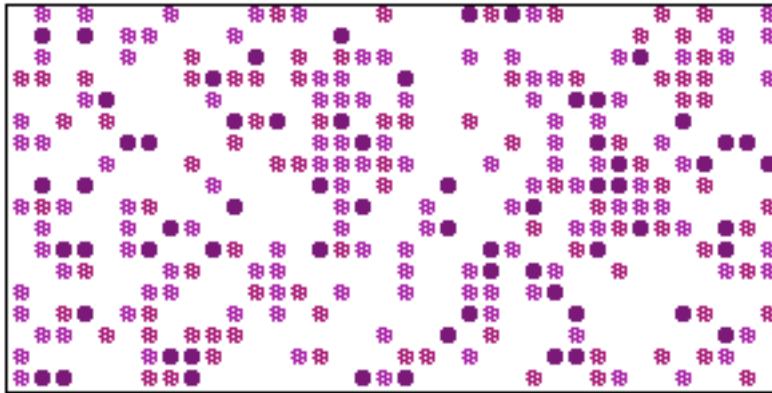


Assignment 1: The Game Of Life

A brilliant assignment from Julie Zelenski.

Let the fun begin! Your first real assignment centers on the use of a two-dimensional grid as a data structure in a cellular simulation. It will give you practice with control structures, functions, templates, and even a bit of string and file processing. More importantly, the program will exercise your ability to decompose a large problem into manageable bits. What's especially nice about this program is that you can construct a beautiful and elegant solution if you take the time to think through a good design. It can be as lovely to look at the code as it is to watch it run.

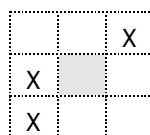


Due: Friday, October 2nd at 11:00 a.m.

The Problem

Your mission is to implement a hyped-up version of the game of Life, originally conceived by the British mathematician J.H. Conway in 1970 and popularized by Martin Gardner in his *Scientific American* column. The game is a simulation that models the life cycle of bacteria. Given an initial pattern, the game simulates the birth and death of future generations using simple rules. Think of it as a Lava Lamp for mathematicians. (Yeah, mathematicians need to get out a little more often – but that's not important right now...)

The game 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 cells in the surrounding eight adjacent locations. In the following example, the shaded middle location has three "live" neighbors:



The Rules

The simulation starts with an initial pattern of cells on the grid and computes successive generations of cells 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. Sad!
2. A location with two neighbors remains "stable"—i.e. if it contained a cell, it still contains 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 were unoccupied before, a new cell is born. If it currently contains a cell, the cell remains. Good times.
4. A location with four or more neighbors will be empty in the next generation. If there was a cell in that location, it dies of overcrowding. Bad times.
5. The births and deaths that transform one generation to the next must all take effect simultaneously. Thus, when computing a new generation, new births and deaths in that generation cannot affect other births and deaths in that generation. If this condition is violated, the order that you happen to go through the grid computing the next generation will affect the answer you get. To keep the two generations separate, you will need to work on two versions of the grid—one for the current generation, and a second that allows you to compute and store the next generation without impacting the current grid.

Check your understanding of these rules with the following example. The two patterns below should alternate forever:

	X	X	X	

		X		
		X		
		X		

Boundary Conditions

The definition of "neighbor" becomes a little fuzzy when you're working with locations along the boundary. What about an edge or corner cell that has walls for neighbors? To make our version of Life extra nifty, we allow the user to select one of the following three boundary strategies:

Plateau: Consider all locations off the grid to be empty.

Donut: Wrap the grid around on itself both vertically and horizontally. The world becomes a *torus* or donut shape. Any reference that disappears off the right-hand side is mapped around to the leftmost column of the same row. Similarly, any reference past the top reappears on the bottom, and so on. As a specific example, suppose you are considering location (0, 3), which is shaded in the 5 by 5 grid below. In plateau mode, this square has just 3 live neighbors, but in donut mode, it has 5:

		X		
			X	X
X				X
		X	X	

Corners are a little tricky in this mode. Here's another example to clarify. On this board, 'N' marks the 8 possible neighbors of the shaded square in donut mode:

N			N	
N			N	N
N			N	N

Mirror: References off the grid are bounced back as though the wall were a mirror. Sometimes this means the reflection of a cell gets counted as its own neighbor. In the picture below, the shaded square has 4 neighbors in mirror mode if the shaded square itself is blank.

		X		
			X	X
X				X
		X	X	

If the shaded square above is alive, it has 5 neighbors (since the "neighbor" directly above the shaded square is a reflection of the square itself). A corner cell counts as its own neighbor three times in mirror mode—once each for the reflections in the horizontal and vertical walls and once again for the diagonal reflection.

The Starting Configuration

To get the colony underway, your program should offer the user a chance to seed the grid randomly or read in a colony configuration file. If the user requests a random start, use the functions in the random number library to choose arbitrary cells to be alive (each with a 50% chance of being alive). If the user instead wants to start with a known configuration, you read the starting colony from a text file.

<pre># Any line that begins with a # # is a comment and is ignored 30 25 -----XXXXX-----XXXXX----- -----XXXXX----- -----XXXX-----XXXX----- ... and so on ...</pre>	<pre><- Number of rows <- Number of columns per row <- Each line is one row of the grid <- X means cell is live, dash is not</pre>
--	--

You can read the file line-by-line using the standard **getline** method (remember this is distinct from the CS106-specific **GetLine** that just reads from the console). All colony files obey the following format, and you may assume that the files are properly formatted (i.e. it is not required that you do error-checking).

Managing the Grid

Basically what's needed for storing the grid is a two-dimensional array. However the built-in C++ array is fairly primitive. C arrays (and thus C++ arrays) are long on efficiency but low on safety and convenience. They don't track their own length, they don't check bounds, and they require messing with pointers and dynamic memory and so on. We're taking a more modern approach—that of using a **Grid** class that provides the abstraction of a two-dimensional array in a safe, encapsulated form with various conveniences for the client. Just as the **Vector** provides a cleaner and less error-prone abstraction for a one-dimensional array, the **Grid** offers the same upgrade for two-dimensional needs. These classes, among others, are from the CS106 tool set that we will be using throughout the quarter.

You can read about the specifics of class interface in the **grid.h** header file available in the starter project. The class has member functions to set/retrieve the grid dimensions and set/retrieve values for each location, plus some operators to enable client conveniences such as assigning one grid to another. Because the class is supplied in template form, you will need to take care to include the specialization when declaring Grid variables, e.g. **Grid<string>** or **Grid<int>**.

Your grid should store an integer for each location rather than a simple alive-or-dead Boolean value. This allows you to track the age of each cell. Each generation that a cell lives through adds another year to its age. A cell that has just been born has age 1. If it makes it through the next generation, that cell has age 2, on the following iteration it would be 3 and so on. During the animation, cells will subtly lighten as they age to depict the cell life cycle.

As mentioned earlier, you will need two grids: one for the current generation and a separate scratch grid to set up the next generation without interfering with the current one. After you have computed the entire next generation, you can copy the scratch grid contents into the current grid to get ready for the next iteration. Copying a grid is trivial—just assign one grid to another. The **Grid** class implements the regular assignment operator to do a full, deep copy. Yay, client convenience!

Animating the World

We provide the **lifeGraphics** module that exports routines to help you draw the cells in the graphics window. Our cell-drawing function will slowly fade cells as they age. A newly born cell is drawn as a dark dot and with each generation the cell will slightly fade until it stabilizes as a faint cell. See the **lifeGraphics.h** interface in the starter files for details on the specifics of the routines we provide and how to use them.

The extended graphics library (**#include "extgraph.h"**) function **UpdateDisplay** can be used to immediately flush any drawing to the screen. The function **Pause** can be used to wait a bit in order to allow you to see what happened before going on. You should offer the user a couple of options for simulation speed; this will translate to shorter or longer pauses between generations. We also want you to support a manual mode where the user has to explicitly hit return to advance to the next generation. This mode is particularly handy when you are first learning how the rules operate or need to do some careful debugging.

You should continue computing and drawing successive generations until the colony becomes completely stable or the user indicates he or she is done by clicking the mouse button (via **MouseButtonIsDown** function from the **extgraph.h** library). If the colony has evolved to a configuration that has no changes between generations (other than live cells continuing to age) and all cells hit the constant **MaxAge**, you should end the simulation. Colonies that infinitely repeat or alternate are not considered stable and thus the user will need to click to end those. When a simulation ends, you should then offer the user a chance to start a new simulation or quit the program entirely.

Program Strategies

A high-quality solution to this program will take time. It is not recommended that you wait until the night before to hack something together. It is worthwhile to map out a strategy before starting to code. Amazingly concise and elegant solutions can be constructed — aim to make sure yours is one of them!

Decomposition: This program is an important exercise in learning how to put decomposition to work in practice. Decomposition is not just some frosting to spread on the cake when it's all over—it's the tool that helps you get the job done. With the right decomposition, this program is much easier to write, to test, to debug, and to comment! With the wrong decomposition, all of the above will be a chore.

As always, you want to design your functions to be of small, manageable size and of sufficient generality to do the different flavors of tasks needed. Strive to unify all the common code paths. You should be sharing as much code as possible between the three modes. Starting from the top in your decomposition, consider at what point the difference in mode will necessitate that the code branch off into three separate code paths. You want to postpone that split as long as possible, allowing you to make maximum re-use of the simulation code.

A good decomposition will allow you to isolate the details for all tricky parts into just a few functions. Most of the complex details concern handling the boundary conditions. You should encapsulate the intimate details of the different modes in a small function that allows the higher-level functions to simply call it to obtain neighboring values without concern for those details.

Avoid special cases, don't handle inner cells, edge cells, and corner cells differently—write general code that works for all cases. No special cases means no special-case code to debug! Think carefully about how to design these functions to be sufficiently general for all use cases.

Incremental Development and Testing: Don't try to solve it the entire task at once. Even though you should have a full design from the beginning, when you're ready to implement, focus on implementing features one at a time. Start by implementing a basic Life simulation, using the plateau mode boundary. Seed your grid with a known configuration to make it easier to verify the correctness of your simulation. Initialize your grid to be mostly empty with a horizontal bar of three cells in the middle. When you run your simulation on this, you should get the alternating bar figure on page 2 that repeatedly inverts between the two patterns.

It's easiest to get your program working on such a simple case first. Then you move on to more complicated colonies, still in plateau mode, until it all works perfectly. At this point, perhaps you add support for the other two boundary strategies. Next you could work on generating random starting colonies and reading configuration files. If you build and test your program in stages, rather than all at once, you will have a easier time finding and correcting your errors.

Avoid the "extra-layer-around-the-grid": At first glance, some students find it desirable to add a layer around the grid—an extra row and column of imaginary cells surrounding the grid. There is some appeal in forcing all cells to have exactly eight neighbors by adding "dummy" neighbors and setting up these neighbors to have the right values for the current boundary strategy. However, in the long run this approach introduces more problems than it solves and leads to confusing and inelegant code. Before you waste time on it, let us tell you up front that this is a poor tradeoff and a strategy to avoid.

Other Random Notes and Hints

- Error-checking is an important part of handling user input. Where you prompt the user for input, such as asking for file to open or a boundary mode, you should validate that the response is reasonable and if not, ask for another response.
- While you are in development, it is helpful to circumvent your code that prompts for the configuration and just force your simulation to always open "Simple Bar" in plateau mode, a random colony, or whatever you are currently working on. This will save you from having to answer all the prompts each time when you do a test run.
- Although you might be tempted, **you should not use any global variables** for this assignment. Global variables can be convenient but they have a lot of drawbacks. In this class we will use them only sparingly and with good reason. For any of our assignments, you can assume that global variables are forbidden unless we explicitly tell you otherwise.

Coding Standards, Commenting, Style

This program has quite a bit of substance, however, the coding complexity doesn't release you from also living up to our style standards. Carefully read over the style information we have given you and keep those goals in mind. Pay close attention to the feedback your section leader provides on your first assignment. Aim for consistency. Be conscious of the style you are developing. Proofread. Comment thoughtfully.

Test Colonies

We have concocted several colony configuration files for you to use as test cases. There is also a compiled version of our solution, so that you can see what the correct evolutionary patterns should be. Some of our provided colonies evolve with interesting kaleidoscopic regularity; a few are completely stable from the get-go, and others eventually settle into a completely stable or infinite repetition or alternation. Some are good for testing plateau, others for mirror or donut. Each file has a comment at the top of the file that identifies the contributor and tells a bit of what to expect from the colony. Here are a few of the descriptions, for the others, open the data file and read the comments:

SimpleBar	Simple alternating bar
StableMirror	Colony of cells clinging along border that is completely stable in mirror mode
Dinner Table	Intricate symmetric colony infinitely cycling between 10 different patterns
Fish	Little colonies that endlessly swim across grid in donut mode
Glider Gun	A colony of cells that repeatedly fires off gliders heading southwest

We'd love additional colony contributions, so if you create a beautiful bacteria ballet of your own, please send it to us and we can share it with the class!

Getting Started

Follow the Assignments link on the class web site to find the starter bundle for Assignment #1. You should download this file to your own computer and open the bundle. There is a version for the Mac, and another for the PC. For this assignment, the bundle contains our **lifeGraphics.cpp** and an incomplete version of **life.cpp**.

Deliverables

You only need to print and electronically submit those files that you modified, which in this case, will probably just be **life.cpp**. You are to submit both a printed version of your code in class, as well as electronic version. Details of the submission process will come early next week. Both are due by the **beginning of lecture**. Please be sure that your pages are stapled firmly together and that the printout is clearly marked with your name and your section leader's name.

You are responsible for keeping a safe copy to ensure that there is a backup in case your submission is lost or the electronic submission is corrupted. Never turn in your only copy!