Lab 01 - Life

Due: September 18, 2016

```
"Death is the single greatest invention of life."
-Steve Jobs
```

1 Introduction

The purpose of this lab is to give you some experience with the syntax and basic features of the C programming language.

In this lab, you will be programming Conway's Game of Life. The Game of Life, created in 1970 by British mathematician J. H. Conway, simulates a population of lifeforms (or organisms) over a sequence of generations. The simulation takes place on a two-dimensional grid; you will store this data in a one-dimensional array configured in row-major order, so this lab will afford you practice with arrays in C. A description of the Game of Life with an example can be found below.

Since for many of you this is your first time programming in C, we have done the design work for you, providing function prototypes, each of the function skeletons, and ample comments to help you. All of the concepts needed for this lab are covered in the C Primer document, which can be found on the course website. You may wish to refer to it throughout the lab.

1.1 The Game of Life

As stated above, Conway's Game of Life takes place on a two-dimensional grid of cells. Each cell can be inhabited by at most one organism. The game starts off with an arbitrary initial population (dictated by whether a particular cell contains an organism or not). Occupied cells are referred to as "alive," whereas unoccupied cells are "dead."

From this initial population the next generation of organisms is obtained by applying the following rules:

- The neighbors of a cell are the 8 cells that immediately surround it vertically, horizontally and diagonally.
- If a cell is alive and has 2 or 3 live neighbors, it will remain alive in the next generation.
- If a cell is alive and has fewer than 2 live neighbors, it will die of loneliness.
- If a cell is alive and has 4 or more live neighbors, it will die of overcrowding
- If a cell is dead and has exactly 3 live neighbors, a new organism will be born in that cell. Otherwise, it remains dead in the next generation.
- All births and deaths take place at exactly the same time, so that all the cells' neighbors are counted simultaneously (based on the current generation) before the next generation is produced. It is possible for a new cell to be born based on counting a neighbor in the current generation that will be dead in the next generation.

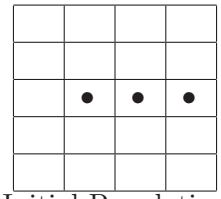
The rules for "Life" assume an infinite grid in all directions. Since we are limited to a finite-size grid, it is unclear how to count neighbors of cells that are on the grid borders, which do not have all eight neighbors.

Note: For this lab, we shall assume that any neighbor that does not exist in the grid is a dead cell.

You can find a Life simulator at `http://pmav.eu/stuff/javascript-game-of-life-v3.1.1/`. You may wish to use this to ensure that you understand the rules and to determine whether or not your implementation is working correctly.

1.1.1 Example

Let's look at a 5x4 grid (where a cell with a dot is alive and all other cells are dead):



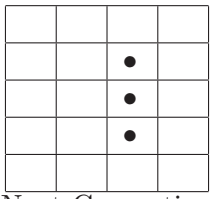
Initial Population

Here we have three cells that are alive. For each cell we will first count how many of its neighbors are alive. The upper-left cell is not adjacent to any living cells, so its neighbor count is zero. The same goes for all the cells along the top and bottom rows. Moving to the left-most cell of the second row, we see it has exactly one living neighbor, the one diagonally to the lower right. Going through the entire grid this way will result in the following count of living neighbors for each cell.

0	0	0	0
1	2	3	2
1	1	2	1
1	2	3	2
0	0	0	0

Neighbor Count

We now combine the neighbor count with the information of whether it was alive or dead to generate the new population. The second cell of the second row has a neighbor count of two. If it had been alive, it would have stayed alive according to the rules outlined. However, since it was dead it will stay dead in the next generation. The third cell of the third row also has a neighbor count of two, but because it was alive it stays alive. Just above this cell is an example of a dead cell coming alive since it has three living neighbors. The next generation will look like the following:



Next Generation

2 Assignment

You will be filling in the provided stencil file, *life.c*, using the C programming language. You must complete each of the provided functions and should not add any additional ones. Each of your functions should have the behavior specified in the explanatory comments, making any and all function calls listed in the comments.

2.1 Installing the Stencil

To get started, run the following command in a terminal:

```
cs033_install lab01
```

This will install the stencil code (and a copy of this handout) in your *~/course/cs033/lab01/* directory.

2.2 Storing the Game Board in a One-Dimensional Array

In other languages, such as Java, it is fairly straightforward to use a 2D array to store structures like the two-dimensional grid used by the Game of Life. However, 2D arrays in C are somewhat more complex; to simplify things, you should store the game board in a one-dimensional array configured in row-major order.

The get_index() function that you will fill in allows you to calculate an index value for a given row and column; using this index, you can access the value in your 1D array that corresponds to the value at the given row and column of the board.

2.3 Filling in the Stencil

There are ten functions for you to fill in within the stencil code. More comprehensive comments for each function have been provided there, but you will find below a brief explanation of each of these functions.

- `do_life()` - This function will execute the Life algorithm for a number of iterations. By default, the number of iterations is set to 10. Initially, `do_life()` will need to create a second array of the same dimension as the initial state array. Since the generation updates for each cell are based on the previous generation, you cannot make any changes to the previous generation until the new generation has been completely calculated, meaning you will need two arrays rather than one. These arrays will be passed around to the various functions in the Life program. `do_life()` must print each generation of the game board and should call `update()` to get the next generation of the game board.
- `get_index()` - This function will retrieve the index of the 1D array that corresponds to a given row and column number, assuming zero-based indexing.
- `update()` - This function should perform one iteration of the Life algorithm. For each cell, it should determine what the state in the next generation will be and set that cell's state in the

next generation array. Note that this approach does not generate the entire neighbor count array before generating the next population. Instead, the approach calculates a cell's next generation value just after counting its number of neighbors.

- `get_next_state()` - This function will determine the next state of a single cell according to the rules of Life specified above. It should also use the function `is_in_range()` to ensure that a valid row and column have been input, terminating the program if the input is invalid.
- `is_in_range()` - This function should determine if the provided row and column are on the game board, assuming zero-based indexing.
- `count_alive_neighbors()` - This function should return the number of alive neighbors that a cell has by adding the results of `is_alive()` together for all 8 neighboring cells. It should terminate the program if the input is invalid.
- `is_alive()` - This function should return 1 if the cell at the given row and column is alive and 0 if it is dead. Moreover, it should return 0 if the input is out of range. This means that calling `is_alive()` on a cell with a row and column of -1 is valid. You will find that this will simplify your implementation of `count_alive_neighbors()`.
- `set_alive()` - This function should set the specified cell to be alive in the provided array, terminating the program if the input is invalid.
- `set_dead()` - This function should set the specified cell to be dead in the provided array, terminating the program if the input is invalid.
- `print_arr()` - This function should print out a visual representation of the board.

2.4 Compiling and Running Your Code

To compile your code, you should `cd` into the directory with your code and run

```
make
```

from the command line to build your program. If there are errors in your code, the compiler (`gcc`) will let you know so you can fix them.

Once your code is error-free, `gcc` will create a binary (executable) file for you to run. You can run this by running the command

```
./life <board_file> <num_rows> <num_cols>
```

Here you may assume that the input `num_rows` and `num_cols` are the correct number of rows and columns in `board_file`. You will need to recompile this binary any time you make changes to your code.

If you would like to clean up your working directory after running your program, run

```
make clean
```

from the command line.

2.5 Testing

As you write your code, pepper it with `assert()` statements, asserting something about the program state for each function. `assert()` tests a condition (which is provided as its only argument), doing nothing if the condition was true, and aborts the program if the condition was false, printing the assertion that failed with its line number. To use `assert()`, you'll need to include the header file `<assert.h>`.

Note that `assert()` statements are generally used for programmer errors, and will terminate your program immediately if it fails. In almost all future assignments you will be asked to perform other types of error-checking (for example, whether the user inputs are valid) where `assert()` statements may not be appropriate. We will cover what to do in those cases in the Maze assignment.

To test your program, you are provided with a test case in the lab stencil, *sample_board.txt* that contains a grid of cells with dimension 9x7. To do this, run the two commands below:

```
./life sample_board.txt 9 7 > my_output.txt
diff -ZB my_output.txt sample_out.txt
```

`>` redirects the output of the program to the file *my_output.txt*.
`diff` prints the differences between the output of your program and the correct output in *sample_out.txt*. For more info on `diff`, type `man diff` on your terminal console. Here, we are using the following flags:

- -Z: ignores trailing whitespace at the end of each line
- -B: ignores completely blank lines

If your program is correct, `diff -ZB my_output.txt sample_out.txt` should not output anything.

3 Getting Checked Off

Once you've completed the lab, run `33lab_checkoff lab01`

Remember to read the course syllabus for information about course requirements and policies regarding labs and assignments.