Programming Assignment #3

1. **This program tests the concepts of:**
   Stacks, Template classes

2. **Program Objective:**
   The purpose of this assignment is to get some practice working with stacks. It is highly recommended that you work out a design to solve this problem on paper first, before writing any code.

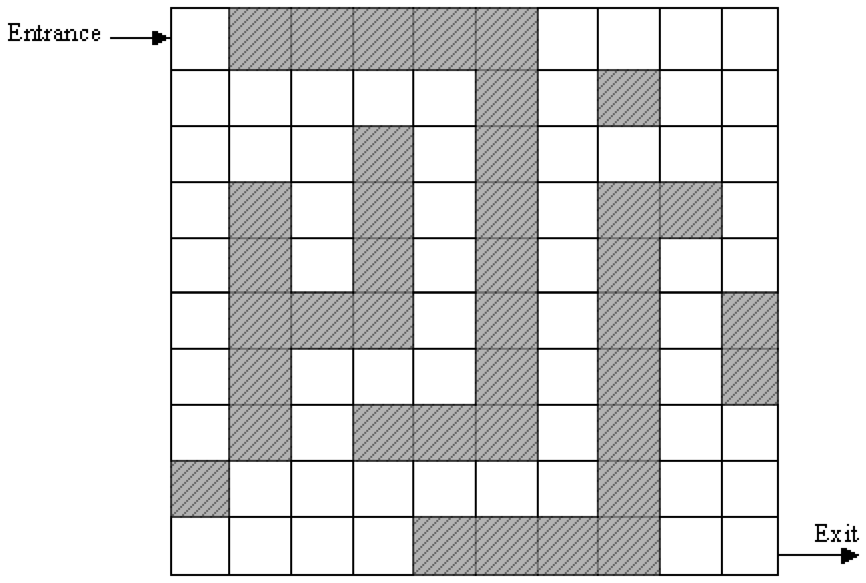
   Submitted files need to be named as follows:

File	Format	Example
Main	lastname_maze.cpp	miller_maze.cpp

3. **Description:**
   In this assignment you will simulate a rat-in-a-maze, and you will use a Stack to a path through the maze. You will use the STL stack data structure.

   A maze is a rectangular area with an entrance and an exit. The interior of the maze contains walls or obstacles that one cannot walk through. In our mazes these obstacles are placed along rows and columns that are parallel to the rectangular boundary of the maze. The entrance is at the upper-left corner, and the exit is at the lower-right corner.

   Suppose that the maze is to be modeled as an *n* by *m* matrix with position (1,1) of the matrix representing the entrance and position (*n*,*m*) representing the exit. *n* and *m* are, respectively, the number of rows and columns in the maze. Each maze position is described by its row and column intersection. The matrix has a 1 in position (*i*, *j*) if there is an obstacle at the corresponding maze position. Otherwise, there is a 0 at this matrix position. Below is an illustration of a sample maze and a corresponding matrix representation of that maze.



Programming Assignment #3

Matrix Representation of Sample Maze

```
0 1 1 1 1 1 0 0 0 0
0 0 0 0 0 1 0 1 0 0
0 0 0 1 0 1 0 0 0 0
0 1 0 1 0 1 0 1 1 0
0 1 0 1 0 1 0 1 0 0
0 1 1 1 0 1 0 1 0 1
0 1 0 0 0 1 0 1 0 1
0 1 0 1 1 1 0 1 0 0
1 0 0 0 0 0 0 1 0 0
0 0 0 0 1 1 1 1 0 0
```

The rat-in-a-maze problem is to find a path from the entrance to the exit of the maze. A *path* is a sequence of positions, none of which is blocked, such that each (other than the first) is the north, south, east or west neighbor of the preceding position.

You are to write a program to solve the rat-in-a-maze problem. You may assume that the mazes for which your program is to work are sufficiently small so that the entire maze can be represented in the memory of the target computer. Your program will be read in command line parameters containing filenames of maxes to process. The first line of the maze file will be ROW_COUNT COLUMN_COUNT. The proceeding lines with be a matrix of 1's and 0's depending on if it is blocked or not. See one of the provided maze text files. You can assume that the input files have correct syntax. The command line arguments will be received as follows:

mazefile1 [mazefile2] ... [mazefilen]

If more then one file argument is provided, your program will process multiple mazes during a single execution.

4. **Hints:**
There are three parts to this problem as follows:

    - Parse maze from file
    - Find a path through the maze.
    - Output the path, or a message if no path is found.

    To find the path through the maze you may proceed as follows: Begin with the entrance as your starting position. If the present position is the exit, then you have found a path and you are done. If you are not at the exit, then block the present position (i.e., place an obstacle there) so as to prevent the search from returning here. Next see whether there is an adjacent maze position that is not blocked. If so, move to this new adjacent position and attempt to find a path from there to the exit. If unsuccessful, attempt to move to some other unblocked adjacent maze position and try to find a path from there. To facilitate this move, save the current position on a Stack before advancing to a new adjacent position. If all adjacent unblocked positions have been tried and no path is found, there is no path from entrance to exit in the maze.

Remember that from interior (i.e. nonboundary) positions of the maze, four moves are possible: right, down, left, and up. From positions on the boundary of the maze, either two or three moves are possible. To avoid having to handle positions on the boundaries of the maze differently from interior positions, you may find it useful to surround the entire maze with a wall of obstacles. For an m by n maze, this wall will occupy rows 0 and m + 1 and columns 0 and n + 1 of the matrix. See the example below.

Matrix Representation of Sample Maze with wall of 1s around it

```
1 1 1 1 1 1 1 1 1 1 1 1
1 0 1 1 1 1 1 0 0 0 0 1
1 0 0 0 0 0 1 0 1 0 0 1
1 0 0 0 1 0 1 0 0 0 0 1
1 0 1 0 1 0 1 0 1 1 0 1
1 0 1 0 1 0 1 0 1 0 0 1
1 0 1 1 1 0 1 0 1 0 1 1
1 0 1 0 0 0 1 0 1 0 1 1
1 0 1 0 1 1 1 0 1 0 0 1
1 1 0 0 0 0 0 0 1 0 0 1
1 0 0 0 0 1 1 1 1 0 0 1
1 1 1 1 1 1 1 1 1 1 1 1
```

All positions in the maze are now within the boundary of the surrounding wall, so you can move to four possible positions from each position (some of these four positions may have obstacles.) Being able to handle each position in the same way should significantly simplify the search algorithm, although this simplification is achieved at the cost of a slightly increased space requirement for the matrix.

4. **Output Layout:** Output must match the sample output. In order for this to be easier for you, use all the functions contained in outputhelp.h to do your output.

   Sample Output:

```
./mazeproject ../src/maze1.txt ../src/maze2.txt ../src/maze3.txt ../src/maze4.txt  ../src/maze5.txt
../src/maze1.txt: Processing Maze
Steps:12
Path:(7,7)<-(7,6)<-(7,5)<-(7,4)<-(7,3)<-(7,2)<-(7,1)<-(6,1)<-(5,1)<-(4,1)<-(3,1)<-(2,1)<-(1,1)

../src/maze2.txt: Processing Maze
Steps:2
Path:(2,2)<-(1,2)<-(1,1)

../src/maze3.txt: Processing Maze
Steps:36
Path:(10,10)<-(9,10)<-(8,10)<-(8,9)<-(7,9)<-(6,9)<-(5,9)<-(5,10)<-(4,10)<-(3,10)<-(3,9)<-
(3,8)<-(3,7)<-(4,7)<-(5,7)<-(6,7)<-(7,7)<-(8,7)<-(9,7)<-(9,6)<-(9,5)<-(9,4)<-(9,3)<-(8,3)<-
(7,3)<-(7,4)<-(7,5)<-(6,5)<-(5,5)<-(4,5)<-(3,5)<-(2,5)<-(2,4)<-(2,3)<-(2,2)<-(2,1)<-(1,1)

../src/maze4.txt: Processing Maze
No path found!

../src/maze5.txt: Processing Maze
Steps:3
Path:(2,3)<-(2,2)<-(1,2)<-(1,1)
```

5. **Other:**

   - You do NOT have to find the SHORTEST PATH, this is outside the scope of this assignment.
   - You must use a stack to find the path
   - For full credit your code should have zero compiler warnings.
   - No modifications to the header files should be done, doing so may render your code unable to be compiled by the grader
   - Your code will be checked for memory leaks. Memory leaks will be verified via a tool called “valgrind” as well as visually.
   - Usage of any STL containers outside of the STL's stack will result in no credit
   - Usage of recursion will result in no credit
   - Avoid all STL items except std::copy (algorithm), std::endl, assert (cassert), size_t, stack
   - You should not need to make any modifications to position.cpp, and outputhelp.h. Any changes to these files could result in breaking your submitted code when I attempt to compile and test.
   - Only submit the the files listed in #2.