Assignment 1 – Program Design and Testing

Implementing Conway's *Game of Life*

Goals -

- Review of programming with arrays
- Convert requirements (i.e. the rules) to a software design
- Think about the testing process, before and after writing the program.

Assignment -

You will design, implement, and test a program that runs the game of life. Conway's *Game of Life* is a standard example of a cellular automaton. This means that you have an array or matrix of cells. Each turn or generation the value of each cell may change based upon its neighbors. To get more background look online, such as: http://en.wikipedia.org/wiki/Conway's_Game_of_Life.
Please do the research so you understand the terms and how the rules are applied.

WARNING - Use the Internet to ONLY research how the 'game' is structured and designed. Design and write your own program. **Do NOT use any code that you did not write for yourself**. That would be cheating. Much of what is available is more complicated or advanced than what is required here. It is harder to use it than to write your own. At the end of the assignment you will find a suggestion on how to design and develop this program for yourself.

The rules are: Each cell has eight neighbor cells. The neighbors are the cells directly above, below, to the right, to the left, diagonally above to the right and left, and diagonally below to the right and left.

1. If an occupied cell has zero or one neighbor, it dies of loneliness.
2. If an occupied cell has more than three neighbors, it dies of overcrowding.
3. If an empty cell has exactly three occupied neighbor cells, there is a birth of a new cell to replace the empty cell.
4. Births and deaths are instantaneous and occur at the changes of generation.

NOTE - Think this through carefully before you start coding. If you change a cell it can impact those around it. You should determine the changes required for all cells before changing any cell in the display!

There are significant initial questions that you must address in your design. We will limit the visible "world" to 40 x 20 cells. That's a lot of characters to type in from the keyboard so don't make the user do it. You must give the user the option for one of three starting shapes in the grid; an oscillator, a glider, and a glider cannon.

You will need to handle the cells on the edges. Remember this is a window on an infinite grid. The patterns go on forever. You are just showing them while they are visible in this small window. You may not be able to just stop the patterns at the edge as that may change the behavior. Specifically, any pattern should not change as the object moves out of the visible grid.

After you have everything designed, coded, and tested add an option for the user to specify the starting location of the selected pattern. They will specify a single cell. Make it clear which cell of the pattern they are using. Otherwise they could put a glider at an edge and the glider moves away and they never see it!

You will create your design document BEFORE you start coding. The TA will grade, in part, on how well your implementation matched your design. Remember, in your reflections document you can explain how you had to change the design because your first idea, just didn't work. Just explain what you learned. Simply stated, program design is identifying the problem to be solved, identify the inputs, specify the desired output, and then develop the algorithm(s) to convert the input into the output.

You must also describe how you tested your program. In this case you must demonstrate that the rules work for all cells. The edges will be a special case. You have 2 types of horizontal edges and two types of vertical edges. Each corner involves a different combination of these edges. Your reflections must describe how you planned to test these cases and the results of the testing.

By the due date at midnight, you will submit your program, and the reflections document, which will discuss how well your design worked, and the results of the testing. You must also provide a makefile. If you do not provide a makefile it will not be graded. All the files must be submitted in a zip archive.

WARNING - Submitting any code you did not write is a violation of the university's academic dishonesty policy. **Keep in mind, if you can find it using Google, the TA's can too.**

GRADING -

- Programming style and documentation (**10%**)
- Build the array/matrix providing a fixed simple oscillator (**30%**)
- The display allows the user to see the change(s) in the shape(s) (**10%**)
- Allow the user to specify the starting location of the pattern (**10%**)
- Add the glider pattern (**5%**)
- The glider should simply disappear off an edge (**10%**)
- Add a glider gun or cannon pattern (**5%**)
- Reflections document to include the design description, test plan, test results, and comments about how you resolved problems during the assignment (**20%**)

IMPORTANT POINT - Keep in mind this is a programming assignment. You are not writing a program to share with others for their enjoyment. Too often students forget this and the attempt to make it entertaining which also makes it more difficult. And has little to do with the grading. You are graded on developing the design, writing the code, and testing your program. You will use text-based output. It may not be pretty, but it will demonstrate your design and implementation works. If you have other programming experience and want to use something else check with your grader before doing it.

HINTS:

1. Use the grading breakdown to plan your program. Use incremental development. Get one part working. Test it. Save a copy and continue working on the next step in your plan. This ensures you will have something to submit!
2. On the Internet you will also find many descriptions and examples of stable patterns. Definitely take advantage of that for your testing!
3. There are only 4 rules that you apply to each cell in your array. There are emergent properties so testing logic errors may be difficult. :-)
4. For cells on the edges you need to handle the case that you are displaying a subset of an infinite grid. Get your program working first, ignoring the edge condition. Once you know your code is working then implement the edge algorithm(s). One option is to have some ghost columns and rows. You can continue calculating the movement, but just not display some of the cells.
5. As one website stated, the *Game of Life* is one of the most programmed games in the world. Be very careful about borrowing any code, or ideas you see in someone else's code. As always, any code submitted must be yours and yours alone.

Suggested plan to design and code this project:

First. Get a pencil and some paper. Seriously. Low tech but it works. Develop an algorithm to apply the rules. The rules are defined in terms of how many neighbors are alive. At a minimum you must poll each of 8 neighbors. For a given cell with indices i and j how are you going to access each neighbor ([i-1, j-1], [i-1, j] and so on)? Now that you can read the neighbors you must develop the selection logic to determine the state of the current cell (i.e. unchanged, goes on, turns off). Keep in mind the 2D array will start at 0,0. If you are getting errors, make sure your math does not step out of the bounds of the array. Use an if-else or switch structure to implement the logic. You're not finished with the design yet!

As also discussed in the overview, there is a problem with calculating the state of the array to display. You can't just use a single array. Why? As soon as you change the contents of a cell it will change the logic in one or more of its neighbors. Each of them will impact their neighbors. If you don't defer the changes you quickly end up with garbage! Your design challenge is coming up with a method to save the state, or phrased another way, you need to have each change saved as pending before you draw the new matrix/array to the screen. I've heard students use at least 2 different methods. How do you solve this problem?

Now that you have your algorithms developed convert them to pseudocode or flowcharts are whatever system you chose to represent program logic. Walk through your pseudocode. Did you miss any details? It's much easier to find and fix logic errors now.

Now it is time to dig out the keyboard and start entering code. ☺