```
1  #ifndef ASSIGNMENT_2_STACK_H
2  #define ASSIGNMENT_2_STACK_H
3  #include <iostream>
4  #include <memory>
5  #include "position.h"
6
7  class Stack {
8      struct Node {
9          Position _position;
10         std::unique_ptr<Node> _next;
11
12         Node(Position position) : _position(position), _next(nullptr) {}
13     };
14
15     std::unique_ptr<Node> _top;
16
17 public:
18     Stack();
19     void push(Position position);
20     void pop();
21     [[nodiscard]] bool empty() const;
22     [[nodiscard]] Position top() const;
23 };
24
25 #endif
26
```

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

using namespace std;

int main(int argc, char** argv) {
    //expecting 3 arguments, the program name and the input file and the output file
    if (argc != 3) {
        std::cout << "Error, incorrect number of arguments." << std::endl;
        return 1;
    }

    MazeSolver maze_solver;
    maze_solver.run(argv[1], argv[2]);

    return 0;
}
```

```cpp
#include <iostream>
#include <filesystem>
#include "stack.h"
#include "position.h"

Stack::Stack() : _top(nullptr) {}

void Stack::push(Position position) {
    /// push the position to the stack

    auto node = std::make_unique<Node>(position);
    node->_next = std::move(_top);
    _top = std::move(node);
}

void Stack::pop() {
    /// pop the position from the stack

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

bool Stack::empty() const {
    /// check if the stack is empty, if it is empty set the top to nullptr

    return _top == nullptr;
}

Position Stack::top() const {
    /// get the top position of the stack

    if (!empty()) {
        return _top->_position;
    }
    // if the stack is empty, return an invalid position
    return Position{-1, -1};
}

```

```
1  #ifndef ASSIGNMENT_2_POSITION_H
2  #define ASSIGNMENT_2_POSITION_H
3
4  class Position {
5  public:
6      int _row, _col;
7
8      Position(int x, int y);
9      bool operator==(const Position &other) const;
10 };
11
12 #endif
13
```

```cpp
#include "position.h"

Position::Position(int r, int c) {
    _row = r;
    _col = c;
}

bool Position::operator==(const Position &other) const {
    return _row == other._row && _col == other._col;
}


```

```
1  #ifndef ASSIGNMENT_2_MAZE_SOLVER_H
2  #define ASSIGNMENT_2_MAZE_SOLVER_H
3  #include <vector>
4  #include <fstream>
5  #include <filesystem>
6  #include "stack.h"
7  #include "position.h"
8
9  class MazeSolver {
10     static const int MAX_ROWS = 51;
11     static const int MAX_COLS = 52;
12     std::fstream _unsolved_file;
13     std::fstream _solved_file;
14     char _maze[MAX_ROWS][MAX_COLS];
15
16 public:
17     bool file_io(const std::filesystem::path& file_path_unsolved,
18                  const std::filesystem::path& file_path_solved);
19     void solve_maze();
20     void run(const char* unsolved_filename, const char* solved_filename);
21     void find_path(Stack& path_stack, Position& current, bool& moved);
22     void mark_path(Stack& path_stack);
23     void save_maze();
24     void print_maze();
25 };
26
27 #endif
28
```

```cpp
#include <iostream>
#include <filesystem>
#include "maze_solver.h"
#include "stack.h"
#include "position.h"

void MazeSolver::run(const char *unsolved_filename, const char *solved_filename) {
    /// run the program
    /// \param unsolved_filename the file name of the unsolved maze
    /// \param solved_filename the file name of the solved maze

    std::filesystem::path unsolved_path = unsolved_filename;
    std::filesystem::path solved_path = solved_filename;

    bool file_opened = file_io(unsolved_path, solved_path);
    if (file_opened) {
        std::cout << "File Opened Successfully" << std::endl;
    } else {
        return;
    }
    solve_maze();
    print_maze();
    save_maze();
}

bool MazeSolver::file_io(const std::filesystem::path& file_path_unsolved,
                         const std::filesystem::path& file_path_solved) {
    ///opens the file and puts the contents into an array
    /// \param unsolved_filename the file name of the unsolved maze
    /// \param solved_filename the file name of the solved maze

    if (!std::filesystem::exists(file_path_unsolved)) {
        std::cout << "Error: File '" << file_path_unsolved
        << "' does not exist." << std::endl;
        return false;
    } else {
        _unsolved_file.open(file_path_unsolved);
    }

    if (!_unsolved_file.is_open()) {
        std::cerr << "Error: Unable to open file '"
        << file_path_unsolved << "'" << std::endl;
        return false;
    }

    for (int i = 0; i < MAX_ROWS; i++) {
        for (int j = 0; j < MAX_COLS; j++) {
            _unsolved_file.get(_maze[i][j]);
```

```
49          }
50      }
51
52      _solved_file.open(file_path_solved, std::ios::out);
53
54      return true;
55  }
56
57  void MazeSolver::solve_maze() {
58      ///solve the maze
59
60      Stack path_stack;
61      Position start_position = {1, 0};
62      Position end_position = {49, 50};
63      path_stack.push(start_position);
64
65      while (!path_stack.empty()) {
66          Position current = path_stack.top();
67          bool moved = false;
68
69          if (current == end_position) {
70              mark_path(path_stack);
71              break;
72          }
73          find_path(path_stack, current, moved);
74      }
75  }
76
77  void MazeSolver::mark_path(Stack &path_stack) {
78      /// mark the correct path of the maze and unmark the wrong path
79
80      while (!path_stack.empty()) {
81          Position pos = path_stack.top();
82          // Mark as part of the solution path and pop the position from the stack to
83          // move to the next position
84          _maze[pos._row][pos._col] = '#';
85          path_stack.pop();
86
87          for (int i = 0; i < MAX_ROWS; i++) {
88              for (int j = 0; j < MAX_COLS; j++) {
89                  if (_maze[i][j] == 'X') {
90                      _maze[i][j] = ' ';
91                  }
92              }
93          }
94      }
95  }
96
```

```
 97 void MazeSolver::save_maze() {
 98     /// save the solved maze to the solution file
 99
100     for (int i = 0; i < MAX_ROWS; i++) {
101         for (int j = 0; j < MAX_COLS; j++) {
102             _solved_file << _maze[i][j];
103         }
104     }
105 }
106
107 void MazeSolver::print_maze() {
108     /// print out the maze
109
110     for (int i = 0; i < MAX_ROWS; i++) {
111         for (int j = 0; j < MAX_COLS; j++) {
112             std::cout << _maze[i][j];
113         }
114     }
115 }
116
117 void MazeSolver::find_path(Stack &path_stack, Position &current, bool &moved) {
118     /// find the path of the maze
119     /// \param path_stack the stack with the path
120     /// \param current the current position
121     /// \param moved if the position has been moved
122
123     // array of directions to check surrounding positions
124     // (up, down, left, right)
125     const Position directions[] = {{-1, 0},
126                                    {1,  0},
127                                    {0,  -1},
128                                    {0,  1}};
129
130
131     for (const auto &dir: directions) {
132         int next_row = current._row + dir._row;
133         int next_col = current._col + dir._col;
134
135         // Check if the next move is valid (an empty space to move into)
136         if (next_row >= 0 && next_row < MAX_ROWS && next_col >= 0 &&
137             next_col < MAX_COLS && _maze[next_row][next_col] == ' ') {
138             path_stack.push({next_row, next_col});
139             // Mark the move as 'X' to indicate it has been visited
140             _maze[next_row][next_col] = 'X';
141             moved = true;
142             break;
143         }
144     }
```

```
145
146     // If no valid moves were found (dead end), pop the current position from the stack
147     if (!moved) {
148         path_stack.pop();
149     }
150 }
```