CS246E—Assignment 4 (Fall 2023)

B. Lushman

Due Date 1: Friday, November 24, 5pm
Due Date 2: Tuesday, December 5, 5pm

Part of this assignment is due on Due Date 1; the rest is due on Due Date 2. See the assignment questions for specifics. All code is to be written in C++20. Use file inclusion instead of modules.

1. In this problem, you will use C++ classes to implement the game of Minesweeper. (`https://en.wikipedia.org/wiki/Minesweeper_(video_game)`). An instance of Minesweeper consists of an $n \times n$-grid of cells, each of which can be empty or contain a mine, though initially the presence or absence of a mine in each cell is not revealed. Once the board is set up, the player is advised of the number of mines on the board. The player then "presses" cells on the board, revealing their contents. If the cell contained a mine, the player loses. If the cell did not contain a mine, the cell now displays the number of adjacent cells that contain mines. If *no* cells adjacent to this cell contain mines, then the game "presses" all (up to) 8 adjecent cells on behalf of the player, and the pressed cells propagate until we encounter cells that are adjacent to mines. Then these cells display the number of adjacent mines, and the turn ends.

   Your implementation of Minesweeper must be built around the Observer pattern.

   To implement the game, you will use the following classes:

   - `class Subject` — abstract base class for subjects (see provided `subject.h`);
   - `class Observer` — abstract base class for observers (see provided `observer.h`);
   - `class Cell` — implements a single cell in the grid (see provided `cell.h`);
   - `class Grid` — implements a two-dimensional grid of cells (see provided `grid.h`);
   - `struct State` — provides a representation of a cell's state (see provided `state.h`
   - `class TextDisplay` — keeps track of the character grid to be displayed on the screen (see provided `textdisplay.h`);
   - file `info.h` structure definition for queries issued by the observer on the subject.

   The provided files implement the subject and observer classes as template classes, parameterized by two types: `State` and `Info`, both of which together are meant to encapsulate the fields of the subject. The intended distinction is that `State` would include those aspects of a cell's state that, if changed, should trigger a notification to observer objects, and that `Info` would include those aspects of a cell's state that are unlikely to change (such as its position in the grid). As such, the state is a field of the abstract subject, and the info is associated with the concrete subject. You are free to use other ways to arrange your objects' state if you want.

   Note: you may make minor changes to the public interface of these classes, so long as the spirit of the problem and the overall general structure of the solution are maintained.

   Your solution to this problem must employ the Observer pattern. Each cell of the grid is an observer of all of its neighbours (that means that class `Cell` is its own observer). Thus, when

the grid calls `notifyObservers` on a given cell, that cell must then call the `notify` method on each of its neighbours (each cell is told who its neighbours are when the grid is initialized). Moreover, the `TextDisplay` class is an observer of every cell (this is also set up when the grid is initialized). A subject's collection of observers does not distinguish what kinds of observers are actually in the collection (so a cell could have arbitrary cells or displays subscribed to it). When the time comes to notify observers, you just go through the collection and notify them. As a hint for implementation, consider how the rules of the game above can be replicated using an observer pattern where a notification can either be a notification of a button press, a reply indicating whether the replying cell has a mine, or a message that no mines were found, and the search should propagate outwards. A cell that has been pressed can notify its observers (adjacent cells and displays) that it has been pressed. The notified cells can broadcast a response, indicating whether or not they have a mine. Only the cell that was pressed would care about the reply (and do something in reaction to it), but all observers would hear it. Remember that the subject is not allowed to selectively send state updates only to certain observers, because it is not supposed to know anything about its observers; rather it sends its state updates to *all* observers, and it is the observer that will decide whether it cares about that particular state update.

As a guide, we recommend that your cells have *at least* the following states (you may require more):

- `Initial` – How each cell starts out.
- `Pressed` – The cell was just pressed. Neighbouring cells should respond by broadcasting a message saying whether or not they have a mine.
- `Reply` – The cell is replying to a message sent by a cell in state `Pressed`, indicating whether it has a mine or not.
- `Propagate` – The cell has discoverd that it has no neighbours with mines. This is a signal to neighbouring cells that they should act as if they were pressed.
- `Done` – It is the end of the interaction, and the cell's state has been fully resolved.

You are to overload `operator<<` for the text display, such that the entire grid is printed out when this operator is invoked. Each empty cell prints as `-` and a cell that has been pressed displays the number of its neighbours that have mines (this will be a number from 0 to 8).

When you run your program, it will listen on stdin for commands. Your program must accept the following commands:

- `new n` Creates a new $n \times n$ grid,where $n \geq 4$. If there was already an active grid, that grid is destroyed and replaced with the new one.
- `setmine r c` Before the game starts, places a mine at row `r`, column `c`. If the row and column entered correspond to a cell that already contains a mine, or a position outside the grid, then the input is ignored and nothing is done (this does not need to be tested). You may assume that this command will be issued at least once, i.e., that a game does not start with no mines.
- `start` Starts the game. The board is displayed, along with the number of mines on the board. Commands `new` and `setmine` are no longer valid.
- `press r c` Presses the cell at row `r`, column `c`. This command cannot be issued until after the `start` command has been invoked.

The program ends when the input stream is exhausted, or when the game is over. The game is over when the number of empty cells equals the number of mines on the board (a win), or when a cell containing a mine is pressed (a loss).

When the game is over, print either `You win.` or `You lose.` to stdout, as appropriate. If input was exhausted before the game was won or lost, display nothing. A sample interaction follows (responses from the program are in italics):

```
new 5
-----
-----
-----
-----
-----
setmine 1 1
start
Number of mines: 1
press 1 2
-----
--1--
-----
-----
-----
press 1 3
--100
--100
11100
00000
00000
press 0 0
1-100
--100
11100
00000
00000
press 0 1
11100
--100
11100
00000
00000
press 1 0
11100
1-100
11100
00000
00000
You win.
```

Note: Your program should be well documented and employ proper programming style. It should not leak memory. Markers will be checking for these things.

A strong word of advice: Get your program working without propagation of 0's first. Propagation is difficult to get right. We would suggest that if a cell is asked to press itself as a result of a propagation, that it not do that immediately, but rather register itself with a "to-do list" observer. Then once the inital press is settled, main can run through the to-do list until all follow-up actions are completed.

Due on Due Date 1: Test suite (`suite.txt`).

Due on Due Date 2: Submit your solution. You must include a Makefile, such that issuing the command `make` will build your program. The executable should be called `a4`.