File - C:\Users\jamzc\OneDrive\Desktop\college garbage\PROG2400\Assignments\assignment-2-jeverill0462954\src\maze.h

```h
#ifndef ASSIGNMENT_2_MAZE_H
#define ASSIGNMENT_2_MAZE_H

#include <vector>
#include <string>

// A cell in the maze.
struct Coordinates {
    int x_coord{};
    int y_coord{};
    Coordinates();
    Coordinates(int x_input, int y_input);
};

// The Maze class is used for storing the maze itself inside a two-dimensional vector. The constructor
// creates the vector from a text file.
class Maze {
    private:
        std::vector<std::vector<char>> _maze_vector;
    public:
        // Populate _maze_vector by parsing a text file.
        explicit Maze(const std::string& _input_filepath);
        // Outputs a string that displays what the maze looks like.
        std::string output_maze();
        // Find the entrance of a maze.
        Coordinates find_entrance();
        // Test whether a coordinate is empty
        bool is_empty(Coordinates coord);
        // Test whether a coordinate is the middle of an intersection.
        bool is_intersection(Coordinates coord);
        // Test whether a coordinate is the exit
        bool is_exit(Coordinates coord);
        // Draw a part of the solution line.
        void draw_solution_portion(Coordinates coord);
        // Delete a part of the solution line.
        void delete_solution_portion(Coordinates coord);
        // Block a coordinate with an 'X'
        void block_coordinate(Coordinates coord);
};

#endif //ASSIGNMENT_2_MAZE_H

```

File - C:\Users\jamzc\OneDrive\Desktop\college garbage\PROG2400\Assignments\assignment-2-jeverill0462954\src\stack.h

```
#ifndef ASSIGNMENT_2_STACK_H
#define ASSIGNMENT_2_STACK_H

#include <memory>
#include <optional>
#include "maze.h"

// A piece of the stack. Constructor for automatically inputting the coordinates is provided.
struct Node {
    std::unique_ptr<Node> _next{nullptr};
    Coordinates _coords{};
    explicit Node(Coordinates _input_coords);
};

class Stack {
    // The last node in the stack.
    std::unique_ptr<Node> _top{nullptr};
public:
    // Add something on top of the stack.
    void push(Coordinates coords);
    // Return the top of the stack's contents.
    std::optional<Coordinates> peek();
    // Remove the top node of the stack.
    void pop();
    // Get the data from the head
    Coordinates top_coordinates();
};

#endif //ASSIGNMENT_2_STACK_H
```

File - C:\Users\jamzc\OneDrive\Desktop\college garbage\PROG2400\Assignments\assignment-2-jeverill0462954\src\main.cpp

```cpp
#include <fstream>
#include <iostream>
#include "maze_solver.h"

using namespace std;

int main(int argc, char* argv[]) {
    string output_filepath;
    if (argc < 3) {
        cout << "error: program requires input and output filepath";
        return 1;
    }
    else {
        cout << "Running Maze Solver..." << endl;
        MazeSolver solver(argv[1]);
        cout << "Solving..." << endl;
        string solution = solver.solve_maze();
        cout << "Done!" << endl;
        ofstream output;
        output.open(argv[2]);
        output << solution;
        output.close();
        cout << "Saved to " << argv[2] << "." << endl;
        return 0;
    }
}
```

File - C:\Users\jamzc\OneDrive\Desktop\college garbage\PROG2400\Assignments\assignment-2-jeverill0462954\src\maze.cpp

```cpp
#include "maze.h"
#include <fstream>

Coordinates::Coordinates(int x_input, int y_input) {
    x_coord = x_input;
    y_coord = y_input;
}

Coordinates::Coordinates() = default;

Maze::Maze(const std::string& _input_filepath) {
    std::ifstream _file(_input_filepath);
    int _row_number = 0;
    // ideally we'd like to populate with booleans, but since we're printing this it doesn't make
    // sense to waste memory on a completely new vector that works *slightly* better for parsing
    for (std::string _row_content; std::getline(_file, _row_content); _row_number++) {
        _maze_vector.emplace_back();
        for (int i = 0; i < _row_content.length(); i++) {
            _maze_vector[_row_number].push_back(_row_content[i]);
        }
    }
}

std::string Maze::output_maze() {
    std::string _output;
    for (auto & i : _maze_vector) {
        for (char j : i) {
            // clean output of Xs
            if (j == 'X') {
                _output.append(1,' ');
            }
            else {
                _output.append(1,j);
            }
        }
        _output.append(1,'\n');
    }
    return _output;
}

Coordinates Maze::find_entrance() {
    // declaring this array as static prevents undefined behavior from returning an array
    Coordinates entrance;
    for (int i = 1; i < _maze_vector.size()-1; i++) {
        if (_maze_vector[i][0] == ' ') {
            entrance = Coordinates(0,i);
            break;
        }
        else if (_maze_vector[i][_maze_vector.size()-1] == ' ') {
            entrance = Coordinates((int)_maze_vector.size()-1,i);
            break;
        }
    }
    return entrance;
}

bool Maze::is_empty(Coordinates coord) {
    return (_maze_vector[coord.y_coord][coord.x_coord] == ' ');
}

bool Maze::is_intersection(Coordinates coord) {
    int _path_counter = 0;
    if (_maze_vector[coord.y_coord + 1][coord.x_coord] == ' ') {
        _path_counter++;
    }
    if (_maze_vector[coord.y_coord - 1][coord.x_coord] == ' ') {
        _path_counter++;
    }
    if (_maze_vector[coord.y_coord][coord.x_coord + 1] == ' ') {
        _path_counter++;
    }
    if (_maze_vector[coord.y_coord][coord.x_coord - 2] == ' ') {
        _path_counter++;
    }
    return (_path_counter >= 2);
}

bool Maze::is_exit(Coordinates coord) {
    return (coord.x_coord == 0 ||
            coord.x_coord == _maze_vector[1].size()-1 ||
            coord.y_coord == 0 ||
            coord.y_coord == _maze_vector.size()-1
            );
}

void Maze::draw_solution_portion(Coordinates coord) {
    _maze_vector[coord.y_coord][coord.x_coord] = '#';
}

void Maze::delete_solution_portion(Coordinates coord) {
    _maze_vector[coord.y_coord][coord.x_coord] = ' ';
}

void Maze::block_coordinate(Coordinates coord) {
    _maze_vector[coord.y_coord][coord.x_coord] = 'X';
}

```

File - C:\Users\jamzc\OneDrive\Desktop\college garbage\PROG2400\Assignments\assignment-2-jeverill0462954\src\stack.cpp

```cpp
#include "stack.h"

Node::Node(Coordinates _input_coords) {
    _coords = _input_coords;
}

void Stack::push(Coordinates coords) {
    auto node = std::make_unique<Node>(coords);
    node->_next = std::move(_top);
    _top = std::move(node);
}

std::optional<Coordinates> Stack::peek() {
    if(_top == nullptr) return std::nullopt;
    return std::make_optional<Coordinates>(_top->_coords);
}

void Stack::pop() {
    if (_top != nullptr) {
        _top = std::move(_top->_next);
    }
}

Coordinates Stack::top_coordinates() {
    return _top->_coords;
}

```

File - C:\Users\jamzc\OneDrive\Desktop\college garbage\PROG2400\Assignments\assignment-2-jeverill0462954\src\maze_solver.h

```h
#ifndef ASSIGNMENT_2_MAZE_SOLVER_H
#define ASSIGNMENT_2_MAZE_SOLVER_H

#include "stack.h"
#include "maze.h"

// A stack that moves through a maze until it finds the exit.
class MazeSolver {
    Maze _maze;
    Stack _path;
    bool _dead_end_ahead = false;
    bool _is_solved = false;
public:
    // Finds the entrance of the maze and starts the stack.
    explicit MazeSolver(const std::string& _input_filepath);
    // While loop that executes this classes functions until the maze is solved
    std::string solve_maze();
    // Make a move forward. Marks _dead_end_ahead as true if it cannot move.
    void move_forward();
    // Move backwards. Blocks the path and marks _dead_end_ahead as false if it finds another valid path.
    void move_backward();
    // Push the top of the stack and draw on the maze vector simultaneously.
    void push_and_draw(Coordinates _coordinates);
    // Pops the top of the stack and removes its segment on the maze vector simultaneously.
    void pop_and_erase();
};


#endif //ASSIGNMENT_2_MAZE_SOLVER_H

```

File - C:\Users\jamzc\OneDrive\Desktop\college garbage\PROG2400\Assignments\assignment-2-jeverill0462954\src\maze_solver.c

```cpp
#include "maze_solver.h"

MazeSolver::MazeSolver(const std::string &_input_filepath) : _maze(_input_filepath) {
    // create the stack/snake at the starting point of the maze, get the foot out the door
    _path.push(_maze.find_entrance());
    _maze.draw_solution_portion(_path.top_coordinates());
    move_forward();
}

std::string MazeSolver::solve_maze() {
    while (!_is_solved) {
        if (_maze.is_exit(_path.top_coordinates())) {
            _is_solved = true;
        }
        else {
            move_forward();
            while (_dead_end_ahead) {
                move_backward();
            }
        }
    }
    return _maze.output_maze();
}

void MazeSolver::move_forward() {
    Coordinates _current_pos = _path.top_coordinates();
    Coordinates _new_position;
    // right
    if (_maze.is_empty(_new_position = Coordinates(_current_pos.x_coord+1,_current_pos.y_coord))) {
        push_and_draw(_new_position);
    }
        // down
    else if (_maze.is_empty(_new_position = Coordinates(_current_pos.x_coord,_current_pos.y_coord+1))) {
        push_and_draw(_new_position);
    }
        // left
    else if (_maze.is_empty(_new_position = Coordinates(_current_pos.x_coord-1,_current_pos.y_coord))) {
        push_and_draw(_new_position);
    }
        // up
    else if (_maze.is_empty(_new_position = Coordinates(_current_pos.x_coord,_current_pos.y_coord-1))) {
        push_and_draw(_new_position);
    }
    else {
        _dead_end_ahead = true;
    }
}

void MazeSolver::move_backward() {
    Coordinates _deleted_coords = _path.top_coordinates();
    pop_and_erase();
    if (_dead_end_ahead && _maze.is_intersection(_path.top_coordinates())) {
        _dead_end_ahead = false;
        _maze.block_coordinate(_deleted_coords);
    }
}

void MazeSolver::push_and_draw(Coordinates _coordinates) {
    _path.push(_coordinates);
    _maze.draw_solution_portion(_coordinates);
}

void MazeSolver::pop_and_erase() {
    _maze.delete_solution_portion(_path.top_coordinates());
    _path.pop();
}

```