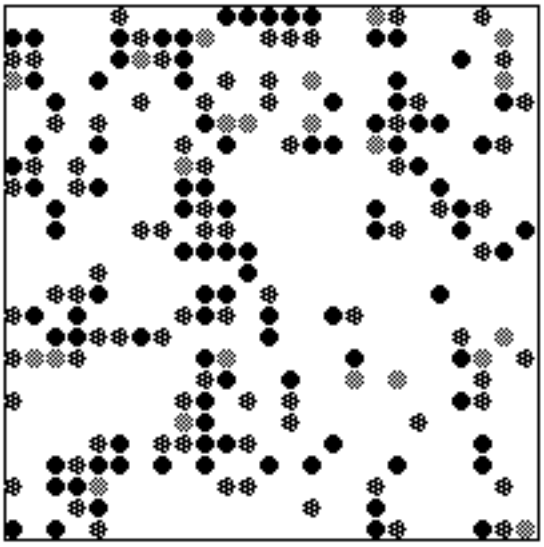
Assignment #2: The Game of Life

Due: Friday, April 14th in class

Your first assignment was a good warm-up, this next assignment provides a bit more of a challenge. It will give you further practice with all the basic C control structures, designing functions, using graphics and random numbers and a bit of strings and file processing. Most importantly the program will exercise your ability to decompose a large problem into manageable bits. This is your first difficult and fairly long program—don't delay getting started.



The problem

In this assignment you will implement a version of the game of Life (no, not the board game). This game was originally conceived by the British mathematician J.H. Conway in 1970 and was 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 of bacteria. It's like a Lava Lamp for mathematicians.

The game is played on a two-dimensional grid. Each grid location is either empty or occupied by a single cell (X). A location's *neighbors* are any cells in the surrounding eight adjacent locations. In the following example, the middle location has three live neighbors:

		X
X	?	
X		

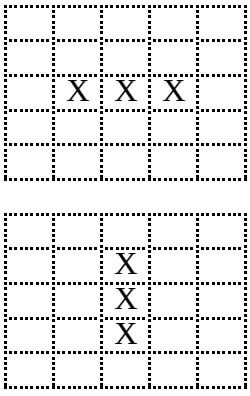
The rules

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

1. A location that has one or fewer neighbors will be empty in the next generation. If there were a cell in that location, it dies of loneliness.
2. A location with two neighbors remains "stable"—i.e. if it contained a cell, it still contains a cell. If it were empty, it's still empty.
3. A location with three neighbors will contain a cell in the next generation. If it were unoccupied before, a new cell is born. If it currently contains a cell, the cell remains.

4. A location with four or more neighbors will be empty in the next generation. If there were 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 cannot effect other births and deaths in that generation. If this condition is violated, the order that you happen to go through the grid computing the next generation will affect the answer you get. To keep the two generations separate, you will need to work on two versions of the grid, one that keeps the data for the current generation and another "scratch" grid which allows you to compute and store the next generation without affecting the current grid.

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



The gridADT

To help you out, we are providing a simple library of routines which can store and manipulate grids and draw them to the graphics window. The library exports a new type **`gridADT`** (ADT = *Abstract Data Type*, something we will spend much time on during the 106B part of the quarter.) The grid stores an integer for each location. Grids are organized into rows and columns, using a **zero-based** indexing scheme. The corner in the upper-left-hand corner is considered Row 0, Column 0 and the lower-right corner would be Row 9, Column 19 for a grid of size 10 rows and 20 columns. The gridADT stores an integer for each location rather than a simple boolean value to allow you to track the age of each cell. 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. Our grid drawing functions lightens the cells as they age to visually represent the cell life cycle.

From the **`grid.h`** header file, you can learn about the operations supported on gridADT variables. That header file is attached to this handout. The executive summary is that it includes functions to create new grids, set and retrieve values for each location, copy the values from one grid to another, and draw the grid in the graphics window.

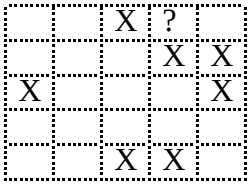
Boundary conditions

As you may have realized, the notion of a "neighbor" becomes a little fuzzy when you're working with locations along the boundary. What about an edge or corner cell that has walls for neighbors? Your program should allow the user to select one of the following three boundary strategies:

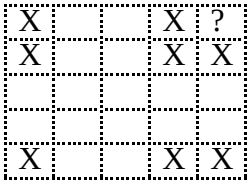
Plateau Mode: Consider all locations off the grid to be empty.

Donut Mode: Wrap the grid around on itself both vertically and horizontally. The world becomes a *torus* or donut shape. Any reference that disappears off the right-hand side is mapped around to the left-most column of the same row. Similarly, any reference

past the top reappears on the bottom, and so on. As a specific example, suppose you are considering location (0, 3) of the 5 by 5 grid below (marked with a '?'). In plateau mode, this square would just have 3 neighbors, but in donut mode, it has 5 neighbors:



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



Mirror Mode: References off the grid are bounced back as though the wall were a mirror. Sometimes this means the reflection of a cell gets counted as its own neighbor. Using the *first* picture above, the '?' square has 4 neighbors in mirror mode if the square itself is blank. If '?' is alive, the square has 5 neighbors (since the "neighbor" directly above the '?' square is a reflection of the square itself). A corner cell counts as its own neighbor three times in mirror mode—once each for the reflections in the horizontal and vertical walls and once again for the diagonal reflection.

The starting configuration

To get your colony underway, your program should offer the user a chance to read in a colony configuration file or seed the grid randomly. If the user chooses a random start, use the functions in the random library to pick some arbitrary cells to be born. Choose a "birth probability", let's say 40%, and then iterate over all locations in the grid giving each a cell that chance to become live. Set up the random grid to be of a fixed size, something like 30 rows by 40 columns seems to be a good choice. You may want to start with a smaller grid at first, for easier development and debugging.

If the user instead wants to start with a known configuration, you will read the grid data from a text file. All input files obey the following format, and you may assume that the files are properly formatted:

```
20                                  <- Number of rows in grid
25                                  <- Number of columns
-----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 should read the file line-by-line with **`ReadLine`** and use string library functions such as **`StringToInteger`** and **`IthChar`** to convert the strings into the data you need. We haven't yet gotten to file processing, but the minimal amount necessary for this assignment you can pick up from skimming section 3.4 from the text. The following example code opens the file named "Spiral", reads it line-by-line and prints out each line read:

```
FILE *configFile;
string line;

configFile = fopen("Spiral", "r");
if (configFile == NULL) Error("Couldn't open file!\n");
while (TRUE) {
   line = ReadLine(configFile);
   if (line == NULL) break;     /* stop at end-of-file */
   printf("%s\n", line);
}
fclose(configFile);  /* always close files when you're done */
```

Program strategies

This is a pretty tough program and will take some time. We recommend that you do not leave it for the night before. It is important to think carefully about your strategy for solving this problem before starting to code.

Incremental Development and Testing: Don't try to solve it the entire task at once. Instead, focus on implementing features one at a time. You can start by working on the basic Life simulation, using the plateau mode boundary. Seed your grid with a known configuration to make it easier to verify the correctness of your simulation. For example, build a bar with the following code:

```
gridADT grid;

grid = CreateGrid(10,10);
SetValueAt(grid, 5, 5, 1);
SetValueAt(grid, 5, 6, 1);
SetValueAt(grid, 5, 7, 1);
```

This puts a horizontal bar in the middle of the grid. When you run your simulation on this, you should get the alternating bar figure on page 2 that repeatedly inverts between the two patterns.

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

Decomposition: Think very carefully about how you're going to structure your program. This program is an important exercise in learning how to put decomposition to work in practice. Decomposition is not just some frosting you layer on the cake when it's done, it's the tool that helps you get the job done. With the right decomposition, this program is much easier to write, to test, to debug, and to comment! With the wrong decomposition, all of the above can become a real chore.

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

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

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

Avoid the "extra-layer-around-the-grid": At first glance, some students find it desirable to add a layer around the grid—an extra row and column of imaginary cells surrounding the grid. There is some appeal in forcing all cells have exactly 8 neighbors by adding "dummy" neighbors and setting up these neighbors to have the right values for the current boundary strategy. However, in the long run this approach introduces more problems than it solves and leads to confusing and inelegant code. Before you waste time on it, let us tell you up front this is a poor tradeoff and a strategy to avoid.

Some facts about the gridADT

The **gridADT** is different than other variables in that you can pass a grid as an argument to a function, change the grid (such as by calling **SetValueAt**) and after the function returns, the grid will retain the changed values. (This is unlike other parameters we've seen which work on a local copy and no changes are reflected back to the calling procedure). For example, consider this code:

```
void ChangeGrid(gridADT grid)
{
   int col;

   for (col = 0; col < 10; col++)
      SetValueAt(grid, 3, col, 5); /* set values in row 3 to 5*/
}

main()
{
   gridADT grid;

   grid = CreateGrid(10, 10);
   SetValueAt(grid, 3, 3, 8);
   ChangeGrid(grid);
   /* after function call, values in grid HAVE changed */
}
```

After **main** calls **ChangeGrid**, all of the values in row 3 of the grid have been changed to 5, including location (3,3) which we previously set to 8. (For those of you looking ahead, gridADT is a pointer type and thus the contents of the grid are passed by reference.) This makes it possible for you to pass a grid to a function to compute and sets the values for the next generation and when that function returns, the grid will retain the newly assigned values.

Remember that you will need two grids: one for the current generation and a separate scratch grid to set up the next generation without interfering with the current one. Only after you have computed all the values for the next generation will you use the **CopyGrid** function to copy all the results from the scratch grid into the current grid and update the graphics window by calling **DrawGrid**.

You never need more than two grids. Since grids happen to be a bit expensive you should create only the two that you need and no more.

Other random notes and suggestions

- As mentioned earlier in the handout, you should store the age of a cell in the grid rather than just a simple live-or-dead boolean value. Each generation that a cell lives through adds another year to a cell's life. The grid-drawing functions will use the age information to slowly fade cells as

they age. A newly born cell is drawn as a dark dot and with each generation the cell lives it will slightly fade until it stabilizes as a very faint cell in generation 8.

- Error-checking is an important part of handling user input. Where you prompt the user for input, such as asking for file to open or a boundary strategy choice, you should validate that the response is reasonable and if not, ask for another response.

- While you are in development, it is helpful to circumvent your code that prompts for a filename and a mode and just force your simulation to always open "Simple Bar" in plateau mode, "Mirror" in mirror mode, a random colony, or whatever you are currently working on. This will save you from having to answer all the prompts each time when you do a test run.

- The extended graphics library function **`UpdateDisplay`** can be used to immediately flush the drawing to the screen. Alternatively, function **`Pause`** can be used to flush the display and then wait a few seconds in order to allow you to see what happened before going on. This can be helpful to slow things down when you're trying to debug.

- You should continue computing and drawing successive generations until the user indicates they are done by clicking the mouse button. You can call the **`MouseButtonIsDown`** function from the extgraph library to test whether the mouse button is currently held down.

- Although you might feel tempted, you should not use any global variables for this assignment. Global variables can be convenient but they have a lot of drawbacks too, and in this class we will use them only sparingly and with good reason. For any of our assignments, you can assume that we will explicitly tell you if we expect you to use any globals. In absence of any other statement to the contrary, you should assume that globals are not allowed.

- When you prompt the user for the name of the file to be opened, youu can read their response with the simpio.h **`GetLine`** function.

- As with the quilt assignment, on a Mac you may need to set the monitor to display 256 colors to see the colors. To change it, use the Monitors & Sound control panel. You also may need to increase the amount of heap memory for the CW project to support the memory needed. Set the heap size under the Edit->Project Settings->PPC Target to something like 2 or 3K depending on the size of your window. Don't miss out, the color version can be simply breathtaking!

Coding Standards, Commenting, Style

This program has quite a bit more substance than your last, however, the added coding complexity doesn't release you from also living up to our coding standards. Remember what we've talked about in class about readability and decomposition. Carefully read over the handouts we gave you and keep those goals in mind. Strive for consistency. Be conscious of the style you are developing. Proofread. Comment thoughtfully.

Test colonies

In the assignment folder, there are several starting 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. Here is a brief description of some of the test files:

SimpleBar	Simple alternating bar
TicTacToe	Larger 3 by 3 arrangement of alternating bars
Stable	Colony that is immediately stable in donut mode and quickly evolves to stable configurations for mirror and plateau (different in each mode)

Mirror	Colony of cells clinging along border that is completely stable in mirror mode
Cheshire	Starts as a “cat” and ends up as a stable square of four cells
Spiral	Interesting symmetric pattern that infinitely repeats
Snowflake	Large intricate symmetric colony that cycles between 3 different patterns.
Seeds	Starting colony of seeds that will evolve into the repeating Snowflake pattern
Flower	Symmetric pattern that evolves to a stable colony of 30 or cells
Spaceships	Two little ships that repeatedly walk across grid to the east (good for testing donut mode wraparound)
Glider	A little walking colony that travels southeast across the grid (good for donut)
Glider Gun	A colony of cells that repeatedly fires off gliders heading southwest.

Feel free to make up your own and share them with others!

Accessing Files

As before, on the class web site, there are two folders of starter files: one for Mac CodeWarrior and one for Visual C++. Each folder contains the grid library and interface file and an incomplete version of the source file `life.c` with some starting #defines, you should create a new project using these files. There are also some colony test files and a compiled version of the program that you can run to see the behavior of a working program.

Deliverables

You should turn a manila envelope containing a printout of your well-structured, well-commented code, along with a disk containing your entire project in class next Friday. Everything should be clearly marked with your name, CS106X and your section leader's name!

If you can read this, you've got too much spare time. Get working on your assignment!

```
/* File: grid.h
 * Last modified on Wed Oct 3 21:03:11 1996 by jzelenski
 * ------------------------------------------------------
 * Defines an abstraction for a two-dimensional grid of integers.  This ADT
 * is not entirely general, in that it was designed to specifically support
 * the 106X Life assignment.  Grids are numbered (row,col) starting at the
 * upper left corner.  The row and columns are zero-based indexes, thus
 * location (0,0) is the upper left corner.
 */

#ifndef _grid_h
#define _grid_h

#include "genlib.h"

/* Constants: MAX_ROWS and MAX_COLS
 * --------------------------------
 * These define the absolute largest size grid one may create.
 */
#define MAX_ROWS 48
#define MAX_COLS 64


/* Type: gridADT
 * -----------------
 * Provides an abstract data type for manipulating 2-dimensional
 * grids.  The type is a purely abstract type in this interface, defined
 * entirely in terms of  the operations.  The client has no access to the
 * record structure used to implement the actual type.
 */
typedef struct gridCDT *gridADT;


/* Function: CreateGrid
 * Usage: grid = CreateGrid(10,20);
 * --------------------------------
 * Dynamically allocates enough memory for the underlying representation
 * and initializes it to represent an empty grid of numRows by numCols.
 * All locations are initialized to zero. An attempt to create a grid of
 * a size larger than permitted by the constants will raise an error.
 */
gridADT CreateGrid(int numRows, int numCols);


/* Function: FreeGrid
 * Usage: FreeGrid(grid);
 * ----------------------
 * Frees the storage associated with the grid. (You don't need
 * to be worried about freeing things at this point in quarter.)
 */
void FreeGrid(gridADT grid);


/* Function: NumRows
 * Usage: rowCount = NumRows(grid);
 * --------------------------------
 * Returns the number of rows in the grid.
 */
int NumRows(gridADT grid);
```

```
/* Function: NumCols
 * Usage: columnCount = NumCols(grid);
 * -----------------------------------
 * Returns the number of columns in the grid.
 */
int NumCols(gridADT grid);


/* Function: SetValueAt
 * Usage: SetValueAt(grid, row, col, newValue);
 * --------------------------------------------
 * Sets the value at the given location to the specified new value.  If the
 * location is out of bounds for this grid, an error message is generated
 * and the program halts.
 */
void SetValueAt(gridADT grid, int row, int col, int value);



/* Function: GetValueAt
 * Usage: value = GetValueAt(grid, row, col);
 * ------------------------------------------
 * Retrieves the value at the given location and returns it.  If the
 * location is out of bounds for this grid, an error message is generated
 * and the program halts.
 */
int GetValueAt(gridADT grid, int row, int col);


/* Function: CopyGrid
 * Usage: CopyGrid(source, destination);
 * -------------------------------------
 * Copies all the values from the source grid to the destination grid.
 * Afterwards, the destination grid has exactly the same contents as the
 * source grid.  The source grid is unchanged.  The grids must be the same
 * size in order to copy.  If they are not, an error message is generated
 * and the program halts.
 */
void CopyGrid(gridADT from, gridADT to);


/* Function: DrawGrid
 * Usage: DrawGrid(grid);
 * ----------------------
 * Draws the current state of the grid in the Graphics Window.  This will
 * erase the window completely and then draw the grid of cells in a box
 * centered in the grahics window.  The cell size is chosen to be as large
 * as will fit given the grid geometry, so larger grids have smaller cells
 * and vice versa.  At startup, the grid chooses a random color to
 * color the cells, and each cell will displayed in a shade which tells
 * its age.  Cells that have just been born (i.e. that have value 1) are
 * the darkest, they get lighter with age as the values go to 2, 3, and
 * so on.  The cells stabilize as very faint at generation 8 and older.
 * You probably will want to call UpdateDisplay() after calling
 * DrawGrid() in order to immediately flush the drawing on-screen.
 */
void DrawGrid(gridADT grid);

#endif
```