/* File: mazesolver.cpp */
* -------------------
* This program solves a maze by recursive backtracking, and compares
* this approach to iterative approaches using Stack (depth-first)
* and Queue (breadth-first).
* @author Cynthia Lee and Jerry Cain
* @version 1920-Aut */

#include <fstream>
#include <iostream>
#include <string>
#include "console.h"
#include "simpio.h"
#include "gwindow.h"
#include "direction.h"
#include "point.h"
#include "stack.h"
#include "queue.h"
#include "maze.h"
using namespace std;

/* Constants */
const int WINDOW_SIZE = 540;

/* Function prototypes */
bool solveMaze(Maze & maze, Point pt);
bool solveMazeStack(Maze & maze, Point start);
bool solveMazeQueue(Maze & maze, Point start);
Point adjacentPoint(Point start, Direction dir);

/* Main program */
int main() {
  GWindow gw(WINDOW_SIZE, WINDOW_SIZE);
  while (true) {
    Maze maze("SampleMaze.txt", gw);
    string response = getline("How do you want to solve the maze?
      Recursive, or Stack, or Queue?");
    bool foundSolution = false;
    if (response == "") {
      break;
    } else if (response == "r" || response == "R") {
      foundSolution = solveMaze(maze, maze.getStartPosition());
    } else if (response == "s" || response == "S") {
      foundSolution = solveMazeStack(maze, maze.getStartPosition());
    } else if (response == "q" || response == "Q") {
      foundSolution = solveMazeQueue(maze, maze.getStartPosition());
    }
    if (foundSolution) {
      cout << "The marked path is a solution." << endl;
    } else {
      cout << "No solution exists." << endl;
    }
    getline("[Press enter to continue]");
    cout << endl;
  }
  return 0;
}

/* Function: solveMaze */
* Usage: solveMaze(maze, start);
* -------------------------------
* Attempts to generate a solution to the current maze from the specified
* start point. The solveMaze function returns true if the maze has a
* solution and false otherwise. The implementation uses recursion
* to solve the submazes that result from marking the current square
* and moving one step along each open passage.
*/

//Maze functions:
// maze.isOutSide(pt)
// maze.isMarked(pt)
// pause(200)
// maze.markSquare(pt)
// looping over directions: for (Direction dir = NORTH; dir <= WEST; dir++) {
// maze.wallExists(pt,dir)
// maze.adjacentPoint(pt, dir)
// maze.unmarkSquare(start);
bool solveMaze(Maze & maze, Point start) {
    // base case: success
    if (maze.isOutside(start)) {
        return true;
    }
    // base case: failure
    if (maze.isMarked(start)) {
        return false;
    }
    maze.markSquare(start);
    pause(200);
    // recursive case: try other options
    for (Direction dir = NORTH; dir <= WEST; dir++) {
        if (!maze.wallExists(start, dir)) {
            if (solveMaze(maze, maze.adjacentPoint(start, dir))) {
                return true;
            }
        }
    }
    maze.unmarkSquare(start);
    return false;
}

/*/ 
* Function: solveMazeQueue 
* Usage: solveMazeQueue(maze, start); 
* ------------------------------ 
* Attempts to generate a solution to the current maze from the specified 
* start point. 
* * 
* THIS VERSION uses breadth-first search (BFS) rather than depth-first 
* search (DFS), which is what the recursive version does. This version 
* pairs with the explicit Stack version to demonstrate how using explicit 
* Queue or explicit Stack (otherwise identical code) results in BFS or 
* DFS, respectively. 
*/
bool solveMazeQueue(Maze & maze, Point start) {
    Vector<Direction> compass;
    compass += WEST, SOUTH, EAST, NORTH;
    Queue<Point> toExplore;
    toExplore.enqueue(start);
    while(!toExplore.isEmpty()){
        Point current = toExplore.dequeue();
        if (maze.isOutside(current)) return true;
        if (maze.isMarked(current)) continue;
        maze.markSquare(current);
        pause(200);
        for (Direction dir : compass) {
            if (!maze.wallExists(current, dir)) {
                toExplore.enqueue(maze.adjacentPoint(current, dir));
            }
        }
        return false;
    }
/*
* Function: solveMazeStack
* Usage: solveMazeStack(maze, start);
* ------------------------------
* Attempts to generate a solution to the current maze from the specified
* start point.
* 
* THIS VERSION works the same way as the recursive version (both execute
* a depth-first search), and demonstrates how use of the BFS code (notice
* this is EXACTLY the same code as the Queue version!) with a Stack
* instead of a queue results in a perfect re-creation of the recursive
* version, in terms of order of exploration. This is of course because
* recursion uses a Stack, implicitly, in the call stack.
*/
bool solveMazeStack(Maze & maze, Point start) {
    Vector<Direction> compass;
    compass += WEST, SOUTH, EAST, NORTH;
    Stack<Point> toExplore;
    toExplore.push(start);
    while(!toExplore.isEmpty()){
        Point current = toExplore.pop();
        if (maze.isOutside(current)) return true;
        if (maze.isMarked(current)) continue;
        maze.markSquare(current);
        pause(200);
        for (Direction dir : compass) {
            if (!maze.wallExists(current, dir)) {
                toExplore.push(maze.adjacentPoint(current, dir));
            }
        }
    }
    return false;
}