```cpp
//
// Created by nick on 10/11/2023.
//

#ifndef INC_2_MAZE_H
#define INC_2_MAZE_H
#include "Queue.h"
#include "FileOperator.h"

class Maze {
    //Tiles represent indexes in our maze array
    struct Tile {
        int m_y_position{-1};
        int m_x_position{-1};
        char m_character{'\0'};
        bool m_visited{false};
        Tile *m_previous_tile{nullptr};
    };

private:

    //first tile the maze will start from will always be y=1 x=0
    const int M_STARTING_Y_POSITION = 1;
    const int M_STARTING_X_POSITION = 0;

    //the last empty tile in the maze before the exit
    const int M_MAZE_END_Y_POSITION = 49;
    const int M_MAZE_END_X_POSITION = 50;

    //empty space is for finding valid tiles when we search the maze and path marker is for when we mark the best path
    const char M_EMPTY_SPACE = ' ';
    const char M_PATH_MARKER = '#';

    //if this is still false when the solve method ends, don't output the maze array
    bool m_solution_found{false};

    //compiler needs to know the size of the array at runtime for all possible instances of the maze class,
    //so we make these static to guarantee they will never change
    static const int M_ROWS = 51;
    static const int M_COLUMNS = 52;

    //array that holds all the maze tiles, the starting tile which will always be y=1 x=0 and the tile found before
    //the exit of the maze
    Tile m_maze_array[M_ROWS][M_COLUMNS];
    Tile *m_starting_tile{nullptr};
    Tile *m_end_tile{nullptr};

    //holds input and output file.
    FileOperator *m_maze_file_operator{nullptr};

    //checks if the current tile is inside the maze
    bool is_inside_maze(int x, int y);

    //checks if the current tile is an empty space
    bool is_valid_target(Tile *tile);

    //finds the quickest path through the maze
    void find_best_path();

    //outputs the maze to the console and to the output file
    friend std::ostream &operator<<(std::ostream &output, Maze &maze);

public:

    //Fills maze array using the contents of the input file
    Maze(FileOperator &file_operator);

    bool get_solution_found();

    //Queue of nodes that hold directions to valid tiles
    Queue m_position_nodes{M_STARTING_Y_POSITION, M_STARTING_X_POSITION};

    //traverses through the maze finding every empty space until it reaches the end, then calls find_best_path
    void solve();

};

#endif //INC_2_MAZE_H

```

```cpp
//
// Created by nick on 10/15/2023.
//

#ifndef ASSIGNMENT_2_QUEUE_H
#define ASSIGNMENT_2_QUEUE_H


class Queue {
    //Position nodes represent directions to maze tiles.
    struct PositionNode {
        int m_x_position{-1};
        int m_y_position{-1};
        PositionNode *next_node{nullptr};
    };

private:
    //first and last node in the queue
    PositionNode *first_node{nullptr};
    PositionNode *last_node{nullptr};

    //deletes all the nodes from memory
    void empty_queue();

public:
    //give queue our first direction node
    Queue(int x_position, int y_position);

    //add directions to next tile to queue
    void add_node(int y_position, int x_position);

    //if the queue is empty, we are out of tiles to search
    bool is_empty();

    //delete the queue node after getting directions to next tile.
    void pop();

    //grabs the position node at the beginning of the queue
    PositionNode peek();

    //destructor for our queue
    virtual ~Queue();
};


#endif //ASSIGNMENT_2_QUEUE_H

```

```cpp
#include <iostream>
#include "Maze.h"
using namespace std;

int main(int argc, char *argv[]) {
    //validate argument count
    if (argc != 3) {
        cout << "Invalid argument count." << endl;
        return -1;
    }
    //attempt to create file operator with argv[1] and argv[2] which should hold our desired file names.
    FileOperator maze_file_operator(argv[1], argv[2]);

    //if the in file is null it means one of the two file names were invalid.
    if (!maze_file_operator.get_in_file().is_open()) {
        cout << "Invalid file name." << endl;
        return -1;
    }

    // create maze, and attempt to solve it.
    Maze main_maze(maze_file_operator);

    //if we go through every node before we find the exit. no solution has been found.
    if (!main_maze.get_solution_found()){
       cout << "No solution found or invalid maze file." << endl;
       return -1;
    }

    //if a solution was found, output the solution and time taken.
    cout << main_maze << endl;

    //save the solution to the output file.
    maze_file_operator.get_out_file() << main_maze;
    maze_file_operator.get_out_file().close();

    return 0;
}

```

```cpp
//
// Created by nick on 10/11/2023.
//
#include "Maze.h"

using namespace std;

Maze::Maze(FileOperator &file_operator) {

    //file_operator will hold both the input maze file and the file to output the solution to.
    m_maze_file_operator = &file_operator;

    //if the file opens correctly, fill in maze array with file contents
    if (m_maze_file_operator->get_in_file().is_open() && !m_maze_file_operator->get_in_file().fail()) {

        //character is the current character from the text file
        char character;
        //Tiles represent indexes in the maze array
        Tile *current_tile;

        //scan the text file and add every character into the maze array
        for (int y_position = 0; y_position < M_ROWS; y_position++) {
            for (int x_position = 0; x_position < M_COLUMNS; x_position++) {
                //get character from file.
                m_maze_file_operator->get_in_file().get(character);
                //assign the current tile to wherever we are in the maze array
                current_tile = &m_maze_array[y_position][x_position];
                //assign the properties of the current tile
                current_tile->m_character = character;
                current_tile->m_x_position = x_position;
                current_tile->m_y_position = y_position;
                //clear the character variable
                character = M_EMPTY_SPACE;
            }
        }
        //close the input file.
        m_maze_file_operator->get_in_file().close();
        //point the starting tile to the first empty index of our maze array which should always be y=1 x=0
        m_starting_tile = &m_maze_array[M_STARTING_Y_POSITION][M_STARTING_X_POSITION];
        //attempt to solve maze
        solve();
    }
}


```

```cpp
void Maze::solve() {

    //this method will search for every empty space available in the maze until it either hits the exit or runs out
    //of spaces to check, if it runs out of spaces to check before hitting the exit no solution has been found.
    int current_x, current_y, target_y, target_x;

    //while our queue has directional nodes
    while (!m_position_nodes.is_empty()) {

        //grab the node at the front of the queue then remove it from the queue
        auto current_location = m_position_nodes.peek();
        m_position_nodes.pop();

        //get the current tiles x,y from the position node then grab the tile from the maze
        current_x = current_location.m_x_position;
        current_y = current_location.m_y_position;
        Tile *current_tile = &m_maze_array[current_y][current_x];

        //check if we've reached the exit of the maze
        if (current_y == M_MAZE_END_Y_POSITION && current_x == M_MAZE_END_X_POSITION) {
            m_solution_found = true;
            //set the end tile to be whenever we are right now than find the most efficient path
            m_end_tile = current_tile;
            find_best_path();
            return;
        }

        //search for valid locations around us
        //diagram:    y
        //           [-1]
        //      x[-1][0][1]
        //           [1]
        int x_directions[] = {1, -1, 0, 0};
        int y_directions[] = {0, 0, 1, -1};

        for (int current_direction = 0; current_direction < 4; current_direction++) {
            //grab the first tile in the direction we are searching
            target_x = current_x + x_directions[current_direction];
            target_y = current_y + y_directions[current_direction];
            Tile *target_tile = &m_maze_array[target_y][target_x];

            //if the current tile is a valid space inside the maze and hasn't been visited yet,
            //add its location to the queue.
            if (is_valid_target(target_tile)) {
                //link the tile we started on to the tile we are
moving toward
                target_tile->m_visited = true;
                target_tile->m_previous_tile = current_tile;
                //add the directions to our target tile to our nodes queue.
                m_position_nodes.add_node(target_y, target_x);
            }
        }
    }
}

void Maze::find_best_path() {

    //grab the last tile
    auto current_tile = m_end_tile;
    cout << "\n\nSolution:" << endl;

    //traces the best path from the exit back to the entrance of the maze
    //starting from the end tile, it follows the path marked by previous tiles
    //to the entrance, marking each tile with the path marker character
    while (current_tile != nullptr) {
        current_tile->m_character = M_PATH_MARKER;
        //if we've returned to the beginning of the maze, break out of the loop.
        if (current_tile == m_starting_tile) {
            break;
        }
        current_tile = current_tile->m_previous_tile;
    }
}

bool Maze::is_inside_maze(int y, int x) {
    //if current y or x are outside the bounds of the array, ignore them
    return (y > 0 && y < M_ROWS) && (x >= 0 && x < M_COLUMNS);
}

bool Maze::is_valid_target(Tile *tile) {
    //if the tile is a valid space inside the maze and hasn't been visited yet return true.
    return (tile->m_character == M_EMPTY_SPACE && !tile->m_visited) &&
           (is_inside_maze(tile->m_y_position, tile->m_x_position));
}

ostream &operator<<(ostream &output, Maze &maze) {
    //go through every maze array index and add its character to the output stream.
    for (int y_position = 0; y_position < Maze::M_ROWS; y_position++) {
        for (int x_position = 0; x_position < Maze::M_COLUMNS; x_position++) {
            output << maze.m_maze_array[y_position][x_position].m_character;
        }
    }

    return output;
}


bool Maze::get_solution_found() {
    return m_solution_found;
}

```

```cpp
//
// Created by nick on 10/15/2023.
//

#include "Queue.h"


Queue::Queue(int y_position, int x_position) {
    //add the first node to the queue which will hold directions to the starting tile y=1, x=0
    this->add_node(y_position, x_position);
}


void Queue::add_node(int y_position, int x_position) {
    //create new node and assign its position according to the target y and x
    auto new_node = new PositionNode;
    new_node->m_y_position = y_position;
    new_node->m_x_position = x_position;

    //check if the queue is empty
    if (first_node == nullptr) {
        this->first_node = new_node;
    } else {
        //if not, add to the end of the queue
        if (this->last_node != nullptr) {
            this->last_node->next_node = new_node;
        }
    }

    //mark the newest node as the last node in the queue.
    this->last_node = new_node;
}


void Queue::pop() {

    //check to see if the queue is empty
    if (first_node != nullptr) {
        //disconnect the node from the queue
        auto node_to_delete = first_node;

        //check if this node is the only one in the queue
        if (first_node == last_node) {
            last_node = nullptr;
        }

        //since we're deleting from the front of the queue, move the next node forward
        first_node = node_to_delete->next_node;

        //delete the disconnected node from memory
        delete node_to_delete;
    }
}

Queue::PositionNode Queue::peek() {
    //grab the reference to the first node of the queue
    return *first_node;
}

bool Queue::is_empty() {
    //if the first node is null, return true
    return first_node == nullptr;
}

void Queue::empty_queue() {
    //delete the first node in the queue until the queue is empty.
    while (!this->is_empty()) {
        this->pop();
    }
}

Queue::~Queue() {
    empty_queue();
}

```

```cpp
//
// Created by nick on 10/18/2023.
//
#ifndef ASSIGNMENT_2_FILEOPERATOR_H
#define ASSIGNMENT_2_FILEOPERATOR_H
#include <iostream>
#include <fstream>
#include <regex>


class FileOperator {

private:
    //regex pattern for valid .txt file
    const std::regex M_FILENAME_REGEX = std::regex(R"(^(?:.*[\\/:])?[^\\/:*?"<>|]+\.txt$)");
    //path used for creating our output file, append to the front of the desired output file name.
    const std::string M_OUTPUT_FILE_BASE_PATH = "../solved/";

    //input maze file
    std::ifstream m_in_file;
    //print maze to this file after maze has been solved
    std::ofstream m_out_file;

    //uses the regex to make sure the file name is a valid txt file
    bool has_valid_name(std::string &file_name_input);

public:
    //create file operator object using valid file name
    FileOperator(std::string in_file_name, std::string out_file_name);

    //getters for file objects
    std::ifstream &get_in_file();
    std::ofstream &get_out_file();

};

#endif //ASSIGNMENT_2_FILEOPERATOR_H

```

```cpp
//
// Created by nick on 10/18/2023.
//
#include "FileOperator.h"
using namespace std;

FileOperator::FileOperator(string in_file_name, string out_file_name) {
    //if both file names are valid, create the file objects.
    if (has_valid_name(in_file_name) && has_valid_name(out_file_name)) {
        m_in_file = ifstream(in_file_name);
        m_out_file = ofstream(M_OUTPUT_FILE_BASE_PATH + out_file_name);
    }
}

ifstream &FileOperator::get_in_file() {
    return m_in_file;
}


ofstream &FileOperator::get_out_file() {
    return m_out_file;
}


bool FileOperator::has_valid_name(std::string &file_name_input) {
    //any file name / path that ends in .txt returns true.
    return regex_match(file_name_input, M_FILENAME_REGEX);
}

```