CPSC 350 – Data Structures
Spring 2017
Assignment #2 – Game of Life
Due 2-28-17 11:59 pm

Overview:

Now that we have gone through the trouble of discussing the "theory" of arrays, we might as well put them to good use in what I hope will be an interesting assignment. Your chore for this week is to implement the game of Life, which should give you the opportunity to work on a non-trivial program and have some fun at the same time.

The game of Life was designed in the 1970s by the mathematician J.H. Conway. The game gained popularity after appearing in a *Scientific American* article, and took the computing world by storm. The game is a simulation that models the life cycle of bacteria, providing entertainment wrapped up in what some would call mathematical elegance. (For some history on the game, as well as the rise of computing in general, I highly recommend the book *Hackers*, by S. Levy.)

The game itself is played on a two-dimensional grid. Each grid location is either empty or occupied by a single cell (X). A location's neighbors are any of the cells in the surrounding eight adjacent locations. (At this point you should be thinking about 2-d arrays.)

Rules of the Game:

The simulation starts with an initial pattern of cells and computes successive generations according to the following rules:

1. A location that has one or fewer neighbors will be empty in the next generation. If a cell was in that location, it dies of loneliness. (The fate of many a computer scientist…)
2. A location with two neighbors remains stable. If there was a cell, there's still a cell. If it was empty, it's still empty.
3. A location with three neighbors will contain a cell in the next generation. If it currently has a cell, the cell lives on. If it's empty, a new cell is born.
4. A location with four or more neighbors will be empty in the next generation due to overcrowding.
5. The births and deaths that take one generation to the next must all take place simultaneously. When computing a new generation, the births and deaths in that generation **can not** affect other births and deaths in that generation. Not adhering to this guideline will really mess up your simulation, so be careful. An easy way to get around this is to have 2 versions of the grid. One is for the current generation, and the other is for computing the next generation (based on the current generation) without side effects. When you are done computing the next generation you can either copy it into the current generation grid, or just switch a reference. (In the database community, this little hack is called shadow paging.)

Boundary Conditions:

One of the gray areas of the program is how to calculate neighbors for cells on the grid boundaries. To make things interesting, at startup your program will ask the user which of the following 2 modes they want to run the simulation in, and calculate accordingly:

Classic mode: All locations off the grid are considered to be empty.

Doughnut mode: The grid is wrapped around itself horizontally and vertically, resulting in a torus (doughnut) shape. In this mode, any reference off the right-hand-side is wrapped around to the left-most column of the same row and vice-versa. Any reference off the top wraps to the bottom in the same column. Corners are best illustrated with an example:

n			n	
n			n	n
n			n	n

In the above, the shaded square has 8 possible neighbors (depending if the cells are occupied), marked with an 'n'.

Mirror Mode: References off the grid are bounced back as though the wall were a mirror. This means the reflection of a cell could count as its own neighbor. For a corner cell, if it was occupied, its reflection would count as 3 neighbors…1 for the vertical reflection, l for the horizontal reflection, and 1 for the diagonal reflection. As an example of mirror mode:

		x		
			x	x
x				x
		x	x	

The "x" stands for an occupied cell. If the shaded cell were empty, then it would have 4 neighbors. If it were occupied, it would have 5 neighbors, since the neighbor directly above the shaded square is a reflection of the shaded square itself.

The Program:

When the program starts, you should ask the user if they wish to provide a map file of the world in which the simulation will be carried out, or if they would like a random assignment. If they want a random assignment, prompt for the dimensions of the world. Then prompt for a decimal value (greater than 0 and less than or equal to 1) representing the initial population density of the world. Randomly generate the initial cell

configuration using the dimension and density inputs. If the user wants to provide a map file, then prompt for the file path. The file should be a text file with the following configuration:

```
5
7
---X--X
-X--X--
-----XX
X-X-X--
-----XX
```

The first line is height of the grid (number of rows) and the second is the width of the grid (number of columns). The following lines represent rows of the grid. An X stands for an occupied cell. A dash (-) represents an empty cell.

Once a generation is calculated you will output it to standard out as a text grid. (Again, a "-" represents an empty grid and a "X" represents an occupied grid.) Before each grid you should output the generation number, starting with 0 for the initial configuration. It is EXTREMELY important that you follow this convention or there will be issues with the automated test scripts I use for grading.

In order to make it easy for the user to see what is going on in the program, you should ask the user at the beginning of the program if they want a brief pause between generations (see the system("pause") function), if they want to have to press "Enter" to display the next generation, or if they want to output everything to a file in which case they should be prompted for a file name.

To summarize, when the program is run it should:

1. Ask the user if they wish to start with a random configuration, or specify a flat-file configuration. Prompt appropriately depending on the response.
2. Ask the user what kind of boundary mode to run in.
3. Ask the user if they want a brief pause between generations, if they want to have to press the "enter" key, or if they want to output to a file.
4. Carry out the simulation
5. If the simulation is infinite, just keep running. But if the world becomes empty or stabilizes, the simulation should halt and ask the user to press "enter" to exit the program.

Programming Strategy:

Implementing the game is not an altogether simple task. An elegant solution will take some time, and so it is important for you to begin ASAP. You should know that if you plan out your program ahead of time, the solution can be surprisingly concise.

Besides getting experience with arrays, this program also offers practice at program decomposition…learning how to break up your code into manageable modules. With the

right decomposition, the program can be much easier to write, debug, and comment. The wrong decomposition can make writing the game a nightmare. Take your time to plan your strategy before you code…it will pay off. (In other words, use an OO approach…don't just do everything in main()) Also, make sure to develop and debug incrementally. With a small program it is often easy to write all your code and then try to debug it. This is unlikely to be the case for this program. Do yourself a favor and test as you go along.

Requirements:

-You may work in pairs on this assignment.
-All code must be your own. Please cite any references you use in a README file.
-You may develop on any platform you wish, but please make sure your code compiles and runs correctly under Cygwin on Windows, since I will use this platform to test your solution.

Grading:

As usual, you will be graded on correctness, elegance of solution, and your adherence to the above requirements. Style and comments are also important, so be aware that a well-documented, clean solution will receive more credit than a sloppy solution without comments.

Due Date:

Submit your .tgz per the submission instructions file via the course website by 11:59 pm on 2-28-17. (Groups need to submit a single copy – include all of the last names in the tgz file name.) The README should contain your names, student Ids, and any comments you have to make about your solution. And of course you should provide Makefiles so I can build your code by typing make from the top-level directory.

Acknowledgement: This assignment was inspired by a similar assignment developed by Mehran Sahami at Stanford University.