1 Introduction

For this assignment, we will be programming a simulation called the Game of Life. Much of the code is provided for you (available on canvas), but some key blocks of code have been omitted. Your job is to fill in the missing pieces. Your working code is due on canvas at **11:59pm Thursday November 3rd**. This handout begins with an overview of the game, then discusses some details of how the code that we are providing works, and finally goes into detail about each of the three places that you need to fill in missing code.

2 The Game of Life

The Game of Life is a simulation designed by mathematician John Conway in 1970. It is an example of a *cellular automata*: the game is made of a grid of cells, and each cell lives and dies based on what its neighbors do. This is not a game in the normal sense. No one plays it. Instead, it is a simulation that plays out in interesting ways. It starts chaotically, but as the game runs interesting patterns often emerge.

Overview. The game board is a n-by-n grid. Each square of the grid is a *cell*, and cells are either alive (on) or dead (off). The game begins with each cell in a random state, and then turns of the game continue happening forever. For example, suppose that a game started like this:

```
. X X .
. . X .
. X . X
X . X X
```

The `X`'s are cells that are alive, and the `.`'s are cells that are dead.

Each game turn we count up how many alive neighbors each cell has. Here are the numbers for our example:

```
3 4 4 5
3 4 4 3
4 3 5 4
4 5 5 4
```

Notice that the maximum number of alive neighbors that a cell can have is 8, and the minimum number is 0. Also, the game board wraps around on the top and bottom. So, every cell has exactly eight neighbors. The cell in the top row, second column from the left has eight neighbors too (but three of which are in the bottom row). See later in this handout for more details about how the board wraps.

In each turn we decide whether a cell ends up alive or dead according to these rules:

- If the cell begins alive
  - If the cell has only 0 or 1 living neighbors, it dies of under-population
  - If the cell has 2 or 3 living neighbors, it continues to live
  - If the cell has 4 or more living neighbors, it dies of over-population
- If the cell begins dead
  - If the cell has exactly 3 living neighbors, it becomes alive by reproduction
  - Otherwise, the cell remains dead.

In our example, this means that after one turn the board now looks like

```
X . . .
X . . X
. X . .
. . . .
```

3 Implementation Plan

The big question here is how to store the board. An array seems natural, but the game board is 2D. Two-dimensional arrays are a bit messy, and the haven't covered them in class yet. We will take a *flattening* approach.

Our game boards will always be square, with size `board_size` by `board_size`. We will give each cell `(row, column)` coordinates. Here is a picture for `board_size = 5`.

(0,0)	(0,1)	(0,2)	(0,3)	(0,4)
(1,0)	(1,1)	(1,2)	(1,3)	(1,4)
(2,0)	(2,1)	(2,2)	(2,3)	(2,4)
(3,0)	(3,1)	(3,2)	(3,3)	(3,4)
(4,0)	(4,1)	(4,2)	(4,3)	(4,4)

In order to store this board in memory, we will reserve one long array of `ints`:

```
int* board = new int[board_size * board_size];
```

Here is the connection between our 1D array and our 2D board: We give every 2D location a 1D index using this formula:

```
index = r * board_size + c;
```

Here's a picture of our flattened indices along with the 2D board coordinates:

(0,0) 0	(0,1) 1	(0,2) 2	(0,3) 3	(0,4) 4
(1,0) 5	(1,1) 6	(1,2) 7	(1,3) 8	(1,4) 9
(2,0) 10	(2,1) 11	(2,2) 12	(2,3) 13	(2,4) 14
(3,0) 15	(3,1) 16	(3,2) 17	(3,3) 18	(3,4) 19
(4,0) 20	(4,1) 21	(4,2) 22	(4,3) 23	(4,4) 24

4 Your Tasks

This section of this handout looks at each of the three code blocks that you need to write.

A Quick Heads-Up. As you read through the code, you might find a few unfamiliar lines. In particular, there are two include directives:

```
#include <chrono>
#include <thread>
```

as well as one complicated line in `main` that are only there to let us pause the program after every update. This lets us see the output of the program as frames of an animation. If you are curious, feel free to comment

out the delay line and see what happens without it. You are not responsible for knowing about these libraries, or how to use them to add delays.

4.1 Task: Initialization

When we create a new game board (in main) like this:

```
int* board = new int[board_size * board_size];
```

we get an array called `board` that is the right size for our game, but our array comes with arbitrary values in it. In the function `initializeBoard`, we loop through the entire array and set every value to either `0` or `1` (randomly). Here is a way of saying that formally, by saying what should be true before and after we call the function.

`initializeBoard(int* board, int board_size)`	
Preconditions:	• `board_size` is the size of our game board.
	• `board` is a `board_size * board_size` array of `ints`
Postconditions:	• The values in `board` have been changed to either `0` or `1` (chosen randomly)

Your first job is fill in the function definition of `initializeBoard`. Right now, it is completely empty!

Hint: To help you out with this, look at the `printBoard` function. The code pattern could be helpful, but also, it will print your board using these symbols:

cell value	symbol printed
1	`X`
0	`.`
else	`?`

This means that if you ever see a `?` get printed, you messed up! That could be helpful with debugging.

4.2 Task: Indexing

Your second task is to write the function `rc2index`. This function finds the index into the `board` array that corresponds to any given (row, column) coordinate. Recall our scheme for representing a 2D array in a 1D array:

(0,0) 0	(0,1) 1	(0,2) 2	(0,3) 3	(0,4) 4
(1,0) 5	(1,1) 6	(1,2) 7	(1,3) 8	(1,4) 9
(2,0) 10	(2,1) 11	(2,2) 12	(2,3) 13	(2,4) 14
(3,0) 15	(3,1) 16	(3,2) 17	(3,3) 18	(3,4) 19
(4,0) 20	(4,1) 21	(4,2) 22	(4,3) 23	(4,4) 24

So for example, `rc2index(4, 2, 5)` returns `22`, because for a board of size `5`, the element in row `4` and column2 gets index `22`. (Notice that rows and columns are numbered in a zero-indexed way, from `0` to `board_size - 1`).

The formula for this conversion is just

```
index = r * board_size + c
```

where (`r`, `c`) are the row/column coordinates of any cell.

However, there is another complication to this function. Remember, the board wraps around the top and the bottom, so for `board_size` 5, (2,0) and (2,4) are next to each other, and (0, 4) and (4, 4) are next to each other. There are a couple of ways that we could have incorporated this into the program. Here is the design choice that we made:

`rc2index(int r, int c, int board_size)`

Preconditions:	• `board_size` is the size of our game board.
	• `r` is an `int` with $-1 \leq r \leq$ `board_size`
	• `c` is an `int` with $-1 \leq c \leq$ `board_size`
Postconditions:	• If `r` equals `board_size`, then `r` gets changed to `0`.
	• If `c` equals `board_size`, then `c` gets changed to `0`.
	• If `r` equals `-1`, then `r` gets changed to `board_size - 1`.
	• If `c` equals `-1`, then `c` gets changed to `board_size - 1`.
	• After these wrapping adjustments, the function returns `r * board_size + c`

You need to fill in the function definition of `rc2index` according to this specification.

Here are some more examples of how this function should work:

r	c	board_size	index
3	2	5	17
4	4	5	24
4	5	5	20
2	-1	5	14
3	2	4	14
2	2	2	0
2	1	2	1

4.3 Task: Update Logic

The final piece of code that you have to write is the key piece of logic for the game. Look at function `updateBoard`. The job of this function is to apply the rules (see Section 2) to decide whether each cell is alive or dead in the next round of the game.

We've given some of this function, but we left out the core logic. Here is what happens. The function starts by creating a (temporary) new game board called `new_board`. This represents what the board will look like in the next time step. Our job in this function is to set each cell in `new_board` to either `0` (dead) or `1` (alive). Then at the end of the function, the final thing we do is copy the new board into the old board. That's how we take one turn.

Between creating `new_board` and copying `new_board` into `board`, there is a double loop that looks at every cell in the game. Inside this loop, it makes a decision based on (1) whether that cell is alive in `board`, and (2) how many of it's neighbors are alive in `board`.

So, your code goes inside the double loop. Here is the specification:

`updateBoard` **inner loop**	
Preconditions:	• `board[index]` is `0` or `1` depending on whether the cell is currently alive or dead • `num_neighbors_on` is an `int` between `0` and `8` containing the number of neighbors of this cell that are alive
Postconditions:	• `new_board[index]` is `0` or `1` depending on whether this cell should be alive or dead in the next turn, according the the Game of Life rules described in Section 2.