Stanford University, CS 106B, Autumn 2013
Homework Assignment 1: The Game of Life
due Monday, October 7, 2013, 2:00pm

Thanks to Julie Zelenski for this assignment idea, with later revisions by Jerry Cain.

The purpose of this assignment is to gain familiarity with basic C++ features such as functions, strings, and I/O streams, as well as interacting with provided libraries. You will also practice decomposing a large problem into smaller tasks and representing those tasks as well-designed functions.

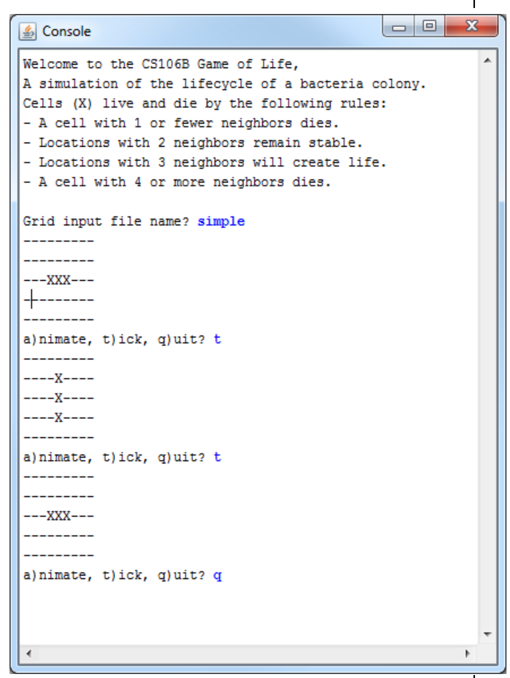
The **Game of Life** is a simulation originally conceived by the British mathematician J. H. Conway in 1970 and popularized by Martin Gardner in his Scientific American column. The game is a simulation that models the life cycle of bacteria using a two-dimensional grid of cells. Given an initial pattern, 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 text 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` 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 the program. All commands are in lowercase; any other key pressed can be ignored.

Here is an example **log of interaction** between your program and the user (with console input underlined):

```
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
---------
---------
---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!
```



Files:

We will provide you with a **ZIP archive** that contains a starter version of your project. You should download this archive from the class web site and write the rest of the code. You will turn in only the following files:

- **life.cpp**, the C++ code Game of Life simulation
- **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:

		X
X	X	
X		

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. 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.

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 at right. The two patterns at right 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 may assume that the file exists and its contents are valid. You do not need to write any code to check whether the file exists, nor 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; 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
9
---------
---------
---XXX---
---------
---------
```

← *number of rows tall*
← *number of columns wide*
← *grid of rows and columns*
`-` *is a dead cell,* `X` *is a living cell*

Implementation Details:

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. Here are some of the useful members of the `Grid` class. The course web site has complete documentation.

Member	**Description**
`Grid<T>()` `Grid<T>(rows, columns)`	Constructs a grid of the given size, where each cell stores a value of the given type of data ***T*** ; or an empty grid if no size is passed (must be resized later).
`grid[row][column]`	Individual grid cells can be accessed and modified using `[][]` brackets.
`get(row, column)`	This method returns the contents of an individual grid cell; an alternative to the `[][]` notation.
`inBounds(row, column)`	Returns `true` if the given row/column index is within the bounds of this grid, or `false` if not.
`numCols()`	Returns the number of columns in the grid.
`numRows()`	Returns the number of rows in the grid.
`resize(nRows, nCols)`	Changes the grid's size to be the given dimensions. Any previous grid contents are discarded.
`set(row, column, value)`	This method sets the contents of an individual grid cell to the given value; an alternative to the `[][]` notation.
`toString()`	Returns a printable string representation of the grid.

You can also use the `=` assignment operator to copy the state of one `Grid` object to another. See the course lecture examples and/or section 5.1 of the *Programming Abstractions in C++* textbook for examples of using a `Grid`.

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.

Your program has a **console**-based user interface. You can produce console output using `cout` and read console input using `cin`. If you like, you may 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:

`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.
`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"`

To help you perform animation, use the following global functions:

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

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 file name, 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.

Avoid "Extra row/column" implementation: One tricky part of this assignment is that the edges of the grid have to be handled with care, because you don't want your code to try to access a grid index that is out of bounds. Your algorithm for examining cells and counting neighbors should handle edges and inner cells elegantly and without redundancy as much as possible. Some students try to achieve this by creating an extra layer around the grid; for example, a 5x9 input file will be put into a 7x11 `Grid` object. We do not want you to solve the problem in this way, for several reasons: it avoids some of the challenges of the assignment; it wastes memory; and it introduces other hacks and kludges in your code to adjust the indexes to work properly. Avoid an "extra row/column" solution.

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:

- You can convert between strings and integers using the functions `stringToInteger` and `integerToString`.
- A common bug when counting neighbors of a given cell is to count that cell itself. Don't do that.
- 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.

Implementation and Grading:

Functionality: We will grade your program's behavior and output to give you a **Functionality score**. You can use the course web site's **Output Comparison Tool** to help check your output for various test cases. Your code should compile without any errors or warnings, and should work with the existing support code and input files as given.

Style: We will also grade your code quality to give you a **Style score**. There are many general C++ coding styles that you should follow, such as naming, indentation, commenting, avoiding redundancy, etc. For a list of these, please see the **Style Guide** document posted on the class web site. Follow its guidelines on this and future assignments. In general you should limit yourself to using C++ syntax that was discussed in lecture or in assigned sections of the textbook; for example, do not use advanced C++ features like pointers, arrays, or STL containers on this program.

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 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 performing the calculations to update the grid from one generation to the next. 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.

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.

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`. Use a single `Grid` object, and not any other collections/arrays/containers/etc., to represent the cells of the game. Do not declare any **global variables**; every variable in your program must be declared inside one of your functions and must exist in that scope.

Parameters, values, and 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. You should demonstrate proper usage of parameters and returns; for example, do not declare unnecessary parameters that are not needed by your function. One particular point of style emphasis on this assignment is that you should show a proper understanding of **value and reference semantics** and when it is appropriate to use each one in C++, especially when used with parameters. For example, do not pass a bulky object such as a `Grid` or `ifstream` by value, because this makes an expensive copy of the object. But do not use a reference parameter in a situation where it is not necessary or beneficial to do so (such as a simple `int` that is not being modified by the function).

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 Style Guide for examples of proper commenting.

Honor Code: Please remember to follow the **Honor Code** when working on this assignment. Submit your own work and do not look at others' solutions. Also please do not give out your solution and do not place a solution to this assignment on a public web site or forum. If you need help, please seek out our available resources to help you.

For reference, our solution is around 160 lines long including comments, and it has 5 functions besides `main`, though you don't need to match these numbers to get full credit; they are just here as a ballpark or sanity check.