```
#include <iostream>
#include "point_stack.h"

using namespace std;

void print_maze(char **maze, int row_count, int col_count) {
  for (int row = 0; row < row_count; row++) {
    for (int col = 0; col < col_count; col++ ) {
      cout << maze[row][col];
    }
    cout << endl;
  }
}

// Read the maze from stdin, returning a 2-d array of characters.
// On exit, row_count contains the number of rows and col_count contains
// the number of columns.
char** read_maze(int &row_count, int &col_count) {
  char **result = NULL;
  cin >> row_count;
  cin >> col_count;
  cin.ignore();
  cout << "row_count:" << row_count << " col_count:" << col_count << endl;
  // Allocate a two-dimensional array of characters, that is,
  // an array of pointers, each of which is a pointer to an array of characters.
  result = (char**) malloc(row_count * sizeof(char*));
  // Allocate an array of characters for each row.
  for (int row=0; row < row_count; row++) {
    result[row] = (char*) malloc(col_count * sizeof(char));
  }
  // Initialize the maze array from stdin
  string line;
  int row = 0;
  while(getline(cin, line)) {
    for (int col = 0; col < line.size(); col++) {
      result[row][col] = line[col];
    }
    row++;
  }
  return result;
}

// Scans the given maze and returns the starting row and
// column via row_out & col_out.
// Returns true if a starting point was found, otherwise
// returns false and the output parameters are left unchanged.
bool get_starting_point(char **maze, int row_count, int col_count,
  int &row_out, int &col_out) {
    bool result = false;
    for (int row = 0; row < row_count; row++) {
      for (int col = 0; col < col_count; col++) {
        if (maze[row][col] == 'S') {
          row_out = row;
          col_out = col;
          result = true;
          break;
        }
      }
    }
    return result;
  }

  // Returns if the given row/col position is available to move to.
  bool can_go(char **maze, int row, int col) {
```

```
  return (maze[row][col] == ' ' || maze[row][col] == 'C');
}

// Marks the given row/col postion as visited.
void mark_visited(char **maze, int row, int col) {
  if (maze[row][col] != 'S')
    maze[row][col] = '.';
}

// Marks the given row/col postion as a dead-end.
void mark_deadend(char **maze, int row, int col) {
  if (maze[row][col] != 'S')
    maze[row][col] = 'X';
}

// Marks the given row/col postion as the current position.
void mark_current_location(char **maze, int row, int col) {
  if (maze[row][col] != 'S' && maze[row][col] != 'C')
    maze[row][col] = '@';
}

// Tries to find the cheese in the given maze.  If successful, row and col
// will contain the location of the cheese, and true is returned. Otherwise
// row and col are the last maze location visited and false is returned.
// If the starting location cannot be found in the maze, row and col are unchanged.
bool solve_maze(char **maze, int row_count, int col_count, int &row, int &col) {
  bool result = false;
  PointStack pointStack;
  if (get_starting_point(maze, row_count, col_count, row, col)) {
    while (maze[row][col] != 'C') {  // Done?
      cout << "row=" << row << ", col=" << col << endl;
      // IMPLEMENT THIS SECTION: BEGIN
        // If we can go left
        //  mark the current location as visited
        //  push the current location on the the pointStack
        //  move left one space
        // else if we can go right
        //  mark the current location as visited
        //  push the current location on the the pointStack
        //  move right one space
        // else if we can go up
        //  mark the current location as visited
        //  push the current location on the the pointStack
        //  move up one space
        // else if we can go down
        //  mark the current location as visited
        //  push the current location on the the pointStack
        //  move down one space
        // else mark this spot as a deadend
        //  Backtrack to previous location (pop pointStack)
        //  If we successfully backtracked, mark the new current location,
        //  otherwise we are stuck and we need to exit the while loop.
      // IMPLEMENT THIS SECTION: END
      print_maze(maze, row_count, col_count);
    }
    // Did we find the cheese?
    result = (maze[row][col] == 'C');
  } else {
    cerr << "Could not find starting point!" <<endl;
  }
  return result;
}

int main() {
  bool result = false;
```

```
    int row_count, col_count;
    char **maze = read_maze(row_count, col_count);
    print_maze(maze, row_count, col_count);
    int row,col;
    result = solve_maze(maze, row_count, col_count, row, col);
    return (result ? 0 : 1);
  }
```