data member **positions** of type **Position***. This is an array of arrays of **Position** pointers. This way, you can write e.g. **positions[0][0]** to get the **Position*** which points to the **Position** object in the upper-left corner, and you could call one of its methods as follows: **positions[0][0]->getX()**. The constructor should allocate the 2-D grid of **Position*** and create a new **Position** object for each element of the grid.

You have been given the declaration of the **maze** class. You only need to implement each of the methods. The real magic of making the program work happens in these methods, which find a path through the maze using the search algorithm given above.

In the maze solver functions, you will need a map (dictionary) to keep track of which Positions have been visited and from which other Positions they were visited. You will need a dictionary that maps strings (keys) to Position pointers (values). This should be implemented using an **unordered_map**. The reason why the keys in this map are strings is complicated, but long story

short, it will work better this way. To map a Position to a string, you will need to program a **to_string()** method in the Position object that will return an unambiguous string representation of the coordinates of the Position object.

Testing the Maze class

Make sure to run tests on your code as you develop. You need to test your code once you've finished the **Maze** class! It will be easier to find bugs in your Maze implementation by direct testing than it will be by trying to find them by hand running the **maze** program.

Your code should pass all the tests from the file **test.cpp** included in the starter code.

Important note: since both **test.cpp** and **main.cpp** contain a function called **main**, only one of them should be in the file **CMakeLists.txt** when you compile your code.

Some of the map files in **labyrinths.zip** have multiple possible solutions. As a result, your output may differ from the provided solutions depending on how you explore the maze. **To get the same results as the provided solutions, you should explore neighboring spaces in the following order: up, left, right, and then down.**

Implementing Main

The implementation of **main** should be less work than the **Maze** class. It just involves assembling the pieces you've constructed above. The **loadMap** and **renderAnswer** functions have been written for you. **loadMap** will read a map file and return a **Maze*** if loading was successful. Otherwise, it will throw a **runtime_error**. **renderAnswer** should produce a string containing a solution, which looks like the map from the input file but contains **@** characters on the path you found. For example, here is a possible execution of the program:

Sample Run 1:

```
$ ./maze
Welcome to the A-maze-ing Race.
Please enter your maze file:                    cycle.map
Which search algorithm to use (BFS or DFS)?     DFS
What is the name of the output file?            Solutions.txt

Loading cycle.map…
5 5
..###
#...#
#.#.#
#....
####.

DFS Searching…
```

```
@@###
#@..#
#@#.#
#@@@@
####@

Path length: 8
# of visited nodes: 9
```

Sample Run 2:

```
$ ./maze
Welcome to the A-maze-ing Race.
Please enter your maze file:                    cycle.map
Which search algorithm to use (BFS or DFS)?     BFS
What is the name of the output file?            Solutions.txt

Loading cycle.map…
5 5
..###
#...#
#.#.#
#....
####.

BFS Searching…
@@###
#@@@#
#.#@#
#..@@
####@

Path Length: 8
# of visited nodes: 12
```

Invalid Inputs: Your maze program should gracefully handle the following cases of invalid input:

- The user provides a search type other than "**BFS**" or "**DFS**"
- The user provides an invalid maze file (to be handled by catching the exception from **loadMap**)

Summary

To summarize the requirements of this project:

- You should have a working maze class.
- You should have a main function that provides a simple interface to solve maze problems.
- The input should be validated, and your code should contain no memory errors.

Submit

Submit a .**zip** or **.tar** containing your entire project folder for the assignment using the submission link on Canvas. Your submission should be named **firstname_lastname_p2.zip** or **firstname_lastname_p2.tar**. If you work in a team of two, you should concatenate the first and last name of each member.

Notes:

- Please keep the **Academic Integrity Policy** in mind---do not show your code to anyone and do not look at anyone else's code for this project. If you need help, please contact the instructor early.
- Include the basic header in your program as below. **Submissions without the Honor Pledge will get a zero.**

```
# Assignment: (Assignment Number):
# Author(s):
# Due Date: (Due Date)
#
# Description: Write a small blurb about what this project's goals are
#            and what tasks it accomplishes
# Comments: (Explain any known issues or questions if any)
# Honor Pledge: I have abided by the Wheaton Honor Code and
#                 all work below was performed by (Your Name(s)).
```

- Make sure you either develop with or test with CLion (to be sure it reports no errors or warnings) before you submit the program. Document any known issues in the comment section above.
- Make sure to document each method and adhere to the Google C++ Style Guide for documentation. When you write source code, it should be readable and well-documented. Each method should have a comment section, including what it returns and the types of parameters that are passed.

```
/* Comment for each function should include the following items
 * A brief description of the function.
 * @param: var, datatype, description
 * @return: var, datatupe, description
 * dependencies: other major member functions invoked in this function.
 */
```

Acknowledgment

This project write-up is based on the Labyrinth lab developed by the Computer Science Department of Swarthmore College and edited by Martin Gagne and Tony Tong at Wheaton College.

Grading

Grades will be given according to the following policy.

Required functions & test functions	**Points**
Correct implementation of the required classes and functions • main.cpp • Maze class (BFS, DFS) • Position class • myDictionary • File input & output • Additional helper functions	50
Testing: • User-friendly interface to test both BFS and DFS with different inputs • Exception handling (invalid option, filename etc.) • Pass all tests in test.cpp	30
Documentation: filename, program header comment, coding style (variable naming, indentation, comment, readability, etc), a hard copy of your code (signature next to the honor pledge) **README**: include the following information • **Project title**: simple overview of use/purpose • **Authors**: list of contributors for the projects • **Description**: an in-depth paragraph about your project • **How to Run**: step-by-step bullets to show how to run/test your projects with sample input(s) and sample output(s). • **Known Issues: any known issues with the code** • **Help**: any advice for common issues or problems • **Version History**: basic git log if applicable Submit one .**zip** or .**tar** file per group containing all your source code and **README** for this assignment via the submission link on Canvas.	20
Total	**100 + 5**