COS 351 Lab: Life Begins

In the moodle page for this lab you will find starter code.

Our Objectives

Why are we doing this? First, to get your "C" skills (back) in good shape; second, to have a simulation that we all know and can discuss; third, so we have a working program that we can time and optimize and "parallel-ize" in a later lab.

Introduction

This is background and preparation for later in the semester. It sets the groundwork for some later project(s). Your task is to build a simple simulation of cellular life. Here's some background on the simulation: "Life" is a cellular automaton formulated by the British mathematician, John Orton Conway in 1970. It simulates simple life in a Petri dish, life that follows four simple rules: (courtesy of Wikipedia)

> The universe of the Game of Life is an infinite 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 overcrowding.
> 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—births and deaths occur simultaneously, 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.

Tasks

Your objective is to write a version of the game of Life in C language. The input file will contain a first line that tells the size of the "board" - the dimensions of the 2-D grid. What follows are an arbitrary number of lines that contain x and y value pairs indicating populated cells at the start of the simulation. Here's a very simple example of input:

```
10 10
3 1
3 2
3 3
```

This declares a 10 x 10 grid and has 3 populated cells: (3,1), (3,2) and (3,3). We will make use of this simulation with much larger grids and with much greater populations, but your simulation will be easier to debug with this size input.

Command Line Interface

The program takes two arguments on the command line, both integers:

- The first (`argv[1]`) is the number of generations to run the simulation.

- The second (`argv[2]`) is the frequency for displaying the grid. If the second argument is zero, then the grid is never displayed ***but*** the final values are written to stdout in the format of an input file (see above), suitable for use to run as a further simulation.

The input to the program reads from stdin. Later versions may support filenames specified as an argument, but that isn't necessary today.

Here's an example of invoking the program:

```
./life 50 10 < ten.data
```

This invokes the binary "`life`" in the current directory, runs 50 generations, printing out the grid every 10 generations (beginning with the first – so we can verify our input). The input comes from `stdin`, redirected from the file `ten.data` in the current directory.

Here's a simple design (in pseudo code) of the main() function:

```
main
   parse command line arguments
   read inputfile (which will also allocate, initialize the grid)
   for each generation
        display the grid every so often (i.e., modulo the period)
                (optionally sleep a second for debug/display purposes)
        swap the edges (into the "halo")
        compute the next generation
```

Two dimensional arrays in C are tricky. C support of 1-D arrays are straightforward, but a 2-D array is really better thought of as a 1-D array of 1-D arrays. You want to allocate X-many ***pointers***-to-data (int? char? double?) and then allocate Y-many items of data. C syntax allows you to reference these data as a 2-D array; `mydata[i][j]` is the syntax. Of course, you could use pointer dereferencing syntax and pointer arithmetic, but the array reference syntax is more readable.

Step #1

Compile and run the framework (starter code) for this program – reading an input file, construction of the array (2D grid) based on the first line of input, and printing out the grid. For this stage, it won't compute the next generation. Just make sure you can get this much working (and demonstrate it to your instructor during lab/class) before proceeding to the next step.

Step #2

Now comes the real work. Fill in the functions for computing the next generation.
How to compute the next generation? Look at the rules for Life; for each cell you need to know the

number of neighbors (or “neighbours” in the UK spelling). How do you count the number of “live” cells that surround a given cell?

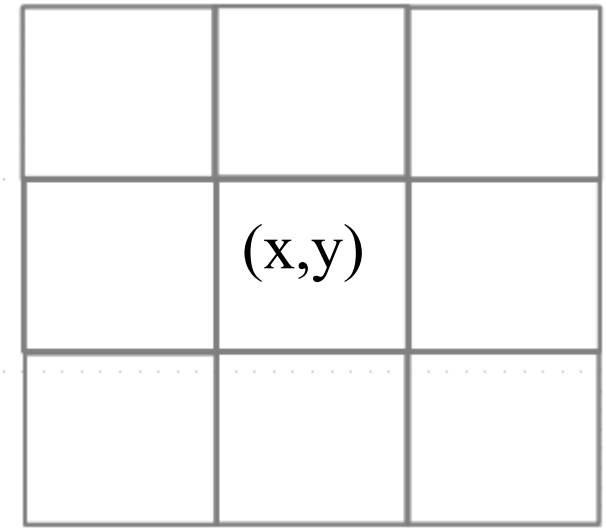
You also need to prepare the grid of data for the computation. You need to copy the left hand edge to the right hand buffer, and the right hand edge to the left hand buffer. You also need to copy the top row to the bottom buffer and copy the bottom row to the top buffer. Since C arrays are zero-based, we'll use the 0 row and column as buffer. The max size row, column values will also be buffer. So, if your input specifies a 10 x 10 board, then the buffers will be in rows 0 and 11, and columns 0 and 11. The "real" data will be in rows 1-10 and columns 1-10.

Know Your Terminology:

Moore's neighbors vs. von Neuman neighbors – do you know the difference? Neighborhood vs. neighbors – do you know the difference?

- The von Neuman neighborhood around the center square has only 4 neighbors.
- The Moore's neighborhood around the center square has 8 neighbors.

Use the following diagram:



The center square is labeled with the location coordinates: “(x,y)”. Label each square around the center square in relative terms, that is, relative to x and y. (example: the square to the left is “(x-1, y)”. Now write the code to compute the count of live cells among the eight (Moore's) neighbors. How many arithmetic operations did this take?

Design Decision #1

What happens at the edge of the board? What do you do at the edges where the neighbors are off the edge? Are the edge spaces “dead zones”? Another approach is to “wrap around” so that the cell to the left of the left edge is the rightmost cell and the cell above the top edge is actually in the bottom row.

To deal with edge conditions, you need to add logic to your code to look for those cases where the neighbors will be off the board. Those “if” statement will be executed for every cell in order to find the cases that matter. There is a run time cost for that. Calculate how many operations are needed for each cell to determine if it is at an edge. (An operation is either an arithmetic operation (e.g., “+”) or a

comparison (e.g., "<"). Multiplied by the number of cells (especially for a large grid) that can be a lot of extra operations.

Another approach is to have a "collar" or extra cells around the board, either a) as a dead zone whose values are 0 and never changed, or b) as a copy of the corresponding edge on the opposite side. With approach "b)", the border is copied at the start of each generation to speed the computation and avoid the extra logic. The top row (just below the top buffer) is copied to the bottom buffer; the bottom row (just above the bottom buffer) is copied to the buffer row above the top row. Such a duplicate row (sometimes called a "halo") makes for simpler code but will the cost of the memory copying be faster or slower than the additional code that it replaces? ***This "halo" is the approach we will use.***

Design Decision #2

One way to implement this is with two grids, where the values in one grid determine the values for the next generation in the other grid. Since we will want to use very large grid sizes, having a single grid *might* be a better idea. How might you do this with a single grid? What is the trade-off in such an approach?

With two grids you need a way to switch back a forth between which is the target and which is the source. You can do this with pointers and with passing pointer to the grids as parameters to a function. Another way is to make a 3-D array – where the first dimension is simply which grid holds the current data and which holds the results (the next generation). ***Use that approach for this lab.***

Timings

Once you have a running version of life, make some timings:

```
$ time ./life 500 0 < myinput.txt
```

will give you millisecond resolution on run times for the whole program. With an 80x24 grid, how many generations do you need to run before you get a second or more of run time? How many generations do you need to run with a 1000 x 1000 grid? Conversely, what size grid can you use before you begin to notice 1 second or greater run time for 500 generations?

You can get a more granular measurement of the compute time by making `times()` calls around the section of code that computes the next generation. That way you can avoid all the I/O overhead and just look at the computational costs. The difference between the return value of those two calls is the number of clock ticks elapsed during the computation. Use `sysconf(_SC_CLK_TCK)` to know how many ticks per second, that is, to convert to seconds.

Submitting Your Code

Use moodle to submit your files. One of the files must be a makefile with a default target that builds an executable named "`life`". I will test the program with the following commands:

```
$ make
$ ./life    # with various arguments and data files for input
```