FA17 MP6 - Game Of Life

Created by Gao, Jianxiong, last modified on Oct 19, 2017

DUE: Thursday, October 19th, by 10:00pm

MP6 - Game Of Life

This week you will implement the famous Game Of Life in C. Your functions will involve pointers and problem solving with arrays.

Game Of Life Background

The **Game of Life**, also known simply as **Life**, is a cellular automaton devised by the British mathematician John Horton Conway in 1970

The universe of the Game of Life is a two-dimensional orthogonal grid of square *cells*, each of which is in one of two possible states, *alive* or *dead*. Every cell interacts with its eight *neighbours*, which are the cells that are horizontally, vertically, or diagonally adjacent. At each step in time, the following transitions occur:

1. Any live cell with fewer than two live neighbours dies, as if caused by under-population.
2. Any live cell with two or three live neighbours lives on to the next generation.
3. Any live cell with more than three live neighbours dies, as if by over-population.
4. Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

The initial pattern constitutes the *seed* of the system. The first generation is created by applying the above rules simultaneously to every cell in the seed—a cell can either die, become live or have no change, and the discrete moment at which this happens is sometimes called a *tick* (in other words, each generation is a pure function of the preceding one). The rules continue to be applied repeatedly to create further generations.

The game board is composed of many cells. A game board has a Width (the number of columns) and a Height (the number of rows). A cell location is specified by its column and row.

- A game board has Width*Height cells in total
- Each cell contains an integer. 0 represents the dead cell and 1 represents live cell.
- Game boards are two-dimensional, having rows and columns. In this MP's code, the game boards will be represented as **one-dimensional arrays**.

Background- Representing 2-D Game Boards as 1-D Arrays

In this MP, the game board will be represented as a 1-D array. The size of the array is Width*Height. In your functions the game board will be passed to you as pointer (for example **int *game_board**). For example, to access the 10th element of the array, use the syntax "game_board[9]" (without the quotes, and remember that index starts from 0).

How do we organize 2-D game board as a 1-D Array? If the game board has width N and height M, we arrange the data in the array so that the first N values are the values of the first row. The second N values are the second row, and so on. The array will have a total of N*M elements.

An example game board of width 4 and height 3 might have values

$$\begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

This game board would be represented as a one-dimensional array by first arranging all the values from the first row, then the second row, and so on

$$game\_board = [0 \quad 0 \quad 0 \quad 1 \quad 0 \quad 0 \quad 1 \quad 0 \quad 0 \quad 0 \quad 1 \quad 1]$$

How can you access the value of a cell at a particular row and column? For a given row and column index, it is possible to calculate the index in the one dimensional array using the game board Width, the row index and the column index.

For example, if you want to access the value at **column index 2** and **row index 1** (the value 1 in this example, remember that indexes start from 0), we can use the fact that the **Width is 4** to access the value using

game_board[1*4 +2], where 4 is the game board width, 1 is the row index we want and 2 is the column index we want

In the functions you write this week, do not create large, static 2-D arrays for the game board. Work with the given pointers to the one-dimensional array (defined in the functions below) or you will lose points.

Update Game Board

For each cell of the game board, a live cell is represented as 1, a dead cell is represented as 0.

Each cell has eight surrounding neighbors.

The game board is updated step by step (each step is a generation).

For each step, a live cell keeps alive if it has 2 or 3 live neighbors. It turns into a dead cell if it has more than 3, or less than 2, live neighbors.

A dead cell turns into a live cell only if it has **exactly** three live neighbors.

Notice that at step *i,* the status of a cell at step *i+1* is determined only based on the status of itself and its surrounding neighbors at step *i*.

Pieces

This week you have quite a bit of code already in your folder. Let's discuss each of the included code in more detail:

updateBoard.h: This header file provides function declarations of the functions that you must write for this assignment.

updateBoard.c: The source file for your functions. Function headers for all functions are provided to help you get started. **Your code goes here!**

main.c: This main function reads input and calls your functions from updateBoard.c. You won't need to modify this unless you want to write some custom tests.

include/ & lib/: These folders contain library files for this MP. You don't need to modify them.

makefile- this file contains commands to compile all your code, including the given functions. You shouldn't need to modify this file.

life*.dat: Test files are included to test your code.

Your Assignment

- You will write all your code in the provided file **updateBoard.c**. You should not have to modify any other files, although you may modify other files for testing if you want. We will only grade **updateBoard.c**
- Implement the countLiveNeighbor function
- Implement updateBoard function
- Implement aliveStable function

```
int countLiveNeighbor(int* board, int boardRowSize, int boardColSize, int row, int col);
```

input: board is the 1-D array of the input game board. boardRowSize and boardColSize are the Height and Width of the game board respectively, together they define the number of rows and number of columns of the game board. row and col is the location of the cell that you need to count the alive neighbors for.

output: The number of the alive neighbors of cell at (row,col).

This function calculates the number of alive cells at the given row and col. A live cell is represented as 1, a dead cell is represented as 0.

```
int updateBoard(int* board, int boardRowSize, int boardColSize)
```

input: board is the 1-D array of the input game board. boardRowSize and boardColSize are the Height and Width of the game board respectively, together they define the number of rows and number of columns of the game board.

output: The game board is updated to the next step. A live cell stays alive if it has 2 or 3 alive neighbors, otherwise the cell dies. A dead cell turns alive if it has exactly 3 live neighbors. A live cell is represented as 1, a dead cell is represented as 0.

This function updates the game board.

```
int aliveStable(int* board, int boardRowSize, int boardColSize)
```

input: board is the 1-D array of the input game board. boardRowSize and boardColSize are the Height and Width of the game board respectively, together they define the number of rows and number of columns of the game board.

output: The function returns 1 if the current board stays the same for the next step. Otherwise return 0.

This function checks if the game board is going to change or not for the next step. It is used in main.c to check if the game should be stopped.

Specifics

- Your program must be written in C in the file **updateBoard.c** provided for you. **Make sure your code compiles!**
- Your function's outputs must be correct.
- Don't change the function definitions for these problems, or the test functions will fail. You may write additional helper functions if you wish in updateBoard.c
- Don't import files or libraries that were not imported for you.
- Be sure to access the game board using the one-dimensional arrays defined in the functions. Do not attempt to create two-dimensional arrays in your functions.
- Your code must be well commented and include an introductory paragraph (in updateBoard.c, near the top)
- Do not 'svn add' any compiled file to your subversion directory. Doing so results a penalty to your grade.

Building and Testing

While you should get in the habit of writing your own tests, we have provided a test program that individually checks each of the functions implemented (unit tests). It checks the output of your functions with the output from the gold solution.

Compilation

You can compile the project using the given Makefile. A Makefile is used by the program GNU Make (learn more here http://www.gnu.org/software/make/) to build code projects. If you look in the file, you'll see it consists of a number of gcc commands to compile and link your code automatically. This is useful for complicated projects. You use make using the command "make" and different targets, which specify which commands will be run. The target "all" is the default.

The command to build your project and compile your code is simply

```
make
```

This will compile your code and create executable files. To delete the compiled files, type "make clean". If you want to compile a specific target you see in the file, type make and the target name

Testing Script (To Do)

You can **test** your functions by running the test executable from the mp6 directory as:

```
./test
```

The test executable displays the results of a series of tests for each function.

Running the code and debugging

Once compiled, execute the code using test 1 (life1.dat) as following:

```
./gameoflife < life1.dat
```

You may find debugging frustrating (many of us do). A useful tool is gdb (http://www.gnu.org/software/gdb/), introduced in lab last week. You can use this tool to set breakpoints, step through your code and print values.

You can start gdb using

```
gdb test
```

Type run to begin execution. You can also run gdb gameoflife to run the ./gameoflife executable, and begin execution with run < life1.dat.

Your code is written in functions.c. As an example, to set a breakpoint at line 12, you would type

```
break updateBoard.c:12
```

You can also break at a function, for instance "break updateBoard.c:countLiveNeighbor". Once at a break point you can us the next and step commands to execute lines, and the print command to view variables. For instance

```
print game_board[0]
```

would print the first value of the game_board

Do not 'svn add' any compiled file to your subversion directory. Doing so results a penalty to your grade.

Grading Rubric

Functionality (90 pts)

- 30pts- countLiveNeighbor function works correctly
- 40pts- updateBoard function works correctly
- 20pts- aliveStable function works correctly

Style (10 pts)

- 5pts - Introductory paragraph explaining what you did. Even if it is just the required work. Write this in updateBoard.c
- 5pts - Code is clear and well commented.

Some point categories in the rubric may depend on other categories. If your code does not compile at all, you may receive a score close to 0 points.

No labels