Assignment week 6: Conway's Game of Life

Course 'Imperative Programming' (IPC031)

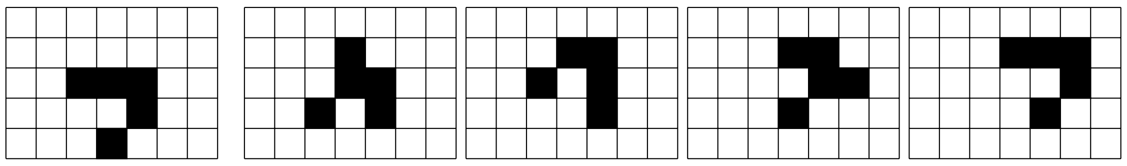
Make sure to start working on the assignment **before the lab** session, otherwise you will be too late for the deadline

1 Background

In this assignment you work with text files and one-dimensional and two-dimensional arrays. You develop an implementation for a theoretically interesting problem known as the "Game of Life", introduced by John Conway in 1970. Conway's problem illustrates the concept of a cellular automaton invented by John von Neuman, in the 1940s. The "Game of Life" is played in a two-dimensional grid of cells, called the universe. It is played by creating an initial configuration and observing how this configuration evolves step by step. Each cell of the universe is in one of two possible states: dead or alive. Every cell interacts with its eight neighbors, which are the cells that are horizontally, vertically, or diagonally adjacent to it. In each step, a transition occurs that determines the state of the cells in the next configuration of the universe, on the basis of their state in the current configuration. The transition rules are amazingly simple:

1. A live cell with fewer than two live neighbors dies, as if caused by under-population.
2. A live cell with two or three live neighbors lives on to the next generation.
3. A live cell with more than three live neighbors dies, as if by overcrowding.
4. A dead cell with exactly three live neighbors becomes a live cell, as if by reproduction. Otherwise the cell remains dead.

Despite their simplicity, these rules can lead to surprisingly complex configurations of live cells. Here is a small example of a frequently occurring pattern (the glider pattern):



initial configuration after one step after two steps after three steps after four steps

Before going any further, use desktop testing and reproduce the transitions in the diagrams (check for at least one complete transition).

On Brightspace the file "`assignment-06-mandatory-files.zip`" contains a number of text files with "Game of Life" configurations. The above sequence is found in the files "`glider0.txt`" up to "`glider4.txt`". The other patterns are "`10_cell_row.txt`", "`pulsar.txt`", and "`Gosper_glider_gun.txt`". The text files are formatted as follows: there is a fixed number of text lines (`NO_OF_ROWS`). Each line has the same number of cell characters (`NO_OF_COLUMNS`). Each cell character is either `'*'` indicating a live cell, or `'.'`, indicating a dead cell. Additionally, there are files "`main.cpp`" and "`main_test.cpp`" that you extend in this assignment.
The universe in Conway's Game of Life is unbounded. This is a luxury that we can not afford. In this assignment, a bounded universe is represented by means of a two-dimensional array that contains `NO_OF_ROWS` rows of `NO_OF_COLUMNS` `Cell` values:

```
enum  Cell {Dead=0, Live};      // a cell is either Dead or Live (we rely on the fact that Dead has a zero value)
const char DEAD          = '.'; // the representation of a dead cell (both on file and screen)
const char LIVE          = '*'; // the representation of a live cell (both on file and screen)
const int NO_OF_ROWS     = 40;  // number of rows    (height) of the universe (both on file and screen)
const int NO_OF_COLUMNS  = 60;  // number of columns (width) of the universe (both on file and screen)
```

Any cell 'outside' of the bounded universe is considered to have value `Dead`. Hence, all cells at the edges of the bounded universe have at least three `Dead` neighbor cells 'outside' of the bounded universe (five of them for corner cells, and three of them for other edge cells).

2 Learning objectives

After doing this assignment you are able to:

- Define pre- and postconditions;
- Work with text files for input and output purposes;
- Work with arrays;
- Work with pre- and postconditions.

Every function must be documented with preconditions given as `assert` statements and postconditions given as comments.

3 Assignment

Part 1: Pre- and postconditions

For each of the below functions, develop the most precise preconditions and postconditions. Add your pre- and postconditions to the corresponding functions in “`main.cpp`”.

1.
```
int compare (int a, int b)
{
    if (a < b)
        return -1;
    if (a > b)
        return 1;
    return 0;
}
```

2.
```
enum Direction {North, East, South, West};
bool opposite (Direction a, Direction& b)
{
    switch (a)
    {
        case North: b = South; return true;
        case East:  b = West;  break;
        case South: b = North; break;
        case West:  b = East;  break;
    }
    return false;
}
```

3.
```
void write_numbers_to_file (int number, ofstream& out_file)
{
    while (number >= 0)
    {
        out_file << number << '␣';
        number--;
    }
}
```

4.
```
void read_numbers_from_file (ifstream& infile)
{
    int number;
    infile >> number;
    while (!infile.fail ())
    {
        cout << number;
        infile >> number;
    }
}
```

5.
```
int compare (int a [], int b [], bool c [], int n)
{
    int x = 0;
    for (int i = 0; i < n; i++)
    {   if (a[i] == b[i])
        {
            c[i] = true;
            x++;
        }
        else
            c[i] = false;
    }
    return x;
}
```

Part 2: Game of Life

Part 2.1: Get cell in bounded universe

Design and implement a function that returns the cell value of a bounded universe at some given coordinate, if the coordinate is a valid value. If the coordinate is not a valid value (it is 'outside' of the bounds of the bounded universe), the function returns the cell value `Dead`.

```
Cell cell_at (Cell universe [NO_OF_ROWS][NO_OF_COLUMNS], int row, int column)
```

Testing The file "`main_test.cpp`" contains only a limited number of tests for `cell_at`. Add a number of additional tests to achieve proper test coverage.

Part 2.2: Setting the scene

Design and implement a function that displays the universe on the console and a function that reads a "Game of Life" configuration from a file (*e.g.*, "`glider0.txt`").

```
void show_universe (Cell universe [NO_OF_ROWS][NO_OF_COLUMNS])
bool read_universe_file (string filename, Cell universe [NO_OF_ROWS][NO_OF_COLUMNS])
```

The function `show_universe (universe)` displays the rows of `universe` one by one, each on a new line. Cells of value `Dead` must be displayed as the character `DEAD`, and cells of value `Live` as the character `LIVE`. The function `read_universe_file (filename, universe)` attempts to open the text file with file name `filename` and, if successful, attempt to read the content of that text file and fill the `universe` bounded array representation, line by line in the rows of `universe`. The `bool` result is true if and only if no error has occurred.

Testing The file "`main_test.cpp`" contains both positive and negative tests for `read_universe_file`. Make sure that all tests pass for your implementation. It is not required to add additional tests, but if you add auxilary functions it is recommended to add tests for those.

Part 2.3: The next generation

Design and implement a function that computes the next configuration of the universe according to the "Game of Life" rules.

```
void next_generation (Cell now [NO_OF_ROWS][NO_OF_COLUMNS], Cell next [NO_OF_ROWS][NO_OF_COLUMNS])
```

The array `now` is the current configuration of the universe, the array `next` contains the next configuration of the universe. Use `cell_at` (Part 2.1) to retrieve the correct (neighbor) cell values in `now`.

Testing Add tests for `next_generation` to "`main_test.cpp`". You should test that your implementation is correct for the glider pattern. The file "`main_test.cpp`" contains a hint how to write such tests.

Part 2.4: The final program

Implement the `main` function such that your program asks for a universe configuration file name, computes one transition step for this file, and shows the resulting universe on the console.

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 contains your own tests for `cell_at` and `next_generation`. You should also add non-trivial unit tests for each new function that you have developed in "`main.cpp`".

Important

From now on, in all assignments, including the bonus, you must document every function with preconditions (using `assert`) and postconditions (using comments). This message will not be repeated in later assignments.

Deadline

Lab assignment: Friday, October 11, 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.