Assignment 13: Ice sliding puzzle - the rocky version

Course 'Imperative Programming' (IPC031)

Make sure to start working on the assignment **before the lab** session,
otherwise you will be too late for the deadline

Important: Assignment 14 is a continuation of this assignment. As such it is vital you have working puzzle code, be it after feedback from your TA. Otherwise you will struggle to get started with assignment 14 next week, which is typically very important for the exam. Make sure to leave yourself enough time to improve upon your code after receiving feedback if need be, and to ask your TA for help as soon as possible if you still struggle to correct your code after receiving your feedback.

1 Background

The Ice Sliding puzzle was brought to our attention by Luko Maarsen. In such a puzzle, due to inexplicable circumstances, a flamingo finds itself trapped on a rectangular ice floe. On the ice floe immobile rocks can be found (there might also be no rocks at all). There is exactly one rescue location that the flamingo needs to reach. The flamingo can only move into the directions north, east, south, and west. As soon as the flamingo starts moving, it keeps sliding on the ice until it either slides off the ice or bumps into a rock, or reaches the rescue location. When bumping into a rock, the flamingo stands still immediately before the rock. The flamingo drowns if it slides off the ice. The puzzle is finished if the flamingo has reached the rescue location.

2 Learning objectives

After doing this assignment you are able to:

- Design a data structure for a puzzle game.
- Populate this data structure by loading data from a text file.
- Implement the puzzle rules and moves using this data structure.
- Write unit tests for the puzzle code.
- Implement a `main` function that allows a user to play the puzzle.

3 Assignment

On Brightspace, you find the file "`assignment-13-mandatory-files.zip`". It contains a number of text files with names "`challenge.m.nsteps.txt`", where `m` is the number of the challenge and `n` is the least number of steps required to solve the challenge. In these text files a puzzle starting configuration is described by a matrix of cells that have the following representation:

1. an empty ice cell by `'.'`
2. a rock by `'r'`
3. the flamingo by `'f'`, and if it is on the rescue location by `'F'`
4. the empty rescue location by `'x'`

Part 1: Data structures and output

Design data structures to represent the puzzle. You must use the `Puzzle` structure as your main data structure, though you are allowed to use and define more structures if need be. To aid testing, we represent puzzle configurations as a 2D grid of characters in the form of `vector<vector<char>>` constants. When importing a puzzle configuration from a text file, we first read the file and convert it to such a `vector<vector<char>>` value. The code for this conversion has already been given, and should not be touched. You must implement the `load_puzzle` function, which given a `vector<vector<char>>` value will populate your data structure. Note that the `vector<vector<char>>` value may not represent a valid puzzle configuration. You must validate that it does, and only return true in `load_puzzle` if the configuration is valid. Once you are able to load puzzle configurations, implement the `<<` operator on your `Puzzle` structure. You can then verify your code works correctly by loading a puzzle configuration, and displaying it to the console output.

Hint: We strongly discourage designing your `Puzzle` structure as simply storing a single member of type `vector<vector<char>>`, as this makes implementing the puzzle logic in part 2 much harder. Think about what data you need to be able to access easily, and accommodate for that in your design. Avoid using `char` types in general for your `Puzzle` structure, as you should try to separate your data structure from the textual representation we use when loading or displaying the `Puzzle` structure.

Part 2: Puzzle logic

Design and implement functions to implement the puzzle logic. As a bare minimum, you must be able to:

1. Check if a given puzzle state is solved (flamingo on rescue location).
2. Check if a given puzzle state is unsolvable (no moves can be made, for instance because the flamingo has drowned).
3. Move the flamingo to the north, east, south, and west.

Part 3: Writing unit tests

Design and implement unit tests to verify that your code to load puzzle configurations, and perform puzzle logic, is working correctly. You are given some puzzle configurations in the form of `vector<vector<char>>` constants. Some of them represent invalid puzzle configurations that should fail to load, whereas others are valid. When testing the puzzle logic, use your load puzzle function to populate your `Puzzle` data structure, then call the respective puzzle logic function, and finally check the result. In order to check if the result is correct when moving the flamingo, you will likely have to load another puzzle configuration, and compare that against the puzzle with the flamingo moved. This means you will have to implement the `==` operator on `Puzzle`, as we have done in prior weeks. You are free, and even encouraged, to add more puzzle configurations in the form of `vector<vector<char>>` constants in your testing.

Part 4: Implementing main

Complete the partially implemented `main` function. Once completed, a user should be able to load a puzzle configuration from a text file. If successfully loaded, the configuration is displayed, and the user can repeatedly choose which move to perform, to reset the puzzle to the initial state, or to quit the application. After every user action the (updated) puzzle state is displayed. If the flamingo drowns, the user is informed via a console message, after which the puzzle is reset. If the flamingo stops on the rescue location, the user is informed the puzzle has been solved, along with the amount of steps that were performed, after which the application quits.

4 Products

As product-to-deliver you upload to Brightspace:

- "`main.cpp`" that you have created with solutions for each part of the assignment.
- "`main_test.cpp`" that has been extended with unit tests for part 3 and each new function that you have developed in "`main.cpp`".

Deadline

Lab assignment: Friday, December 15, 2023, 23:59h