```
 1 cmake_minimum_required(VERSION 3.17)
 2 project(Assignment_2)
 3
 4 set(CMAKE_CXX_STANDARD 20)
 5
 6 add_executable(Assignment_2
 7         src/main.cpp
 8         src/maze_solver.cpp
 9         inc/maze_solver.h
10         src/maze.cpp
11         inc/maze.h
12         inc/stack.h
13         inc/coordinate.h
14         inc/colours.h)
15
16 add_executable(solve
17         src/main.cpp
18         src/maze_solver.cpp
19         inc/maze_solver.h
20         src/maze.cpp
21         inc/maze.h
22         inc/stack.h
23         inc/coordinate.h
24         inc/colours.h)
25
```

```
 1 //
 2 // maze.h
 3 // Created by Kaitlyn Archambault on 2023-10-03.
 4 //
 5
 6 #ifndef ASSIGNMENT_2_MAZE_H
 7 #define ASSIGNMENT_2_MAZE_H
 8
 9 #include <string>
10 #include "coordinate.h"
11
12 const int MAZE_SIZE = 51;
13
14 /**
15  * @brief A class representing a 2D maze from a text file in coordinates.
16  *
17  */
18 class Maze {
19 private:
20     char m_maze[MAZE_SIZE][MAZE_SIZE]{}; // The maze grid
21
22     Coordinate<int> m_maze_start; // coordinate representing start of the maze
23     Coordinate<int> m_maze_end;  // coordinate representing end of the maze
24
25 public:
26     /**
27      * @brief Constructor to create a Maze object from a maze file.
28      *
29      * Initializes the maze grid and the maze start/end coordinates by reading a file.
30      *
31      * @param maze_file The file containing the maze data.
```

```
32      */
33     explicit Maze(const std::string &maze_file);
34
35     /**
36     * @brief Destructor.
37     */
38     ~Maze() = default;
39
40     /**
41      * @brief Get the start coordinate of the maze.
42      *
43      * @return The start coordinate.
44      */
45     Coordinate<int> get_maze_start() const;
46
47     /**
48      * @brief Get the end coordinate of the maze.
49      *
50      * @return The end coordinate.
51      */
52     Coordinate<int> get_maze_end() const;
53
54     /**
55      * @brief Indicate the solution with a special
   character at a specific coordinate.
56      *
57      * @param p The coordinate at which to set the
   character.
58      * @param c The character to set at the coordinate
   .
59      *
60      * @note permanently alters the maze grid, should
   only be used when the full solution has been found
61      */
62     void set_solution_at(Coordinate<int> p, char c);
63
64     /**
65      * @brief Get the character at a specific
```

```
65 coordinate in the maze.
66      *
67      * @param p The coordinate for which to get the
   character.
68      * @return The character at the specified
   coordinate.
69      */
70     char at(Coordinate<int> p);
71
72     /**
73      * @brief Check if a coordinate is within the
   bounds of the maze.
74      *
75      * @param p The coordinate to check.
76      * @return `true` if the coordinate is within the
    maze bounds, `false` otherwise.
77      */
78     bool contains(Coordinate<int> p);
79
80     /**
81      * @brief Save the maze solution to a file.
82      *
83      * @param filename The name of the file to save
   the solution to.
84      */
85     void save_to_file(std::string filename);
86
87
88     /**
89      * @brief  operator to print the maze.
90      *
91      * @param output The output stream.
92      * @param maze The Maze object to print.
93      * @return The output stream with the printed
   maze.
94      */
95     friend std::ostream &operator<<(std::ostream &
   output, Maze &maze);
```

```
 96 };
 97
 98
 99 #endif //ASSIGNMENT_2_MAZE_H
100
```

```
 1 //
 2 // stack.h
 3 // Created by Kaitlyn Archambault on 2023-10-03.
 4 //
 5
 6 #ifndef ASSIGNMENT_2_STACK_H
 7 #define ASSIGNMENT_2_STACK_H
 8
 9 /**
10  * @brief A class to represent Stack data structure.
11  *
12  * Includes push, pop, peek, and size operations, and
   implements smart pointers.
13  *
14  * @tparam T The type of data stored in the stack.
15  */
16 template<class T>
17 class Stack {
18 private:
19     //stack nodes contain data and a smart pointer to
   the next node
20     struct Node {
21         T m_data;
22         std::unique_ptr<Node> m_next;
23     };
24
25     std::unique_ptr<Node> m_top; // a smart pointer to
    the top of the stack
26     int size; // the number of nodes in the stack
27
28 public:
29     /**
30      * @brief Default constructor to initialize an
   empty stack.
31      */
32     Stack() { size = 0; }
33
34     /**
```

```
35      * @brief Default destructor.
36      */
37     ~Stack() = default;
38 
39     /**
40      * @brief check if the stack contains any data
41      *
42      * @return `true` if the stack is empty, `false` otherwise.
43      */
44     bool empty() {
45         return m_top == nullptr;
46     }
47 
48     /**
49      * @brief Add new data to the top of the stack
50      *
51      * @param data The data to be added.
52      */
53     void push(T data) {
54         std::unique_ptr<Node> node = std::make_unique<Node>();
55         node->m_data = data;
56 
57         if (!empty()) {
58             // new node is linked to the current top element
59             node->m_next = std::move(m_top);
60         }
61 
62         // new node goes to the top of the stack
63         m_top = std::move(node);
64         size++;
65     }
66 
67     /**
68      * @brief Pop and return the data of the top node on the stack.
```

```
 69      *
 70      * @return The data of the top node on the stack.
 71      */
 72     T pop() {
 73         Node *temp = m_top.get();
 74         T temp_data = temp->m_data;
 75
 76         // pop the current top node off the stack
 77         m_top = std::move(m_top->m_next);
 78         size--;
 79
 80         return temp_data;
 81     }
 82
 83     /**
 84      * @brief Get the data at the top of the stack
    without removing it.
 85      *
 86      * @return The data of the top node on the stack.
 87      */
 88     T peek() {
 89         return m_top->m_data;
 90     }
 91
 92     /**
 93      * @brief Get the number of nodes in the stack.
 94      *
 95      * @return The size of the stack.
 96      */
 97     int get_size() {
 98         return size;
 99     }
100 };
101
102
103 #endif //ASSIGNMENT_2_STACK_H
104
```

```
 1 //
 2 // colours.h
 3 // Created by Kaitlyn Archambault on 2023-10-22.
 4 //
 5
 6 #ifndef ASSIGNMENT_2_COLOURS_H
 7 #define ASSIGNMENT_2_COLOURS_H
 8
 9
10 #define RESET   "\033[0m"
11 #define BLACK   "\033[30m"
12 #define RED     "\033[31m"
13 #define GREEN   "\033[32m"
14 #define YELLOW  "\033[33m"
15 #define BLUE    "\033[34m"
16 #define MAGENTA "\033[35m"
17 #define CYAN    "\033[36m"
18 #define WHITE   "\033[37m"
19 #define BOLD_BLACK   "\033[1m\033[30m"
20 #define BOLD_RED     "\033[1m\033[31m"
21 #define BOLD_GREEN   "\033[1m\033[32m"
22 #define BOLD_YELLOW  "\033[1m\033[33m"
23 #define BOLD_BLUE    "\033[1m\033[34m"
24 #define BOLD_MAGENTA "\033[1m\033[35m"
25 #define BOLD_CYAN    "\033[1m\033[36m"
26 #define BOLD_WHITE   "\033[1m\033[37m"
27
28
29 #endif //ASSIGNMENT_2_COLOURS_H
30
```

```
 1 //
 2 // coordinate.h
 3 // Created by Kaitlyn Archambault on 2023-10-04.
 4 //
 5
 6 #ifndef ASSIGNMENT_2_COORDINATE_H
 7 #define ASSIGNMENT_2_COORDINATE_H
 8
 9 #include <array>
10 #include <concepts>
11
12 /**
13  * @brief A concept for arithmetic types (integral or
   floating-point).
14  *
15  * https://stackoverflow.com/questions/14294267/class-
   template-for-numeric-types
16  */
17 template<typename T>
18 concept numeric = std::integral<T> or std::
   floating_point<T>;
19
20 /**
21  * @brief A class representing 2D coordinates with x
   and y values of a numeric type.
22
23  *
24  * @tparam T The type for x and y values (must satisfy
    numeric concept).
25  */
26 template<class T> requires numeric<T>
27 class Coordinate {
28 private:
29     T x; // x-coordinate
30     T y; // y-coordinate
31
32 public:
33     /**
```

```
34      * @brief Constructor, initializes x and y to 0.
35      */
36     Coordinate() : x(0), y(0) {};
37 
38     /**
39      * @brief Constructor to initialize coordinates
   with specified values.
40      *
41      * @param x coordinate.
42      * @param y coordinate.
43      */
44     Coordinate(T x, T y) : x(x), y(y) {};
45 
46     /**
47     * @brief Copy constructor.
48     *
49     * @param p The Coordinate to copy.
50     */
51     Coordinate(const Coordinate &p) = default;
52 
53     /**
54      * @brief Default destructor.
55      */
56     ~Coordinate() = default;
57 
58     T get_x() const {
59         return this->x;
60     }
61 
62     T get_y() const {
63         return this->y;
64     }
65 
66     /**
67      * @brief Get an array of coordinates north, east
   , south, west of the current coordinate
68      *
69      * @param step_size distance between coordinates
```

```
70      *
71      * @returns An array of 4 Coordinate objects
72      * @note in the future, step_size will ideally be
    determined within this class
73      */
74     std::array<Coordinate, 4>
   get_adjacent_coordinates(T step_size) const {
75
76         std::array<Coordinate, 4> options;
77
78         options[0] = Coordinate(this->x + step_size,
   this->y); // east
79         options[1] = Coordinate(this->x, this->y +
   step_size); // south
80         options[2] = Coordinate(this->x, this->y -
   step_size); // north
81         options[3] = Coordinate(this->x - step_size,
   this->y); // west
82
83         return options;
84     }
85
86     /**
87      * @brief Equality comparison operator.
88      *
89      * @param p The Coordinate to compare with.
90      * @return bool
91      */
92     bool operator==(const Coordinate &p) const {
93         return (this->x == p.x) && (this->y == p.y);
94     }
95 };
96
97
98 #endif //ASSIGNMENT_2_COORDINATE_H
99
```

```
 1 //
 2 // maze_solver.h
 3 // Created by Kaitlyn Archambault on 2023-10-03.
 4 //
 5
 6 #ifndef ASSIGNMENT_2_MAZE_SOLVER_H
 7 #define ASSIGNMENT_2_MAZE_SOLVER_H
 8
 9
10 #include <vector>
11 #include "stack.h"
12 #include "maze.h"
13 #include "coordinate.h"
14
15 /**
16  * @class MazeSolver
17  *
18  * @brief A class for solving grid-based mazes using a
    stack data structure.
19  */
20 class MazeSolver {
21 private:
22     Stack<Coordinate<int>> m_successful_moves; //
   stores maze solution
23     std::vector<Coordinate<int>> m_attempted_moves;
   // stores all moves attempted
24     long m_elapsed_time{}; // how long it took to
   solve the maze
25
26 public:
27     /**
28      * @brief Default constructor.
29      */
30     MazeSolver() = default;
31
32     /**
33      * @brief Default destructor.
34      */
```

```
35      ~MazeSolver() = default;
36
37      /**
38       * @brief Solves a maze using depth first search and a stack
39       *
40       * The main algorithm loop will run until available path options are exhausted or a solution is found.
41       *
42       * @param maze The Maze object to be solved.
43       * @return `true` if the maze is able to be solved, `false` otherwise.
44       */
45      bool solve(Maze &maze);
46
47      /**
48       * @brief Tries available paths from a given position in the maze.
49       *
50       * Loops through adjacent options for the given position, attempts the first valid path, and
51       * updates both the attempted and successful moves.
52       *
53       * @param maze The Maze object to navigate.
54       * @param position The current position in the maze.
55       */
56      void try_available_paths(Maze &maze, Coordinate<int> position);
57
58      /**
59       * @brief Gets the elapsed time for solving the maze.
60       *
61       * @return The elapsed time in microseconds.
62       */
```

```
63      long get_elapsed_time() const;
64  };
65
66
67  #endif //ASSIGNMENT_2_MAZE_SOLVER_H
68
```

```
 1 //
 2 // main.cpp
 3 // Created by Kaitlyn Archambault on 2023-10-03.
 4 //
 5
 6 #include <iostream>
 7 #include <regex>
 8 #include "../inc/colours.h"
 9 #include "../inc/maze.h"
10 #include "../inc/maze_solver.h"
11
12 using namespace std;
13
14 bool is_valid_file_path(const string &
   provided_file_path);
15
16 int main(int argc, char *argv[]) {
17
18     // validate program arguments for maze and
   solution files
19     if (argc == 3) {
20         if (!is_valid_file_path(argv[1]) || !
   is_valid_file_path(argv[2])) {
21             cout << RED << "Must enter a valid .txt
   format file name for both the maze and solution file."
    << RESET << endl;
22             cout << CYAN << "Arguments should resemble
    \"./solve ../tests/test.txt ../solved/solution.txt\"
    " << RESET << endl;
23
24             return 1;
25         }
26     } else {
27         cout << RED << "Invalid arguments: should
   resemble \"./solve ../tests/maze.txt ../solved/
   solution.txt\" " << RESET << endl;
28
29         return 1;
```

```
30      }
31
32      string maze_file = argv[1];
33      string solution_file = argv[2];
34
35      // initialize maze with given file
36      Maze maze(maze_file);
37
38      // attempt to solve the maze and indicate the
    result
39      MazeSolver maze_solver;
40      if (maze_solver.solve(maze)) {
41          cout << GREEN << "Solved maze at " <<
    maze_file << " in " << maze_solver.get_elapsed_time
    () << " microseconds" << RESET << endl;
42          maze.save_to_file(solution_file);
43      } else {
44          cout << RED << "Could not find a solution.
    Make sure the provided file is a valid maze." << RESET
     << endl;
45      }
46
47      return 0;
48  }
49
50  /**
51   * Validate provided file path with regex
52   *
53   * @param provided_file_path file path input by user
54   *
55   * @returns true if the provided file path is in a
    valid format, false otherwise
56   * @note Expected file path format is "(/path/)
    file_name.txt"
57   */
58  bool is_valid_file_path(const string &
    provided_file_path) {
59      regex valid_file_path(R"(^(?:..)?\/?(?:[a-zA-Z0-9_
```

```
59 -]+[\/])?(?:[a-zA-Z0-9_-])+\.txt$)");
60
61     return regex_match(provided_file_path,
   valid_file_path);
62 }
```

```cpp
//
// maze.cpp
// Created by Kaitlyn Archambault on 2023-10-03.
//

#include "../inc/maze.h"
#include <fstream>
#include <iostream>

Maze::Maze(const std::string &maze_file) {

    m_maze_start = Coordinate<int>(1, 0); // assumes maze entrance is always the first cell on the left
    m_maze_end = Coordinate<int>(MAZE_SIZE - 2, MAZE_SIZE - 1); //assumes maze exit is always last cell on the right

    std::ifstream ifs;
    std::string line;

    //attempt to open the provided file and read its contents into the maze grid
    try {
        ifs.open(maze_file, std::fstream::in);

        int row = 0;
        while (getline(ifs, line)) {
            int column = 0;
            for (char ch: line) {
                //initialize the unsolved maze
                m_maze[row][column] = ch;
                column++;
            }
            //move to the next line
            row++;
        }
    } catch (std::ifstream::failure &e) {
        std::cout << "Exception reading source file"
```

```
34  << std::endl;
35     }
36 }
37
38 Coordinate<int> Maze::get_maze_start() const {
39     return m_maze_start;
40 }
41
42 Coordinate<int> Maze::get_maze_end() const {
43     return m_maze_end;
44 }
45
46 std::ostream &operator<<(std::ostream &output, Maze &
   maze) {
47     // print maze grid character by character
48     for (auto &i: maze.m_maze) {
49         for (char j: i) {
50             output << j << ' ';
51         }
52         //move to the next line
53         output << std::endl;
54     }
55
56     return output;
57 }
58
59 void Maze::set_solution_at(Coordinate<int> p, char ch
   ) {
60     m_maze[p.get_x()][p.get_y()] = ch;
61 }
62
63 char Maze::at(Coordinate<int> p) {
64     return m_maze[p.get_x()][p.get_y()];
65 }
66
67 bool Maze::contains(Coordinate<int> p) {
68     return p.get_x() < MAZE_SIZE &&
69            p.get_x() >= 0 &&
```

```
70              p.get_y() < MAZE_SIZE &&
71              p.get_y() >= 0;
72
73  }
74
75  void Maze::save_to_file(std::string file_name) {
76      std::ofstream ofs;
77      std::string line;
78
79      try {
80          //overwrite the file completely if it already
     exists
81          ofs.open(file_name, std::fstream::trunc);
82
83          //copy maze grid character by character into
    the file
84          for (auto &i: m_maze) {
85              for (char j: i) {
86                  ofs << j;
87              }
88              ofs << std::endl;
89          }
90
91      } catch (std::ofstream::failure &e) {
92          std::cout << "Exception writing to solution
    file" << std::endl;
93      }
94
95      std::cout << "Solution saved to " << file_name
     << std::endl;
96  }
97
```

```cpp
//
// maze_solver.cpp
// Created by Kaitlyn Archambault on 2023-10-03.
//

#include "../inc/maze_solver.h"

#include <iostream>
#include <chrono>

typedef std::chrono::high_resolution_clock Clock;

long MazeSolver::get_elapsed_time() const {
    return m_elapsed_time;
}

bool MazeSolver::solve(Maze &maze) {

    // start the timer
    auto start = Clock::now();

    // initialize starting position and solved status
    Coordinate<int> current_position = maze.get_maze_start();
    bool solved = false;
    m_successful_moves.push(current_position);

    while (!solved && !m_successful_moves.empty()) {

        // try to move forwards along an available path from the current position
        current_position = m_successful_moves.pop();
        this->try_available_paths(maze, current_position);

        // If the stack is empty at this point the maze can't be solved- don't allow .peek() on an empty stack.
```

```
34          if (m_successful_moves.empty()) break;
35
36          // check if the maze is solved:
37          if (m_successful_moves.peek() == maze.
    get_maze_end()) {
38
39              // stop the timer
40              auto duration = Clock::now();
41              m_elapsed_time = std::chrono::
    duration_cast<std::chrono::microseconds>(duration -
    start).count();
42
43              // add special characters to the maze
    solution file along the successful path
44              int stack_size = m_successful_moves.
    get_size();
45              for (int i = 0; i < stack_size; i++) {
46                  maze.set_solution_at(
    m_successful_moves.pop(), '#');
47              }
48
49              solved = true;
50          }
51      }
52
53      return solved;
54  }
55
56  void MazeSolver::try_available_paths(Maze &maze,
    Coordinate<int> position) {
57      // get an array of coordinates 1 step in each
    compass direction from the current coordinate
58      auto options = position.get_adjacent_coordinates(1
    );
59
60      // from the available options, only move to a
    space if it's valid and hasn't been attempted yet
61      for (auto &&option: options) {
```

```
62          if (maze.contains(option) &&
63              maze.at(option) == ' ' &&
64              std::find(m_attempted_moves.begin(),
    m_attempted_moves.end(), option) == m_attempted_moves.
    end()) {
65
66              // log valid option as attempted and
    successful, and return the last coordinate to the
    stack
67              m_attempted_moves.push_back(option);
68              m_successful_moves.push(position);
69              m_successful_moves.push(option);
70
71              // break after finding a valid move
72              break;
73          }
74      }
75  }
76
77
```