Assignment week 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 at least 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 a 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 a 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 a rescue location by '`F`'
4. an empty rescue location by '`x`'

Part 1: Data structures and equality testing

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 will implement the `load_puzzle` function in Part 2, which given a `vector<vector<char>>` value will populate your data structure. Once you have created the `Puzzle` data structure, implement the `==` operator for it.

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 4 much harder. Think about what data you need to be able to access easily, and accommodate for that in your design.

Part 2: Loading

Implement the `load_puzzle` function, which given a `vector<vector<char>>` value will populate your `Puzzle` 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.

Testing: Make sure the `loading` tests pass provided in "`main_test.cpp`".

Part 3: Output

Implement the `<<` operator on your `Puzzle` structure, which displays the current state of the puzzle. Use the same format that was used for the puzzle configurations you loaded from in Part 2.

Testing: Make sure the `output` tests pass provided in "`main_test.cpp`".

Part 4: 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 a rescue location).
2. Check if a given puzzle state is unsolvable (flamingo has drowned).
3. Move the flamingo to the north, east, south, and west.

Testing: The "`main_test.cpp`" file contains many test cases to test your puzzle logic functions. Some of them are only partially implemented, as you must add the code to perform the actual flamingo moves. These parts are clearly marked with `TODO` comments. Finish implementing all these tests, and then make sure the `is_solved`, `move_north`, `move_east`, `move_south`, `move_west`, and `is_solvable` tests pass.

Part 5: 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 a 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 fully implemented for Part 4 and extended with new unit tests for each non-trivial new function that you have developed in "`main.cpp`".

Deadline

Lab assignment: Friday, December 13, 2024, 23:59h

Important notes:

1. check that you have actually submitted your solution in Brightspace.
2. the deadline is firm, and it is impossible to upload a solution after expiry. In that case you fail the assignment.
3. you can upload solutions to Brightspace several times, but only the last submission is stored.
4. identify yourself and your lab partner in every uploaded document. The identification consists of your first and last names, student numbers, and number of (sub) assignment. By identifying yourself, you declare that the uploaded work has been created solely by you and your lab partner.
5. your work will be judged and commented upon. We expect that you obtain the feedback, read it, and use it to for the next exercises.
6. it is essential that you only submit your own solution, never copy somebody/something else's solution, and never share your solution—in particular: **AI tools (including but not limited to Github Copilot or ChatGPT) are not permitted**, solutions from previous year cannot be reused, and finally, you and your lab partner take joint responsibility for the assignment you submit.