Design and Testing Reflections
Collin James

My original design looked like this:

```
<pseudo-code>

Class Oscillator
    private members
        startPosition (supplied by user, x/y)
        array* cells[][] / array* newCells[][]
        rowSize = 3;
        colSize = 3;

    public members
        countNeighbors() -- the main algorithm
        drawCells() -- draws on screen
        updateCycle() -- copies new array into original;
                        called from drawCell
</end pseudo-code>
```

I started with Oscillator and decided to adapt the next class (glider) from it. Those two at least were sufficiently close to allow this. I didn't know if I would get to the gun, which would need a significantly larger array to complete. I planned to make countNeighbors() basically if/else statements that corresponded to the instructions in the assignment.

When it came time to code I separated startPosition into to two ints and for awhile I had a static constant called SIZE that I used to create the static arrays. Yes, I started coding with static arrays even though my design called for dynamic. I eventually moved to dynamic, which I'll talk about later.

I honestly had no idea how I was going to draw the cells, other than that I needed to use ncurses and that it would mostly be done from within the class. Using ncurses was a big pain, and maybe the hardest part of this assignment for me. The main algorithm for countNeighbors() came together pretty quickly. I added some new functions as I saw fit to separate some of logic and reduce the complexity of my planned functions. These included initArrays() (used in the constructor) and initWindow (), which is used from main() to draw a blank window. I also added a clearNewArrays() function at this point. I eventually decided to make several functions private when it became clear that they would only be used within the class. I added a destructor to clean up the ncurses cruft.

My testing plan for Oscillator worked like this: first make sure I could draw a blank "window" of '-' signs on the terminal screen; then try to get a stationary Oscillator on there so I could test my initArrays() function; then try to make it oscillate using the countNeighbors function (if that worked, it meant it could be reused on the future classes); and finally test the user input of x and y coordinates. Of course, in main I had to make sure that the prompts and inputting of variables was taken care of correctly.

My plan for modifying my Oscillator class for the Glider class was to keep the countNeighbors(), initWindow(), and clearNewArrays() functions, but to alter initArrays() for the glider pattern. I knew I would also need to alter drawCells() and updateCycle() for my approach, which was to have a 4x4 array that shifted itself whenever necessary. Then in my drawing function I would include logic to make the glider move based on how the array was shifting itself. I did this based on the animated gif file of a glider from wikipedia.org. I eventually added a clearCurrentArray() function to the class to abstract some new logic from the updateCycle. In my planning I also added new private variables xMove and yMove, to track the movement of my glider.

To test the Glider class, I first made sure that I could get all the states of a stationary glider to draw. This took some trial and error with the placement of initial live modules in initArrays() (I had done extensive drawings of the placement and movement of the modules with pencil and paper while looking at the animated gif), but because the countNeighbors() logic was solid, it actually came together pretty quickly once the initial pattern was correctly set. Next I tested whether I could make it move with a combo of updated logic in updateCycle() and drawCells(). Because of my approach, I didn't need to worry about what happened at the edges of the "window", as the array would just try to draw itself outside of its bounds with no side effects. It simply didn't display.

At this point I realized that my Oscillator and Glider classes had several things in common, and I decided to challenge myself to create a base class to reduce repetition. To do this I realized that I would have to go back to my initial design of dynamic arrays; I would also need to make my static constant SIZE a regular variable since the size of the arrays would need to be different for each class. I included all of the variables for each class in the base Cell class, which means that Oscillator gets some extra, unused variables. 4 common functions and the constructors and destructors are used by the child classes, leaving 3 classes to be handled by the child classes.

This was hard to test; it took a lot of coaxing to get the dynamic arrays initialized (and properly destroyed with an updated destructor), but once those were working everything else in the classes fell into place and worked as well as it had before the base class.

The plan for the Gun class was to use a much larger array, 38x38, to allow for all of the movement. Since the gun is aiming down, and that section of the array will be off of the window, meaning I shouldn't have to worry about what happens to it. I would need to change the initArrays() function quite a bit to place the pattern correctly. This involved a lot of || operators in my if statement, as with the Glider class. Since the base class and the countNeighbors() logic seemed to be solid, once I got everything placed correctly it should work well.

Main testing for the Gun class involved using static views of the pattern to make sure everything is placed correctly. Then the countNeighbors() needed to be tested to make sure it behaves correctly in a bigger array. After that, I needed to make sure that the gliders fly off the bottom of the window and don't rebound. And finally test on flip.

I stuck pretty close to my original design for the Gun class (2 paragraphs above). The countNeighbors logic was fine. I was able to simplify the drawCells() function quite a bit because of the larger array size; I didn't have to manually move the array around the grid. I greatly simplified the updateCycle() function for the same reason. The main testing involved making sure that I had placed the cells in the correct coordinates in the array. Everything worked correctly after that, although as I said I was able to simplify some functions.

What I hadn't foreseen was the big difference between compiling on OS X vs. flip. I forgot to compile from my CentOS virtual machine on my computer, so when I compiled on flip, my program generated a segmentation fault! I don't know why I didn't get this error on OS X. In any case, that led to a bit of pointer research, and finally to a rewrite of my handling of main class pointers in main.cpp. I initialized all of my objects to null at first; at the end of the function, I made sure to only delete the objects that had been re-instantiated with actual data. That cleared up the segmentation fault issue.

I really liked the way that this program came together. I was particularly proud of the fact that I didn't need to change the countNeighbors() algorithm throughout the program. I'm also proud of the fact that for each of the cell objects, you can choose any coordinates (even negative) and if you choose the glider or gun at least you will eventually see it on the grid. My only dissatisfaction was with the way the program looked on flip; the movement looks much worse than on my test computer, as if flip is skipping frames (?). Maybe it is because I am viewing over ssh. For future programs I want to continue to get better at thinking more long-term in my design; I know that I'm still too eager to "get coding" when I should probably still be planning.