

ReadyPython SP19 Project

Game of Life

Adapted from Stanford CS106B assignment, the game of life is a fun simulation of a bacteria colony.

1. Read the specifications for CS106B project to get a basic understanding of the game. Focus less on the language-specific details (the Stanford project is in C++), but more on:

- the rules of the game
- the flow of the main program
- the way input file is formatted

2. Download project files from Google Drive or OneDrive. There are easy, medium, and hard versions available, with descending amount of skeleton codes.

3. In the project directory, type `python3 game_of_life_demo.py` to play a Game of Life. Note that the available world files can be found in `worlds` folder. Input the name of the file (extension included) when prompted.

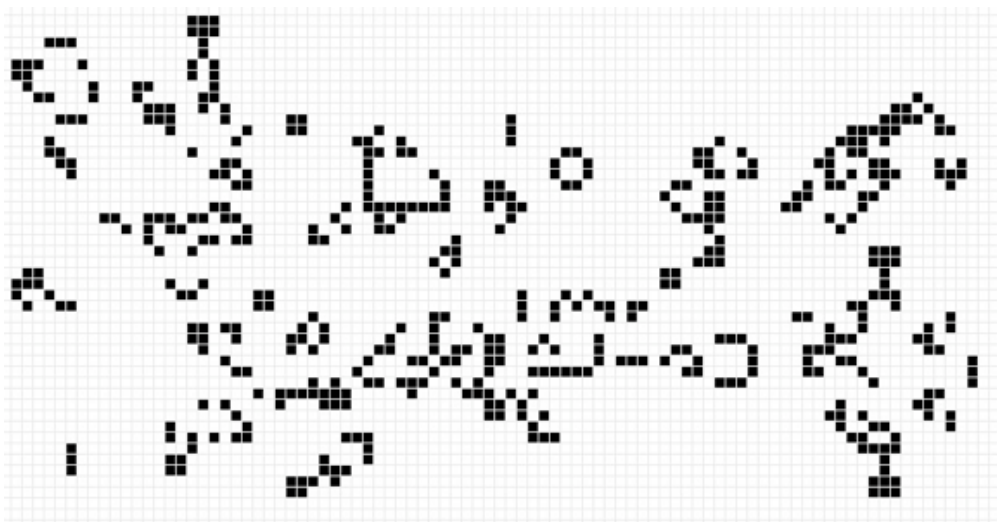
4. Start working on your own project! When unsure about how your program should behave, refer to the demo program provided.

Contact me if you have any questions.

Assignment 1: The Game of Life

Thanks to Julie Zelenski for the idea and Jerry Cain, Keith Schwarz, Cynthia Lee and Marty Stepp for revisions.

JUNE 26TH, 2017



A still image of the Game of Life

For your first assignment... We make the Game of Life!

The purpose of this assignment is to gain familiarity with basic C++ features such as functions, strings, and I/O streams, using provided libraries, and decomposing a large problem into smaller functions. This is an individual assignment. You should write your own solution and not work in a pair on this program.

The Game of Life is a simulation originally conceived by the British mathematician J. H. Conway in 1970 and popularized by Martin Gardner in his Scientific American column. The game models the life cycle of bacteria using a two-dimensional grid of cells. Given an initial pattern, the game simulates the birth and death of future generations of cells using a set of simple rules. In this assignment you will implement a simplified version of Conway's simulation and a basic user interface for watching the bacteria grow over time.

The starter code for this project is available as a ZIP archive:



Starter Code

Due Date: Life is due July 5th at 12:00pm.

Submit: You can submit multiple times. We will grade your latest submission.

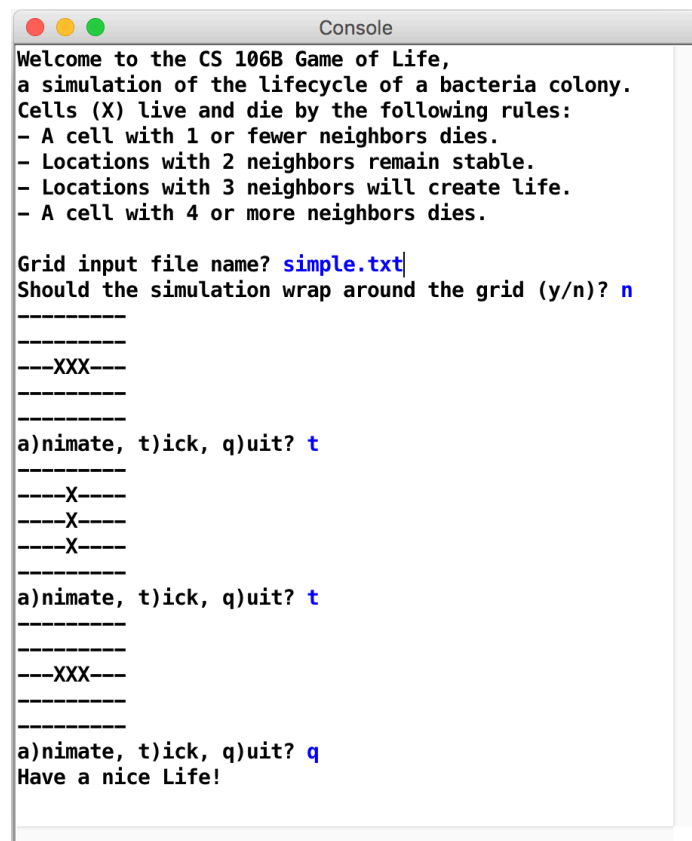
More about Life:
Wikipedia
Life and space-time.

Overview

Your Game of Life program should begin by prompting the user for a file name and using that file's contents to set the initial state of your bacterial colony grid. Then, it should ask if the simulation should wrap around the grid (see below for the details of wrapping). Then the program will allow the user to advance the colony through generations of growth. The user can type t to "tick" forward the bacteria simulation by one generation, or a to begin an animation loop that ticks forward the simulation by several generations, once every 50 milliseconds; or q to quit. Your menu should be case-insensitive; for example, an uppercase or lowercase A, T, or Q should work (hint: you can use the `toLowerCase()` function to convert a string to lowercase).

Simple Example:

Here is an example log of the interaction between your program and the user (with console input in blue).



```

Welcome to the CS 106B Game of Life,
a simulation of the lifecycle of a bacteria colony.
Cells (X) live and die by the following rules:
- A cell with 1 or fewer neighbors dies.
- Locations with 2 neighbors remain stable.
- Locations with 3 neighbors will create life.
- A cell with 4 or more neighbors dies.

Grid input file name? simple.txt
Should the simulation wrap around the grid (y/n)? n

-----
---XXX---
-----

a)nimate, t)ick, q)uit? t
-----
---X---
---X---
---X---
-----

a)nimate, t)ick, q)uit? t
-----
---XXX---
-----

a)nimate, t)ick, q)uit? q
Have a nice Life!
  
```

Many Examples:

Of course there are many other interactions you could have. Here is a bunch more examples. They show what to do in many cases, for example bad input filenames.



Example Run #1



Example Run #2



Example Run #3



Example Run #4



Example Run #5



Example Run #6



Example Run #7



Example Run #8



Example Run #9
(tick wrap)

When ready, compare your programs to these "Example Runs" using the Example Diff Tool. We want your output to match ours exactly:



Example Diff Tool

Files

You will turn in only the following files:

1. **life.cpp**, the C++ code for the Game of Life simulation
2. **mycolony.txt**, your own unique Game of Life input file representing a bacterial colony's starting state

The ZIP archive contains other files and libraries; you should not modify these. When grading/testing your code, we will run your `life.cpp` with our own original versions of the support files, so your code must work with them.

Game of Life Simulation Rules

Each grid location is either empty or occupied by a single living cell (X). A location's neighbors are any cells in the surrounding eight adjacent locations. In the example at right, the shaded middle location has three neighbors containing living cells. A square that is on the border of the grid has fewer than eight neighbors in the non-wrapping version of the simulation (see below for the wrapping version). For example, the top-right X square in the example at right has only three neighboring squares, and only one of them contains a living cell (the shaded square), so it has one living neighbor.

		X
X	X	
X		

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 there, it dies.
2. A location with two neighbors is stable. If it had 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.
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.

The births and deaths that transform one generation to the next all take effect simultaneously. When you are computing a new generation, new births/deaths in that generation don't impact other cells in that generation. Any changes (births or deaths) in a given generation k start to have effect on other neighboring cells in generation $k+1$.

Check your understanding of the game rules by looking at the following example. The two patterns should alternate forever:

		X		
		X		
		X		

	X	X	X	

Here is a second example. The pattern at right does not change on each iteration, because each cell has exactly three living neighbors. This is called a "stable" pattern or a "still life".

	X	X		
	X	X		

Input Grid Data Files:

The grid of bacteria in your program gets its initial state from one of a set of provided input text files, which follow a particular format. When your program reads the grid file, you should re-prompt the user if the file specified does not exist. If it does exist, you may assume that all of its contents are valid. You do not need to write any code to handle a misformatted file. The behavior of your program in such a case is not defined in this spec; it can crash, it can terminate, etc. You may also assume that the input file name typed by the user does not contain any spaces.

In each input file, the first two lines will contain integers *r* and *c* representing the number of rows and columns in the grid, respectively. The next lines of the file will contain the grid itself, a set of characters of size *r* x *c* with a line break (`\n`) after each row. Each grid character will be either a '-' (minus sign) for an empty dead cell, or an 'X' (uppercase X) for a living cell. The input file might contain additional lines of information after the grid lines, such as comments by its author or even junk/garbage data; any such content should be ignored by your program.

The input files will exist in the same working directory as your program. For example, the following text might be the contents of a file `simple.txt`, a 5x9 grid with 3 initially live cells:

```
5                ← number of rows tall
9                ← number of columns wide
-----
-----          ← - is a dead cell
---XXX---        ← X is a living cell
-----
-----
```

Implementation Details

Grid: The grid of bacterial cells could be stored in a 2-dimensional array, but arrays in C++ lack some features and are generally difficult for new students to use. They do not know their own length, they cause strange bugs if you try to index out of the bounds of the array, and they require understanding C++ topics such as pointers and memory allocation. So instead of using an array to represent your grid, you will use an object of the Grid class, which is part of the provided Stanford C++ library.

A Grid object offers a cleaner abstraction of a 2-dimensional data set, with several useful methods and features. See the course lecture examples and/or section 5.1 of the Programming Abstractions in C++ textbook for a list of members of the Grid class (e.g., the **grid.inBounds()** function will come in very handy). You can also use the = assignment operator to copy the state of one Grid object to another.

Your main function should create your Grid and pass it to the other functions. Since it is expensive to make copies of a Grid, if your code passes a Grid object as a parameter from one function to another, it should always do so by reference (Grid&, not Grid). See the lecture notes for examples. Since you don't know the size of the grid until you read the input file, you can call **resize** on the Grid object once you know the proper size.

I/O: Your program has a console-based user interface, although there is a relatively easy extension that includes a GUI. You can produce console output using **cout** and but you should be careful when using **cin** for input: you should use the Stanford C++ library's console-related functions such as **getline** (uppercase L) to read from the console. See the Stanford C++ library documentation on the class web site.

You will also write code for reading input files. Read a file using an **ifstream** object, along with functions such as **getline** (lowercase L) to read lines from the file. Here are some useful **ifstream**-related functions from **filelib** and **strlib**:

Command	Description
openFile(ifstream & stream, string filename);	Opens the file with the given filename/path and stores it into the given ifstream output parameter.
getline(ifstream & stream, string & line);	Reads a line from the given stream and stores it into the given string variable by reference.
fileExists(string & fileName);	Checks if a file with the corresponding fileName exists. Returns a bool.
stringToInteger(str)	Returns an int value equivalent to the given string; for example, "42" → 42
integerToString(n)	Returns a string value equivalent to the given integer; for example, 42 → "42"

Make sure to close your input file streams when you are done reading the given file: **stream.close()**.

Checking for valid input: Note that your program needs to check for valid user input in a few places. When the user types the grid input file name, you must ensure that the file exists, and if not, you must re-prompt the user to enter a new file name. If the user is prompted to enter an action such as 't' for tick or 'a' for animate, if the user types anything other than the three predefined commands of a, t, or q (case-insensitively), you should re-prompt the user to enter a new command. If the user is prompted to enter an integer such as the number of frames of animation for the 'a' command, your code should re-prompt the user if they type a non-integer token of input. (If they do type an integer, you may assume that it is a valid value greater than 0; you don't need to explicitly test or check its value.) See the expected output files on the class web site for examples of this output. There are several functions from the Stanford C++ library that can help you with this functionality, such as **fileExists** and **getInteger**.

Animation: When the user selects the animation option, the console output should look like the following:

```
a)nimate, t)ick, q)uit? a
How many frames? xyz
Illegal integer format. Try again.
How many frames? 5
(five new generations are shown, with screen clear and 50ms pause before
each)
```

It is hard to show an example of animation output in this handout because the output does not translate well to a plain-paper format. The screen is supposed to clear between each generation of cells, leading to what looks like a smooth animation effect. Run the Sample Solution from the class web site to see how animation should work.

To help you perform animation, use the following global functions from the Stanford C++ library:

Command	Description
<code>pause (ms) ;</code>	Causes the program to halt execution for the given number of milliseconds
<code>clearConsole () ;</code>	Erases all currently visible text from the output console (call this between frames)

Wrapping:

The non-wrapping version of the assignment treats the edges of the grid as the end of the game world. Cells on the border do not always have eight neighbors. In the wrapping version, all cells will have eight neighbors, as follows: the right-most squares are considered to be "neighbors" of the left-most, and the top-most are considered to be "neighbors" of the bottom-most. In order to provide wrapping functionality, modify your game logic so that these rules are followed. This will allow moving patterns such as "gliders" to wrap around indefinitely.

The logic for this is not too difficult, and you may want to use the remainder operator (%) to perform part of this task. The remainder function works as follows:

(a % b) returns the remainder of a / b

For positive values, the operator returns the value of the remainder, which "wraps" around to the value. E.g., `6 % 5` is `1`, which would be correctly wrapped on a grid from 0-4 (which has 5 values).

For negative values, the remainder function does not wrap in C++, but wrapping can be accomplished by simply adding the number of rows or columns in the grid to the negative value. In fact, to wrap properly in all cases, simply add the number of rows or columns to the location, and then apply the remainder operator.

For example, let's say you were checking the bottom left corner of a 5x5 grid (with indexes 0-4 for both the rows and columns), at location (4,0), as shown in the diagram below.

	0	1	2	3	4
0					
1					
2					
3					
4	X				

The blue squares show the neighbors with wrapping, and going clockwise from the top-left corner of the neighbors, would be at locations (3,-1), (3,0), (3,1), (4,-1), (4,1), (5,-1), (5,0), and (5,1). But, both the negative values and the values above 4 are out of bounds. If we apply the remainder operator as detailed above, we will get a proper wrapping of the values. If we add the corresponding number of rows or columns (5 in this case, for both), and then apply the remainder operator with the same value to each of the coordinate pairs, we will get a proper wrap. Using coordinate (5, -1) as an example, this would become:

$$((5 + 5) \% 5, (-1 + 5) \% 5) = (10 \% 5, 4 \% 5) = (0, 4)$$

and that coordinate is properly wrapped to the top right corner.

Creative Aspect (mycolony.txt):

Along with your program, submit a file mycolony.txt that contains a valid initial colony that can be used as input. This can be anything you want, as long as it is in the input grid file format described in this document, and should be your own work (not just a copy of an existing colony input file). This is worth a small part of your grade.

Development Strategy and Hints:

Development strategy: It is tempting to try to write your entire program and then try to compile and run it; we do not recommend that strategy. Instead, you should develop your program incrementally: Write a small piece of functionality, then test/debug it until it works, then move on to another small piece. This way you are always making small consistent improvements to a base of working code. Here is a possible list of steps to develop a solution:

1. Intro: Get your basic project running, and write code to print the introductory welcome message.
2. File input: Write code to prompt for a filename, and open and print that file's lines to the console. Once this works, try reading the individual grid cells and turning them into a Grid object. Print the Grid's state on the console using toString just to see if it has the right data in it. Use a simple test case, e.g. simple.txt.
3. Grid display: Write code to print the current state of the grid, without modifying that state.
4. Updating to next generation: Write code to advance the grid from one generation to the next. This is one of the hardest parts of the assignment, so you will probably need lots of testing and debugging before it works perfectly. Insert cout statements to print important values as you go through your loops and code. Try printing out what indexes your code is examining, along with your count of how many neighbors each cell has, to make sure you are counting them properly.
5. Overall menu and animation: Implement the program's main menu and the animation feature. If all of the preceding steps are finished and work properly, this should not be as difficult to get working.

Updating from one generation to the next: When you are trying to advance the bacteria from one generation to the next, you cannot do this "in place" by modifying your grid as you loop over it. Doing so will change the cells and their neighbors and break the neighbor counts for nearby cells. So you will need to create a temporary second grid. Your existing grid represents

the current generation of bacteria, and you can create a second temporary second grid that allows you to compute and store the next generation without changing the current one. Once you have filled the second grid with the next generation's cell information, you can copy its contents back into the original grid and discard the temporary copy. Copying one Grid to another is easy; just assign one to the other using the = assignment operator, which makes a copy of its contents.

Output: We want your output to match ours exactly. This includes identical spacing, such as the extra spaces after the phrase, "Grid input file name? " Some students lose points for minor output formatting errors. Please run the web Output Comparison Tool on several test cases to make sure it matches without any differences.

Hints: Here are some other miscellaneous tips that may help you:

1. You can convert between strings and integers using functions `stringToInteger` and `integerToString`.
2. You can re-prompt for a file name using the library function `promptForFileName`.
3. A common bug when counting neighbors of a given cell is to count that cell itself. Don't do that.
4. If your editor is unable to compile your program, complaining about "cannot open output file ...: Permission denied", you need to close/terminate all windows from previous runs of the program.

Possible Extra Features:

Thats all! You are done. Consider adding extra features.

Though your solution to this assignment must match all of the specifications mentioned previously, it is allowed and encouraged for you to add extra features to your program if you'd like to go beyond the basic assignment. Here are some example ideas for extra features that you could add to your program.

1. **Random world generation:** Add special logic so that when the user is prompted for an input file name, if they type the word "random", instead of loading your input from a file, your program will randomly generate a game world of a given random size. That is, your code will randomly pick a grid width and height (of at least 1), and then randomly decide whether to make each grid cell initially living or dead. This way you can generate infinite possibilities of new game worlds each time you run the program.
2. **Graphical display:** The default version of the assignment displays its output as text. But we also provide a file called `life-gui.cpp` that contains an implementation of a graphical display of the Life game world. For this feature, insert code into your program that uses `life-gui`. Here are the functions you can call:

Command	Description
<code>LifeGUI name;</code>	creates a new graphical user interface (GUI) window
<code>gui.resize(nRows, nCols);</code>	Sets the GUI to use the given number of rows/cols in its grid
<code>gui.drawCell(r, c, alive);</code>	Tells the GUI whether the cell in the given row/column index is alive (true) or dead (false); living cells are drawn in color on the GUI. If you call <code>drawCell</code> on a previously-living cell and pass false for alive, the GUI slowly "fades out" that cell with each generation

3. **GUI enhancements:** Do you want to add a feature to the provided graphical interface? If so, tweak the provided GUI files and submit them with your turnin. We haven't taught about GUI programming, but if you want to look at the provided files and learn how they work, we encourage you to do so.
4. **New Rules:** Add new rules to the game. For instance, you might add rules that apply to cells two away from a target

cell, or you might have random mutations occur in the simulation. If you add new rules, be sure to write a lengthy comment about what your rules accomplish and how that affects the game.

5. **Other:** If you have your own creative idea for an extra feature, ask your SL and/or the instructor about it.

Indicating that you have done extra features:

If you complete any extra features, then in the comment heading on the top of your program, please list all extra features that you worked on and where in the code they can be found (what functions, lines, etc. so that the grader can look at their code easily).

Submitting a program with extra features:

Since we use automated testing for part of our grading process, it is important that you submit a program that conforms to the preceding spec, even if you want to do extra features. If your feature(s) cause your program to change the output that it produces in such a way that it no longer matches the expected sample output test cases provided, you should submit two versions of your program file: a first one named `life.cpp` without any extra features added (or with all necessary features disabled or commented out), and a second one named `life-extra.cpp` with the extra features enabled. Please distinguish them in by explaining which is which in the comment header. Our turnin system saves every submission you make, so if you make multiple submissions we will be able to view all of them; your previously submitted files will not be lost or overwritten.

© Stanford 2017 | Web design by Chris Gregg. CS106B has been developed over decades by many talented teachers.