```cpp
#ifndef ASSIGNMENT_2_GRIDPOS_H
#define ASSIGNMENT_2_GRIDPOS_H

struct GridPos {
    int m_row{-1};
    int m_col{-1};
};

#endif //ASSIGNMENT_2_GRIDPOS_H

```

```cpp
#include "MazeSolver.h"
#include <iostream>

using std::string, std::cout, std::cin;

int main(int argc, char **argv) {

    MazeSolver maze{};
    string file_name;

    // Prompt user for filename if one was not given, then open/read file
    if (argc < 2) {
        cout << "No file path was given, please specify: ";
        getline(cin, file_name);
    } else {
        file_name = argv[1];
    }

    // Solve maze and display output to console
    maze.solve(file_name);
    maze.display_maze();

    return 0;
}
```

```cpp
#include "MazeSolver.h"
#include "Stack.h"
#include <iostream>
#include <fstream>
#include <filesystem>

using namespace std;

MazeSolver::~MazeSolver() = default;

void MazeSolver::read_file(string &path) {
    fstream target_file(path);
    string line;
    auto count{0};

    // Open file and read in maze
    try {
        if (target_file.is_open()) {
            // Populate Maze 2D array with file data
            while (getline(target_file, line)) {
                for (auto i = 0; i < line.length(); i++) {
                    m_maze[count][i] = line[i];
                }
                count++;
            }
        }
    } catch (exception &e) {
        cout << e.what() << endl;
    }
    target_file.close();
}

void MazeSolver::save_solution(string &path) {
    // Save solution to file
    fstream target_file;
    filesystem::path p(path);
    string file_name = "..\\solved\\" + p.stem().string() + "_solution.txt";

    try {
        target_file.open(file_name, ios::out | ios::trunc);

        if (target_file.is_open()) {
            // Output maze to file
            for (auto i = 0; i < MAZE_WIDTH; i++) {
                for (auto j = 0; j < MAZE_LENGTH; j++) {
                    target_file << m_maze[i][j];
                }
                if (i != MAZE_LENGTH - 1) {
                    target_file << '\n';
                }
            }
        } else {
            cout << "Failed to save solution." << endl;
        }
    } catch (exception &e) {
        cout << e.what();
    }
    target_file.close();
    cout << "Solution saved to: " << file_name << endl;
}

void MazeSolver::solve(string &path) {
    bool solved = false;
    Stack pos;
    GridPos curr_loc{1, 0};

    // Read file data into maze
    read_file(path);

    // Loop to find correct path through maze
    while (!solved) {

        // Check if maze was solved
        if (curr_loc.m_row == MAZE_LENGTH - 2 &&
            curr_loc.m_col == MAZE_WIDTH - 1) {
            pos.push(curr_loc);
            solved = true;
        }

        // Check to the right of current position (East)
        if (m_maze[curr_loc.m_row][curr_loc.m_col + 1] == ' ') {
            pos.push(curr_loc);
            m_maze[curr_loc.m_row][curr_loc.m_col] = '#';
            curr_loc.m_col++;
            continue;
        }

        // Check below the current position (South)
        if (m_maze[curr_loc.m_row + 1][curr_loc.m_col] == ' ') {
            pos.push(curr_loc);
            m_maze[curr_loc.m_row][curr_loc.m_col] = '#';
            curr_loc.m_row++;
            continue;
        }

        // Check to the left of the current position (West)
        if (m_maze[curr_loc.m_row][curr_loc.m_col - 1] == ' ') {
            pos.push(curr_loc);
            m_maze[curr_loc.m_row][curr_loc.m_col] = '#';
            curr_loc.m_col--;
            continue;
        }

        // Check above the current position (North)
        if (m_maze[curr_loc.m_row - 1][curr_loc.m_col] == ' ') {
            pos.push(curr_loc);
            m_maze[curr_loc.m_row][curr_loc.m_col] = '#';
            curr_loc.m_row--;
            continue;
        }

        // Check if path was dead end
        GridPos top = pos.peek();

        if (top.m_row != -1 && top.m_col != -1) {
            // Check if dead end is actually end of the maze
            if (top.m_row == MAZE_LENGTH - 2 &&
                top.m_col == MAZE_LENGTH - 1) {
                m_maze[curr_loc.m_row][curr_loc.m_col] = '#';
                pos.push(curr_loc);
            } else {
                // Mark dead end to prevent returning
                m_maze[curr_loc.m_row][curr_loc.m_col] = 'X';
            }
            // Back track and try another route
            curr_loc = pos.peek();
            pos.pop();
            continue;
        }
    }

    // Erase dead end marks from maze
    for (auto &i: m_maze) {
        for (auto &j: i) {
            if (j == 'X') {
                j = ' ';
            }
        }
    }

    // Save solution to file
    save_solution(path);
}

void MazeSolver::display_maze() {
    // Display maze to console
    for (auto &i: m_maze) {
        for (auto j: i) {
            cout << j;
        }
        cout << endl;
    }
}
```

```cpp
#ifndef ASSIGNMENT_2_MAZESOLVER_H
#define ASSIGNMENT_2_MAZESOLVER_H

#include <string>

class MazeSolver {
private:
    auto static const MAZE_LENGTH{51};
    auto static const MAZE_WIDTH{51};
    char m_maze[MAZE_LENGTH][MAZE_WIDTH];
public:
    ~MazeSolver();

    void read_file(std::string &);

    void save_solution(std::string &);

    void solve(std::string &);

    void display_maze();
};

#endif //ASSIGNMENT_2_MAZESOLVER_H

```

```
#ifndef ASSIGNMENT_2_NODE_H
#define ASSIGNMENT_2_NODE_H

#include "GridPos.h"

struct Node {
    GridPos m_location;
    Node *m_next{nullptr};
};

#endif //ASSIGNMENT_2_NODE_H

```

```cpp
#include <iostream>
#include "Stack.h"

using namespace std;

Stack::~Stack() {
    // Remove nodes using stack.pop() function
    while (m_first != nullptr) {
        pop();
    }
}

void Stack::push(GridPos location) {
    // Add node to stack
    auto node = new Node();
    node->m_location = location;
    node->m_next = m_first;
    m_first = node;
}

void Stack::pop() {
    // Pop top node off stack
    if (m_first == nullptr) {
        return;
    }
    auto node = m_first;
    m_first = m_first->m_next;
    delete node;
}

GridPos Stack::peek() {
    // Peek at top node on stack
    if (m_first == nullptr) return {-1, -1};
    return m_first->m_location;
}

std::ostream &operator<<(std::ostream &output, Stack &stack) {
    // Output stack coordinates in X , Y formatted table
    auto node = stack.m_first;
    output << " X , Y" << endl << "+-----+" << endl;
    while (node != nullptr) {
        output << " " << node->m_location.m_col << " , ";
        output << node->m_location.m_row << endl;
        node = node->m_next;
    }
    output << "+-----+";
    return output;
}

```

```
#ifndef ASSIGNMENT_2_STACK_H
#define ASSIGNMENT_2_STACK_H

#include "Node.h"

class Stack {
private:
    Node *m_first{nullptr};
public:
    ~Stack();

    void push(GridPos);

    void pop();

    GridPos peek();

    friend std::ostream &operator<<(std::ostream &, Stack &);
};

#endif //ASSIGNMENT_2_STACK_H

```