Lab 01 - Life

Due: Sunday, September 15, 2019 at 10:00 PM

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.

Note: For this lab, having a partner is *optional*. We encourage you to collaborate with other students within our collaboration policy! For all future labs, you will be required to indicate your partner on a form that we will send out, and you will not be able to have the same partner for more than once.

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:

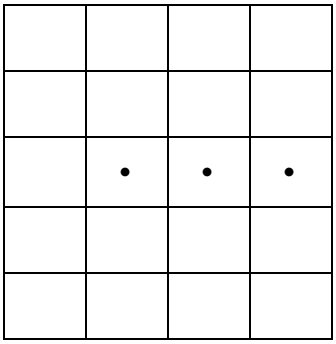
- The neighbors of a cell are the 8 cells that immediately surround it vertically, horizontally and diagonally.
- If a cell is alive and has 2 or 3 live neighbors, it will remain alive in the next generation. • If a cell is alive and has fewer than 2 live neighbors, it will die of loneliness.
- If a 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. Thus, 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):



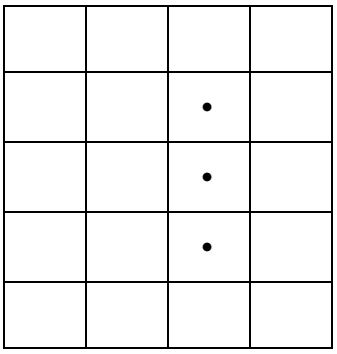
Initial Population

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:

```
cs0330_install lab01
```

This will install the stencil code (and a copy of this handout) in your **`~/course/cs0330/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. You will be using 2D arrays in the upcoming Maze assignment!

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(int rows, int cols, int array[rows * cols], int steps)`** - 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(int row, int col, int num_cols)`** - This function will retrieve the index of the 1D array that corresponds to a given row and column number, assuming zero-based indexing.
- **`update(int rows, int cols, int old_array[rows * cols],`**
  `int new_array[rows * cols])` - 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(int rows, int cols, int array[rows * cols], int row,`**
- **`int col)`** - 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(int rows, int cols, int row, int col)`** - This function should determine if the provided row and column are on the game board, assuming zero-based indexing.
- **`count_alive_neighbors(int rows, int cols, int array[rows * cols],`**
  `int row, int col)` - 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(int rows, int cols, int array[rows * cols], int row, int col)`** - 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(int rows, int cols, int array[rows * cols], int row, int col)`** - This function should set the specified cell to be alive in the provided array, terminating the program if the input is invalid.
- **`set_dead(int rows, int cols, int array[rows * cols], int row, int col)`** - This function should set the specified cell to be dead in the provided array, terminating the program if the input is invalid.
- **`print_array(int rows, int cols, int array[rows * cols])`** - 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. This will execute a script (called a Makefile) that compiles and builds your program using the **`gcc`** compiler. 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 executable 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. If you cannot track down your issue or you are getting segmentation faults, check out the following section on GDB to learn how to track down your bugs

3 gdb

`gdb` is a debugger that you can use to step through your C programs and ensure that your code is behaving as you expect it to. The main tasks that **`gdb`** performs include:

- Starting your program
- Making your program stop when certain conditions are true

- Examining what has happened when your program has stopped
- Changing things in your program so you can experiment with correcting the effects of one bug and continue your program

This document will cover the first three of those tasks.

gdb is best run on an executable file that was compiled using the **-g** flag, which provides debugging symbols so that gdb can refer to variables by name and reference lines of source code. The syntax for running gdb on the executable **`hello`** is

```
gdb hello
```

This will start gdb and permit you to run gdb commands. The debugger is terminated with the **gdb** command **`quit`**.

Below is an explanation of several tasks you can perform with gdb that may be helpful in debugging.

3.1 Backtrace

Backtrace in **gdb** allows you to see the sequence of function calls that occurred up to this point.

This is useful to run after your program has segfaulted in GDB so you know where to begin looking. In addition to running after a program has crashed it can also be run when you are waiting at a breakpoint (see next section). Inside of gdb it can be run using:

```
backtrace
```

Or with the following shorthand:

```
bt
```

3.2 Setting a Breakpoint

A breakpoint is a place in your code where you want the execution of your program to pause.

To set a breakpoint in your program at the start of a function, use the gdb command **`break [file:]function`**. For example, if you wish to create a breakpoint at the start of the function **`bar()`** in the file **`foo.c`**, you would use one of the following commands:

```
break foo.c:bar
break bar
```

To set a breakpoint at line 6 of **hello.c**, you can use one of the following commands:

```
break hello.c:6
break 6
```

A breakpoint can be deleted by running **clear** in the place of **break**, or **delete <num>** on the breakpoint given by **num**.

Running **info break** will print a list of all the currently-set breakpoints.

3.3 Stepping and Continuing

Once your code has stopped (but not terminated), you have several options how to proceed.

- **next** will execute the next line of code, including the entirety of any functions called in that line.
- **step** will execute the next line of code, stepping into and stopping at the first line of any function that line may call.
- **continue** will resume execution of your code until the next breakpoint or the termination of the program.
- **finish** will resume execution of your code until the next breakpoint or the current function returns; if it is the latter, this command also prints the function's return value (if any).

To repeat the previous command, you can just send a blank line and gdb will execute the last-executed command. This is very useful if you want to quickly step through your program.

3.4 Printing

You may print any variable from gdb using **print**. For instance, if we had an **int** variable **x** with a value of 33, we could print it this way:

```
print x
>>> $1 = 33
```

GDB knows the type of variables in the program and will thus represent them correctly. Note that **char*** variables will be printed as null-terminated strings! You can also use other C syntax, such as indexing into an array:

```
print int_arr[3]
>>> $1 = 90
```

4 Getting Checked Off

Once you've completed the lab, run **`33lab_checkoff lab01`**. If you have a partner, you and your partner both have to run the script individually.

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