Discovering Requirements

- Implement conway's game of life
  - 4 rules
    - Occupied Cell has 0 or 1 neighbor, dies
    - Occupied Cell has 4+ neighbors, dies
    - Empty cell has == 3 neighbors, new cell spawned
    - Births+deaths happen all at once
  - 80x22 console board
  - Allow for three starting patterns (oscillator, glider, glider gun)
  - Allow user to "place" starting positions
  - Allow user to advance grid
  - Allow user to stop generations
- Reflection

Design

I will make a few classes to aid in my design:

- Board in Board.cpp and defined in Board.h
- Point in Board.cpp and defined in Board.h (small enough that it doesn't need another file)
- Cell in Cell.cpp and defined in Cell.h

In addition, I will have a main.cpp that provides the main menu and game loops.

On initialization, the program should create an empty board. The board class will take a size pair to define how large it should be. It will (internally) add a margin to this size so that we can have things smoothly slide off the board.

Once we are initialized and have an empty board, I will ask the user what pattern they would like displayed. I'm planning on implementing the oscillator, glider, and glider gun. These patterns will be invoked by calling a method on the board class that has the co-ordinates predefined (with an adjustment modifier so the user can set an offset).

Once the coordinates have been set on the board, we should display the board to the user. They can advance the board, or specify a number of times the board should advance. Advancing the board uses the following algorithm:

```
foreach column in the grid (defined as i)
    foreach row in the grid (defined as j)
        aliveCells = call functionToGetNeighborCells(i, j)
        if cell(i,j) is empty
            if there are three live cells, we should set this cell to be alive on the next generation
        else
            if there are less than two neighbor cells, we should set this cell to die
            else, if there are more than three neighbor cells, we should set this cell to die
```

Loop again over every cell (using the same foreach pattern) and change every cell's state to its nextstate.

After the board is advanced, we should print it out.

Allow the user to pick again, or exit the simulation.

Allow the user to pick a new simulation, or exit the program.

Testing and debugging

There are enough implementations of the game of life, that one could compare each iteration of their game of life against known good implementations on the internet. Additionally, The rules of the game of life work in such a way that if a small (easy to compare) pattern, like the glider, is functioning correctly, a larger pattern (like the glider gun) should work as well. I verified that my glider was working iteration for iteration. I also verified that my glider gun was functioning properly and did not produce strange artifacts. This remained the case even after 10,000 iterations.

Reflection

I was sure to read the requirements on day one and start thinking about them before I started actually sitting down and designing what I eventually coded. I think that this made for a very smooth experience. Having a large outline of what my code was supposed to do made it very easy to break things into smaller functions that did little bits of work.

I think one of the most annoying things was having to create each pattern by hand in the code. I should have just made the user do it :) (just kidding…. but really….). This was one thing that I didn't anticipate being hard, but it was quite time consuming. I ended up making a helper function that could write to the board any given vector of coordinates. Then I just constructed the vector by hand. I couldn't really think of a better way to do it.

Making sense of the output was pretty easy, because it either matched what the next generation should be, or it didn't. I did all the logic part of my coding first (before implementing the UI), so I was pretty confident that everything was working in my algorithms after the first hour of writing or so. I have found that I dislike writing user interface code… it took way longer than any of the actual internal logic I made.