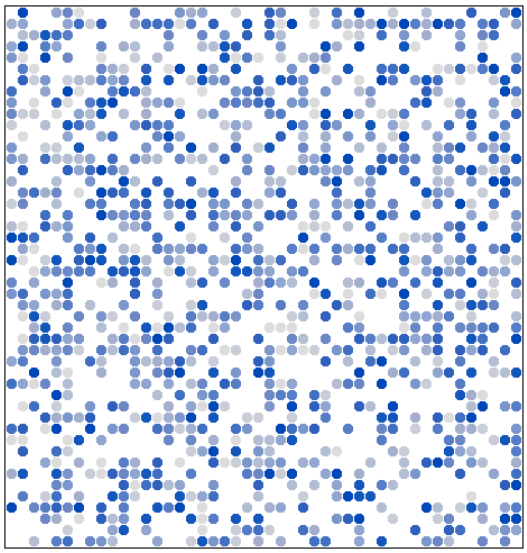
Assignment 1: The Game of Life

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 work. More importantly, the program will exercise your ability to decompose a large problem into manageable components. What's especially stellar about this problem is that you can construct a beautiful and elegant solution if you take the time to think through a good design.

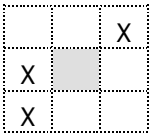


Project Due: Wednesday, October 2nd at 5:00 p.m.

The Problem

Your mission is to implement a 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. It's largely a lava lamp for mathematicians.

The simulation is run within 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 eight adjacent locations. In the following example, the shaded location has three "live" neighbors:

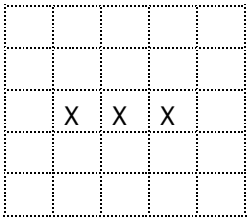


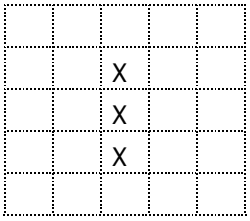
The Rules

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 zero or one neighbors will be empty in the next generation. If a cell was in that location, it dies of loneliness.
2. A location with two neighbors is **stable**—that is, 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 was 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.
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 don't impact other births and deaths in that generation. You'll 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 changing the current one.

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





The Starting Configuration

To get the colony underway, your program should offer the user the option to start everything off with a randomly generated grid, or to read in a starting grid from data file. If the grid is to be randomly generated, then set the grid width to be some random value between 40 and 60, inclusive, and the grid height to be some random value between 40 and 60, inclusive, and flip a fair coin to decide whether a particular cell should be occupied or unoccupied. If a cell is occupied, then set its age to be some random value between 1 and **kMaxAge**, inclusive.

If the user prefers to load a starting configuration from a file, then you can bank on the following file format:

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

You can read the file line-by-line using the standard **getline** method (note that this is distinct from the CS106-specific **getLine** that just reads from the console). All colony files obey the above format, and you may assume that the file contents are properly formatted (i.e. it is not required that you do error checking). When read from a file, all cells start at an age of one.

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 big on efficiency but low on safety and convenience. They don't track their own length, they don't check bounds, and they require you manage computer memory in a way that isn't all that fun in an introductory assignment. 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** (our equivalent of the Java **ArrayList**) provides a cleaner and less error-prone abstraction for a one-dimension 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. Refer to the Queen Safety lecture examples if you want to see the **Grid** in action.

Your grid should store an **integer** for each location rather than a simple alive-or-dead Boolean. 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 fade to light gray 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 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 using **=**. The **Grid** class implements the regular assignment operator to do a full, deep copy.

Animating the World

We provide the **life-graphics.h** module that exports a **LifeDisplay** class to help you draw the cells in its own window. Our cell-drawing methods 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 gray cell. See the **life-graphics.h** interface in the starter files for details on the specifics of the methods we provide and how to use them.

You should offer the user 3 options for simulation speed (slow, medium and fast); this will translate to shorter or longer pauses between generations. You decide how long the pause times are. Use what you think is reasonable. Make sure you have 3 distinct speeds. 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 the rules or need to do some careful debugging.

You should continue computing and drawing successive generations until the colony becomes completely stable or the user clicks the mouse button. Stability is defined as no changes between successive generations (other than live cells continuing to age) and all cells hit or exceed the constant **kMaxAge** (defined in **"life-constants.h"**). Colonies that infinitely repeat or alternate are not considered stable. When advancing to the next generation, you should check if the colony has become stable and if so, stop the simulation. If a non-manual mode has been selected, then if the user clicks the mouse button, the simulation should stop. Note, if the user has selected manual mode for the simulation speed, there is no need to check for user clicks. Instead, in manual mode, if the user enters **"quit"** instead of just hitting enter at the prompt, the simulation should stop. When a simulation ends, you should then offer the user a chance to start a new simulation or quit the program entirely.

Start Early

This assignment is likely bigger and more complicated than anything you built during your time in CS106A or AP Java. To ensure that you're not blindsided by the sheer amount of work, we highly suggest that you start early on this assignment.

By the end of this weekend, you might have a partially working program that:

- prints out all of the text that gets printed to the console, as illustrated by the sample application,
- prompts the user to initialize a grid based on the contents of a data file,
- populates a grid by reading in the contents of the named file, and

- draws the initial state of the grid to the graphics window, waiting for a mouse click to clear the screen and start over.

In a nutshell, the checkpoint implements the setup and all of the plumbing for the game of Life, without playing the actual simulation. Should you find yourself struggling with this suggestion, please feel free to come to office hours or get help in the LAIR.

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 yours one of them!

Decomposition: This program is an important exercise in learning how to put decomposition to work. 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, test, and debug! With the wrong decomposition, all of the above will be a chore. 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. A good decomposition will allow you to isolate the details for all tricky parts into just a few functions.

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 equals 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 off by writing code to meet the list of goals in the Start Early section. Continue by seeding 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 3 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, until it all works perfectly.

Avoid the "extra-layer-around-the-grid" approach: 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 always be empty. 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 strategy and should be avoided. Just assume that locations off the grid are empty and can never count as neighbors.

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, you should validate that the response is reasonable and if not, ask for another. (You can assume that the data files themselves are well formed, though.)
- While developing, it's helpful to circumvent the code that prompts for the configuration and just force your simulation to always open **"Simple Bar"** 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 seem convenient, but they have a lot of drawbacks. In this class we will never use them. Always assume that global variables are forbidden unless we explicitly tell you otherwise. (Global constants are fine, though.)
- Note that all of the data files reside in a directory called **"files"**. When opening a data file name **"Fish"**, you need to type in the filename as "**files/Fish**" else the **ifstream** constructor won't be able to find it.
- Allowing the game to animate while constantly polling for mouse clicks is a tricky thing. Look at the **"gevents.h"** documentation to see how one can poll for mouse events without blocking. I don't want you to invest too much energy into mouse and timer events, so I'm presenting my own approach to non-blocking mouse event detection right here.

```
static void runAnimation(LifeDisplay& display, Grid<int>& board, int ms) {
    GTimer timer(ms);
    timer.start();
    while (true) {
        GEvent event = waitForEvent(TIMER_EVENT + MOUSE_EVENT);
        if (event.getEventClass() == TIMER_EVENT) {
            advanceBoard(display, board);
        } else if (event.getEventType() == MOUSE_PRESSED) {
            break;
        }
    }
    timer.stop();
}
```

You should cannibalize the code snippet above and extend it as necessary as you fold it into your own solution.

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. Aim for consistency. Be conscious of the style you are developing. Proofread. Comment thoughtfully.

Test Colonies

We have provided 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. 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.

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. For this assignment, the bundle contains some custom header files, our **`life-graphics.cpp`** implementation, and an incomplete version of **`life.cpp`**.

Deliverables

You only need to electronically submit those files that you modified, which in this case, will probably just be **`life.cpp`**. There are instructions on the course web site outlining how to electronically submit on the assignments page. 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. You are also responsible for checking to make sure your submission was correctly received by checking your submitted assignments on the submission site (it should show you past submissions and the files they contain). Make sure your submission contains the correct files.