

Game of Life

DUE: JULY 5, 2018, 5PM

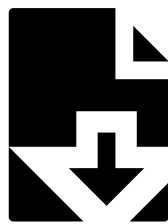
Assignment by Marty Stepp and Victoria Kirst, based on previous version by Julie Zelenski, with other revisions by Jerry Cain, Keith Schwarz, Cynthia Lee, Ashley Taylor, etc.

- [Links](#)
- [Description](#)
- [Logs](#)
- [Overview](#)
- [Input Files](#)
- [Implementation](#)
- [Style](#)
- [Creative Aspect](#)
- [Debugging Aspect](#)
- [FAQ](#)
- [Extras](#)

This problem is about C++ functions, strings, reading files, using libraries, and decomposing a large problem.

This is a pair assignment. You may work in a pair or alone. Find a partner in section or use the course message board. If you work as a pair, comment both members' names atop every code file. Only one of you should submit the program; do not turn in two copies. Submit using the Paperless system linked on the class web site.

Links:



Starter Code

We provide a ZIP archive with a starter project that you should download and open with Qt Creator. You will edit and turn in only the following files. The ZIP contains other files/libraries; do not modify these. Your code must work with the other files unmodified.

- **life.cpp**, the C++ code for your solution
- **mycolony.txt**, your own unique Game of Life input file representing a bacterial colony's starting state
- **debugging.txt**, a file detailing a bug you encountered in this assignment and how you approached debugging it

Program Description:

The Game of Life (Wikipedia) is a simulation by British mathematician J. H. Conway in 1970. 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.

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 will allow the user to advance the colony through generations of growth. The user can type **t** (or just press Enter) 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 **100 milliseconds**; or **q** to quit. Your menu should be case-insensitive; for example, an uppercase or lowercase A, T, or Q should work.

When the user quits, you should prompt the user Y/N to load another file or exit the program. If the user types any string that begins with the letter Y, case-insensitive, you should prompt to load another file. If the user types any string that begins with the letter N, case-insensitive, you should end the program and print "Have a nice Life!". If the user types a string that does not begin with either Y or N, you should re-prompt them with the message, "Please type a word that starts with 'Y' or 'N'.". You may want to use the provided library function **getYesOrNo** from "**simpio.h**" (documentation) to help you with this.

Example Logs of Execution:






Here is an example log of interaction from your program (with user input bolded). Your output must match this format exactly to earn full credit. (We are very picky.)


```
Welcome to the CS 106B/X Game of Life!
This program simulates the lifecycle of a bacterial 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? foobar.txt
Unable to open that file. Try again.
Grid input file name? unknown.doc
Unable to open that file. Try again.
Grid input file name? this file does not exist LOLLOL
Unable to open that file. Try again.
Grid input file name? simple.txt
-----
-----
---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
Load another file? (y/n) maybe
Please type a word that starts with 'Y' or 'N'.
Load another file? (y/n) n
Have a nice Life!
```

Example log of execution

Here are some additional expected output files to compare. Your program's Console window has a File → Compare Output feature for checking your output, as well as a Load Input Script feature that can auto-type the user console input for you. Use them!

-  test #1 (simple/seeds)
-  test #2 (snowflake)
-  test #3 (glider)
-  test #4 (spiral)
-  test #5 (glider)

-  test #6 (diehard)

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 below, the shaded location has three neighbors containing living cells.

					X
			X	X	
			X		
X					

Example Game of Life grid

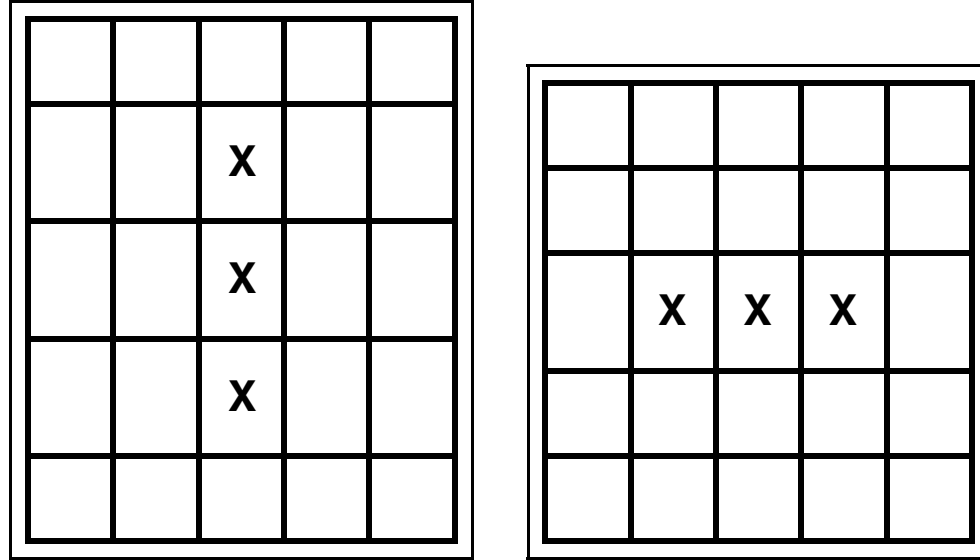
The Game of Life world is *toroidal*; this means that it **wraps around** in both dimensions. In other words, the top-most and bottom-most rows are considered to neighbor each other, as do the left-most and right-most columns. For example, the top-right X square in the example above has two neighbors that contain living cells: the cell down/left from it, and the cell up/right from it (which is the grid's bottom-left cell). To make sure that the definition of "neighbor" is well-defined, you may assume that the board's dimensions are at least 3x3.

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

- A location that has zero or one neighbors will be **empty** in the next generation. If a cell was there, it dies.
- 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.
- 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.
- 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.

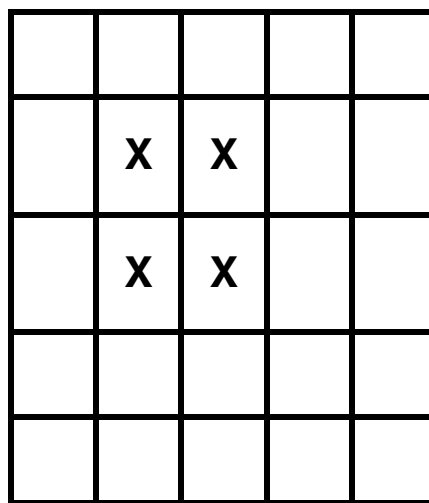
The births and deaths that transform one generation to the next all take effect simultaneously. When you are computing a new generation, new births/deaths in that generation don't impact other cells in that generation. Any changes (births or deaths) in a given generation k start to have effect on other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

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



Alternating pattern

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



Stable "still life" pattern

Input 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 and follow the proper format. 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. Note that the input file name might contain 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 $r \times c$ 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. You may assume that the world's size is at least 3x3.

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:

1	5
2	9
3	-----
4	-----
5	---XXX---
6	-----
7	-----

Contents of input file ***simple.txt***

Implementation Details:

Your program will be graded on functionality and style. To achieve a high functionality score, test your program thoroughly with the provided input files and your own test inputs. Here are some specific details about our expectations for your implementation:

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 must use an object of the **Grid** class (documentation), 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.

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. You can also use the **=** assignment operator to copy the state of one **Grid** object to another. See the course lecture examples, section 5.1 of the *Programming Abstractions in C++* textbook, and/or the online **Grid** documentation for more information about the Grid class.

Checking for valid input: 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**, **q** (case-insensitively), you should re-prompt the user to enter a new command. (Note that just pressing Enter at that prompt should be treated the same as pressing **t** and should 'tick' the colony forward to its next generation.) 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 type a negative integer, you should simply not perform the animation and return to the main menu.) There are several functions from the Stanford C++ library that can help you with this functionality, such as **fileExists** and **getInteger**. These functions come from Stanford library files "**filelib.h**" (documentation) and "**simpio.h**" (documentation).

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 100ms 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 Demo 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 "console.h" (documentation):

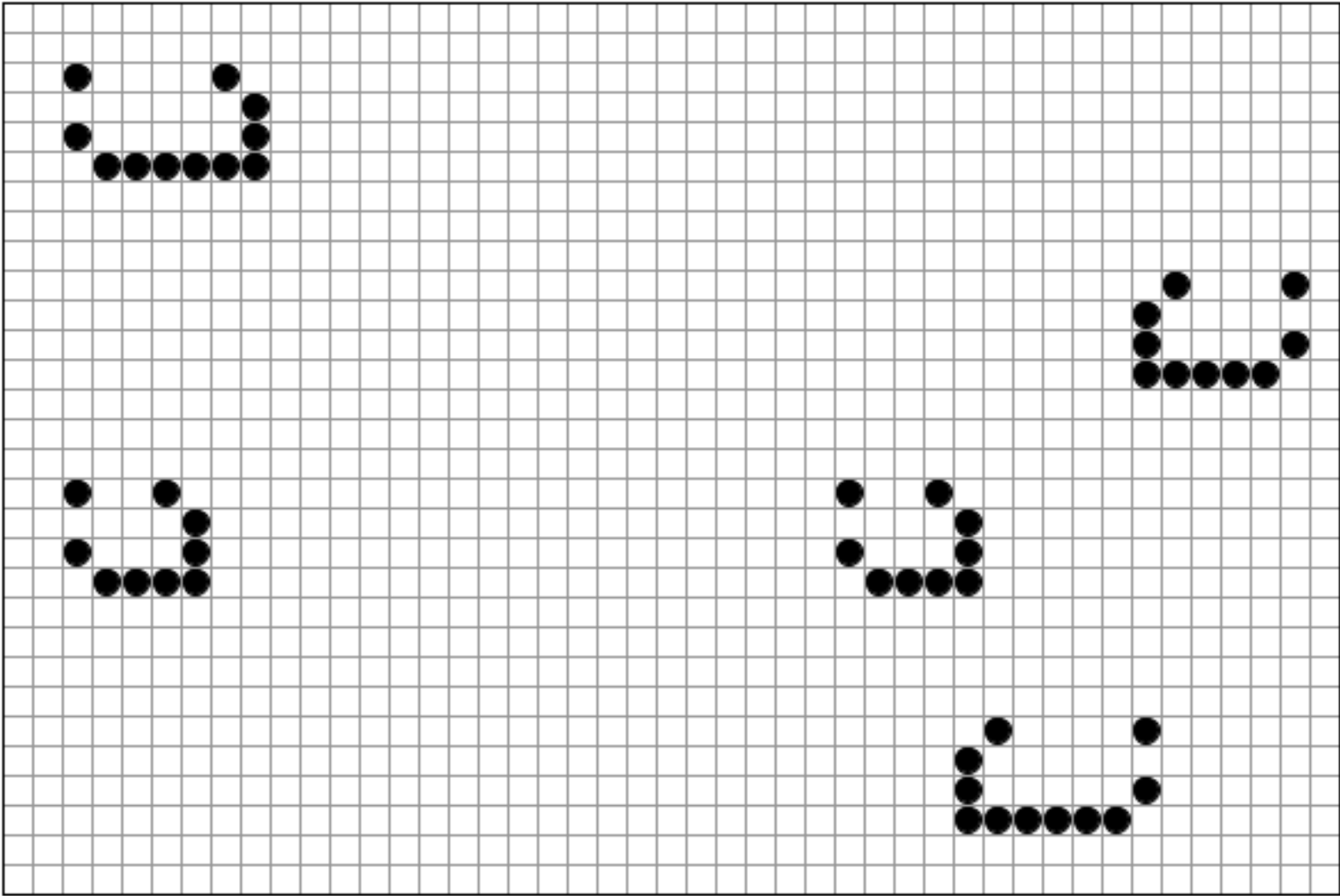
clearConsole();	Erases all currently visible text from the output console. (<i>call this between frames</i>)
pause(ms);	Causes the program to halt execution for the given number of milliseconds.

I/O: Your program has a console-based user interface. You should pop up the Stanford graphical console by including "**console.h**" (documentation) in your program. Produce console output using **cout** and read console input using **cin**. You may use the Stanford C++ library's console-related functions such as **getLine** (uppercase L) to read from the console. See **simpio.h** (documentation) for more details.

You will also write code for reading input files. Read a file using an **ifstream** object (documentation), along with functions such as **getline** (lowercase L) to read lines from the file. If a given line contains an integer or numeric token, call **stringToInteger** to convert it. See **strlib.h** (documentation) for more details. Make sure to close your input file streams when done reading.

Strings: A non-trivial part of the program involves string manipulation. You may want to look up members of the C++ **string** class (documentation) such as **find**, **length**, **substr**, and so on.

Graphical User Interface (GUI):



As a required part of this assignment, you must also add code to use an instructor-provided graphical user interface (GUI) with your program. The GUI does not replace the console UI; it can't be clicked on to play the game, for example. It just shows a display of the current game state. The GUI should first appear when the program's console pops up, and it should draw the colony state for each generation. To use the GUI, include "**lifegui.h**" in your code, then call the functions below:

Function Name	Description
LifeGUI::clear();	Erases any filled circles for cells in the grid.
LifeGUI::fillCell(row, col);	Draws a black circle for the cell at the specific row and column. The cell will not immediately appear on the screen; the client must also call repaint to see any newly drawn cells. The idea is that on each generation of your simulation, you should call fillCell on all living cell locations, and then call repaint once to see all of the changes. If the location given is not in bounds, an error is thrown.
LifeGUI::initialize();	Sets up the state of the GUI and pops up the GUI window on the screen.

	This needs to be called only once by the client.
LifeGUI::repaint();	Redraws the GUI window, showing any newly drawn cells that have been drawn using fillCell since the window was last repainted.
LifeGUI::resize(rows, cols);	Informs the GUI about the given number of total rows and columns in the simulation. Calling this will erase the graphics window completely, draw a black border around the simulation rectangle which is centered in the window, and draw light gray grid lines around each cell. This function can be used at the beginning of a simulation or between generations to clear the window before drawing the next generation.
LifeGUI::shutdown();	Closes the GUI window and shuts down the GUI.

Style Details:

To achieve a high style score, submit a program with high code quality that conforms to the guidelines presented in our course. There are many general C++ coding styles that you should follow, such as naming, indentation, commenting, avoiding redundancy, etc. While there are many valid programming styles in various contexts, the most important overall stylistic trait a programmer can have is the ability to be given a set of style guidelines and follow them rigorously and consistently. In the context of this course, our style guides and constraints are the laws of the land and are not open for debate.

Before getting started, you should read our **Style Guide** for information about expected coding style. You are expected to follow the Style Guide on all homework code. The following are some points of emphasis and style constraints specific to this problem:

C++ idioms: For full credit, you must use C++ facilities (**cout**, **ifstream**, **string**) instead of C equivalents (**printf**, **fopen**, **char***).

Procedural decomposition: Your **main** function should represent a concise summary of the overall program. It is okay for **main** to contain some code, such as calls to other functions or brief console output statements to **cout**. But **main** should not perform too large a share of the overall work itself directly, such as reading the lines of the input file or prompting the user to replace placeholders. Instead, it should make calls to other functions to help it achieve the overall goal. You should declare **function prototypes** (each function's header followed by a semicolon) near the top of your file for all functions besides **main**, regardless of whether this is necessary for the program to compile.

The **main** function should not perform too large a share of the overall work itself directly, such as reading the lines of the input file or performing the calculations to update the grid from one generation to the next.

Each function should perform a single clear and coherent task. No one function should do too large a share of the overall work. As a rough estimate, a function whose body (excluding the header and closing brace) has more than 30 lines is likely too large. You should avoid "chaining" long sequences of function

calls together without coming back to **main**, as described in the Procedural Design Heuristics handout on the course web site. Your functions should also be used to help you avoid **redundant code**. If you are performing identical or very similar commands repeatedly, factor out the common code and logic into a helper function, or otherwise remove the redundancy.

Parameters, Returns, Values, References: Since your program will have several functions and those functions will want to share information, you will need to appropriately pass parameters and/or return values between the functions. Each function's parameters and return values should be well chosen. Do not declare unnecessary parameters that are not needed by your function. A particular point of emphasis on this assignment is that you should demonstrate that you understand when it is proper to **pass by reference**, and when it is proper for a parameter to be declared **const**. You should also demonstrate that you understand when it is better to return a result and when it is better to store a result into an 'output' reference parameter.

Variables and types: Use descriptive variable and function names. Use appropriate data types for each variable and parameter; for example, do not use a **double** if the variable is intended to hold an integer, and do not use an **int** if the variable is storing a **true/false** state that would be better suited to a **bool**. When manipulating strings, favor talking to **string** objects over individual **char** values when possible, and use the **string** object's built-in methods as opposed to rewriting similar behavior yourself. Do not declare any global variables or **static** variables; every variable in your program must be declared inside one of your functions and must exist in only that scope. No single variable's scope should extend beyond a single invocation of a single function.

Collections: On this problem, you are allowed to use a single **Grid** object to represent the cells of the game. If you are updating your simulation from one generation to the next, you may create a temporary backup copy of your grid as needed, but that temporary copy should not live on beyond any one single function of your code. Do not use any other collections/arrays/containers/etc. If you use your simulation grid in many functions of your program, you should pass it between them as a parameter. You should not declare your **Grid** (nor any other variables) globally at the top of your code outside of your program's functions; global variables are forbidden in this course and are considered very poor style.

Commenting: Your code should have adequate commenting. The top of your file should have a descriptive comment header with your name, a description of the assignment, and a citation of all sources you used to help you write your program. Each function should have a comment header describing that function's behavior, any parameters it accepts and any values it returns, and any assumptions the function makes about how it will be used. For larger functions, you should also place a brief inline comment on any complex sections of code to explain what the code is doing. See the programs written in lecture or the Course Style Guide for examples of proper commenting.

Creative Aspect, **mycolony.txt**:

Along with your code, 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 a non-trivial file in the input grid file format described in this document, and is your own work (not just a copy of an instructor-provided colony

input file). This is worth a small part of your grade.

Debugging Aspect, **debugging.txt**:

Along with your code and **mycolony.txt**, you should also submit **debugging.txt**. In this file, you should write 1-2 paragraphs describing a bug you encountered and how you debugged it. If you got help in LaIR or office hours, explain how the help unblocked you. Please read the debugging handout for ideas on the sorts of questions you should be answering.

Frequently Asked Questions (FAQ):

For each assignment problem, we receive various frequent student questions. The answers to some of those questions can be found by clicking the link below.

NOTE: On this first assignment, many students have issues with setting up their Qt Creator software. If your problem or issue is more about that software than it is about HW1 specifically, you should also check out our Qt Creator troubleshooting page for possible solutions.

Q: You said I need to use Qt Creator to write my homework, but I prefer a different editor (Visual Studio, Xcode, Eclipse, Netbeans, vim, emacs, etc.). Can I use my favorite editor instead of Qt Creator?

A: Probably not. We don't support this. We are giving you out a starter project in Qt Creator's format, so if you were trying to use a different editor, you'd have to disassemble that starter project and put it back together in your editor's format, which might be difficult. Also, if you have any problems getting things to work in your editor, we will not be willing to help you fix them.

Q: I'm having trouble getting Qt Creator to work! Help!

A: Please make sure you followed the instructions in our Qt Creator page. If you are still having trouble, please see our Tricky C++ Issues page for some possible solutions.

Q: When I try to compile my program I see an error message, "cannot open output file ...\\Life.exe: Permission denied". What does it mean? How do I fix it?

A: It means that your Life executable is still running from the last time you ran/tested the program a moment ago. Make sure to shut down any previously running instances of your program. Click Qt Creator's "3 Application Output" tab and click any of the red square "stop" sign buttons you see. You might also need to open your operating system's Task Manager to stop them all.

Q: How do I construct a Grid? I tried saying new Grid but it didn't compile.

A: Constructing an object in C++ has a different syntax than in Java. Instead of saying **new**, you just write a statement such as:

```
Grid<type> name;
```

For example, if you wanted to declare a grid of **doubles** and call the variable **myGrid**, you would write:

```
Grid<double> myGrid;
```

Q: Do I have to use a Grid on the assignment? Can I just use an array?

A: Yes, you must use a **Grid**. Using an array will not receive full credit.

Q: Can I use one of the STL containers from the C++ standard library, instead of a Grid?

A: No; you should use the **Grid**.

Q: I already know a lot of C/C++ from my previous programming experience. Can I use advanced features, such as pointers, on this assignment?

A: No; you should limit yourself to using the material that was taught in class so far.

Q: Can I add any other files to the program, other than `life.cpp`? Can I add some classes to the program?

A: No; you should limit yourself to **`life.cpp`** and functions defined in that file.

Q: I am having trouble reading the grid from the file, character-by-character. How do I detect the line endings and how do I know how many characters to read?

A: We don't recommend reading the file character-by-character. Instead, repeatedly call **`getline`** on the file and then process that entire line. If the line contains an integer or numeric token, call **`stringToInteger`** to convert it. See **`strlib.h`** (documentation) for more details.

Q: How do I count the neighbors of a cell?

A: Examine the eight squares around it. Be careful not to go out of the bounds of the grid, if the square you're examining is on an edge of the grid. Also be careful not to count the current cell itself as a neighbor. We suggest inserting temporary debugging statements to **`cout`** to show you what your program is examining along the way. For example, as you look at each square, print the row/column numbers, along with how many neighbors you count for that cell or which cells you think have living neighbors in them.

Q: I am having trouble reading the input files. It always crashes with a "file not found" error. What

is wrong?

A: Remember that the input files are in the same directory as your program, so if the user types "**foo.txt**" you need to open "**foo.txt**" in your code with no path or directory name in front of the file name. The grid input files should be found in the starter ZIP file in the **res** subdirectory of the overall project. That's also where you can put your own custom **mycolony.txt** file later when you make it.

Q: What does this error mean?

```
error: invalid initialization of non-const reference of type 'Grid<foo>&' from an rvalue of type 'Grid<foo> (*)()'
```

A: This can happen when you declare your **Grid** in the wrong way:

```
Grid<type> name();    // no
Grid<type> name;      // yes
```

For example, if you wanted to declare a grid of **doubles** and call the variable **myGrid**, you would write:

```
Grid<double> myGrid;
```

Q: What does this error mean?

```
error: symbol(s) not found for architecture x86_64
```

A: This can happen when you are trying to call a function that you have not declared, or call a function with the wrong parameters. Remember that you need to declare function prototypes (the function's name and parameters followed by a semicolon) at the top of your program for **main** to be able to see them.

Try clicking the "4 Compiler Output" tab button near the bottom of Qt Creator and reading the detailed output about exactly what symbol was not defined.

Q: Why can't I pass an ifstream file input stream as a parameter?

A: You must pass it by reference.

Q: My output seems to be "jerky"; sometimes things won't print out when I think they should, and then later a bunch of output prints all at once. Why would this be happening?

A: It might have to do with `\n` vs `endl` and something called output buffering. When you print to **cout**, it doesn't always immediately send the output to the console. This is because sending output to the console is somewhat slow/expensive, so for example if you print a grid character-by-character, it is slow to individually print every single character. So **cout** has an internal buffer in which it stores the

characters you tell it to print. Periodically, **cout** decides to flush out this buffer and print everything you told it to print.

One way to force **cout** to flush out all of its buffered output is to print **endl**. Note that it does NOT flush the output if you print **"\n"** instead of **endl**. So most of the time, if you're seeing weirdly delayed output, it's because you are doing **cout << something << "\n";** instead of the preferred, **cout << something << endl;**

Another way to force **cout** to flush out any buffered output is to say, **cout.flush();**

Q: How do I do animation?

A: It's the same as doing a single tick to the next generation, but you do it repeatedly. Between ticks, pause the simulation (**pause**) and also clear the console text (**clearConsole**) so that the animation looks smooth.

Q: The expected output files don't appear to "clear" the console between frames of animation.

Why not? Which should I follow, the spec or the expected output files?

A: You should follow the spec and clear the console between frames of animation. The expected output files can't really show the console clearing because there's no way to show that in a flat text file. You can still match their output by copy-pasting from the bottom "3 Application Output" tab of Qt Creator.

Q: Will my solution get full credit? Is it written in the style you want? Will I get marked off for this code?

A: In general we cannot answer these kinds of questions. We call this "pre-grading." The section leader/TA/instructor can't look over your entire program for mistakes or tell you exactly what things you will get marked off for; we don't have the resources to provide such a service, and even if we did, we want you to learn how to gain these intuitions on your own. We'll grade you on the guidelines in the homework document, and we can help you with specific issues and questions about your code, but we cannot pre-evaluate your entire program for you or give you advance warning about every possible mistake or violation.

Q: Is my mycolony.txt file okay? Will it get full credit?

A: Our grading is pretty lenient on a creative part aspect like this. If it meets the criteria in the assignment spec, it should get full credit, even if it is not particularly creative or exciting.

Possible 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. Note that our motivation for allowing extra features is to encourage your creativity, not inflate your grade; so if any points are given for your extra features, they will be minimal. The purpose is to explore the assignment domain further, not to disrupt the class grading curve. Make sure to see the notes below about how to separate your extra code and how to turn it in properly.

Here are some example ideas for extra features that you could add to your program:

- **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.
- **Cells that "age":** In the basic simulation, cells are either alive or dead with nothing in between. But as an extension, perhaps you could make cells that remember how many generations they have been alive and act differently (different character, different color, etc.) on screen depending on their age.
- **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.

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.