CSCI 1120 (Low-Level Computing), Fall 2019

Homework 8

Credit: 20 points.

1 Reading

Be sure you have read, or at least skimmed, the assigned readings for classes through 10/30.

2 Programming Problems

Do the following programming problems. You will end up with at least one code file per problem. Submit your program source (and any other needed files) by sending mail to `bmassing@cs.trinity.edu` with each file as an attachment. Please use a subject line that mentions the course and the assignment (e.g., "csci 1120 hw 8" or "LL hw 8"). You can develop your programs on any system that provides the needed functionality, but I will test them on one of the department's Linux machines, so you should probably make sure they work in that environment before turning them in.

1. (20 points) Your mission for this assignment is to first complete and then modify a starter program that plays mathematician John Conway's Game of Life, described below. (You can also find more information on the Web. The Wikipedia article seems good.)

   The Game of Life is not so much a game in the usual sense as a set of rules for something called a cellular automaton: There are no players, and once the initial configuration is established, everything that happens is determined by the game's rules. The game is "played" on a board consisting of a rectangular grid of cells. Some cells are "live" (contain a simulated organism); others are "dead" (empty). At each step, a new configuration is computed from the old configuration according to the following rules:

   - For each cell, we look at its eight neighbors (top, bottom, left, right, and the four diagonal neighbors) and count the number of cells that are live. Note that this count is based on the configuration at the start of the time step.
   - A dead cell with exactly three live neighbors becomes live; otherwise it stays dead.
   - A live cell with two or three live neighbors stays live; otherwise it becomes dead (of isolation or overcrowding).
   - What do to with cells at the boundary of the board is tricky; to me the option that makes the most sense is to only consider the cells that exist; so for example a cell at one of the four corners only has three neighbors.

   To visualize how this works . . . I found some Web sites that let you "play" online, but none I found immediately appealing. I have a version that uses text-based graphics that you should be able to use on our classroom/lab machines. Download file `game`, tell Linux it's an executable file with the command `chmod u+x game`, and run it with the command `./game`. (If you're curious, the program is written in C++ using a library called `ncurses` to do text-based graphics.) (*NOTE* that this program has sometimes crashed when I try to demo it in class. I can't make this happen consistently and am at a loss to know what might be wrong. If it doesn't work for you please let me know where and when you tried it.)

To implement the basic algorithm in a program, you pretty much need to use 2D arrays. Writing a complete program is straightforward but more than I want to ask you to do; instead I am providing starter code that does everything but the actual update of the grid. To give you some practice with `make` and to reduce code duplication, I've split the code into multiple files and included a `Makefile`.[1] The files:

- `Makefile`. Comments in the file tell you how to use it.
- `game-of-life.c`: Main code for first program. You should not change this file.
- `game-of-life-util.h`: Declarations of most functions used in the main program. You should not change this file for the first program, and you may not need to for the second.
- `game-of-life-util.c`: Definitions of functions declared in `game-of-life-util.h`. You *do* need to change this file; look for the comment `FIXME` to find out what you need to do for the first program. You may not need to make additional changes for the second program.
- `game-of-life-random.c`: Minimal version of second program, included so `make` works. You'll need to write most of this program yourself, but you will likely find it useful to adapt code from the first program.

To make it easier to download all these files, I've made a ZIP file containing all of these files, so it will probably be simplest just to download that and unzip it (command `unzip` on our machines).

NOTE that if you prefer to download individual files, you should use your browser's "download" or "save" function to obtain the makefile rather than copying and pasting text. This is because copy-and-paste will likely replace the tab characters in the file with spaces, with bad consequences (since tabs are semantically significant in makefiles.)

NOTE also that you might want to put the files for this homework in a separate directory, so its makefile doesn't conflict with the one you use for compiling other programs.

You might want to start by just downloading everything and trying compiling the two programs. They won't do anything useful yet, but this will let you check that you know how to build them when you do start writing code.

For this assignment you will develop two programs:

- The first program takes two command-line arguments, the name of a file containing an initial configuration and the number of steps to "play"; it prints the initial configuration and then updated configurations for the specified number of steps. Sample output for input files `game-of-life-in1.txt` and `game-of-life-in2.txt`:

```
[bmassing@dias04]$ ./game-of-life game-of-life-in1.txt 4
Initial board:
. . . . . .
1 1 1 . . .
```

[1] If you really want to do this homework on a system that doesn't support `make`, talk to me about options.

```
. . . . . .
. . . . 1 .
. . . . 1 .
. . . . 1 .

Board after step 1:
. 1 . . . .
. 1 . . . .
. 1 . . . .
. . . . . .
. . . 1 1 1
. . . . . .

Board after step 2:
. . . . . .
1 1 1 . . .
. . . . . .
. . . . 1 .
. . . . 1 .
. . . . 1 .

Board after step 3:
. 1 . . . .
. 1 . . . .
. 1 . . . .
. . . . . .
. . . 1 1 1
. . . . . .

Board after step 4:
. . . . . .
1 1 1 . . .
. . . . . .
. . . . 1 .
. . . . 1 .
. . . . 1 .



[bmassing@dias04]$ ./game-of-life game-of-life-in2.txt 4
Initial board:
. . . . . . . .
. 1 1 . . . . .
. 1 1 . . . . .
. . . . . . . .
. . . . . . . .
. . . . . 1 1 .
. . . . . 1 1 .
```

```
    . . . . . . . .

    Board after step 1:
    . . . . . . . .
    . 1 1 . . . . .
    . 1 1 . . . . .
    . . . . . . . .
    . . . . . . . .
    . . . . . 1 1 .
    . . . . . 1 1 .
    . . . . . . . .

    Board after step 2:
    . . . . . . . .
    . 1 1 . . . . .
    . 1 1 . . . . .
    . . . . . . . .
    . . . . . . . .
    . . . . . 1 1 .
    . . . . . 1 1 .
    . . . . . . . .

    Board after step 3:
    . . . . . . . .
    . 1 1 . . . . .
    . 1 1 . . . . .
    . . . . . . . .
    . . . . . . . .
    . . . . . 1 1 .
    . . . . . 1 1 .
    . . . . . . . .

    Board after step 4:
    . . . . . . . .
    . 1 1 . . . . .
    . 1 1 . . . . .
    . . . . . . . .
    . . . . . . . .
    . . . . . 1 1 .
    . . . . . 1 1 .
    . . . . . . . .
```

The starter code doesn't do the update; your mission is to fix it so it does, by filling in the update function in `game-of-life-util.c` (look for comment `FIXME`). The two examples are a very minimal check that the program works as intended; you should try some of your own tests as well! *HINT:* You might want to look at the code for `print_board()` (in the common-code files) and `read_board()` (in the starter program) for examples of how to work with the 2D arrays this program uses.

- The second program adapts the first one to experiment with generating the initial board "randomly", rather than reading it in from a file, and observing how the game evolves with different board sizes and parameters for generating "random" data. Your program should get all its input from command-line arguments:
  - Size of board (a `long`)[2].
  - Seed for random number sequence (a `long`).
  - Fraction of cells that should be "live" initially (a `double`) (not really a great name, explained better later).
  - Number of steps (a `long`).
  - Optionally, the word "print".

  The program should then work as follows:
  - Generate an initial configuration by first calling `srand()` and then calling `rand()` once per cell, setting the cell to `true` based on whether what `rand()` returns, divided by `RAND_MAX`, is less than the specified fraction.
  - Count and print the number of cells that are "live", and print the board if the "print" option was specified.
  - For each step, update the board (no change from first program), but then rather than always printing the new board, count and print the number of cells that are "live", and print the board if the "print" option was specified.

  (My idea here is that such a program could be a prototype for something that would allow you to try out more-sophisticated ways of generating "random" configurations and observing how they evolve.)

  Here are some sample executions:

```
[bmassing@dias02]$ ./game-of-life-random 6 5 .5 1 print
28 live cells at start (fraction 0.777778)
Initial board:
1 1 . 1 1 1
1 1 1 . 1 1
. 1 1 1 1 1
1 1 1 1 1 1
1 . . 1 1 .
1 . 1 1 . 1

8 live cells after step 1 (fraction 0.222222)
Board after step 1:
1 . . 1 . 1
. . . . . .
. . . . . .
1 . . . . .
1 . . . . .
. 1 1 1 . .
```

[2] The `long` data type is a "long integer". In C `int` only has to be big enough for 16-bit values; `long` has to be big enough for 32-bit values. If you're curious about support for 64-bit values, `long long` (C99 only) does that.

```
[bmassing@dias02]$ ./game-of-life-random 6 7 .5 1 print
18 live cells at start (fraction 0.500000)
Initial board:
1 . . 1 1 .
. 1 . 1 . 1
. . . 1 1 .
. 1 1 1 1 .
1 1 1 . 1 1
1 . . . . .

11 live cells after step 1 (fraction 0.305556)
Board after step 1:
. . 1 1 1 .
. . . . . 1
. 1 . . . 1
1 . . . . .
1 . . . 1 1
1 . . . . .

(different seeds give different configurations)
```

```
[bmassing@dias02]$ ./game-of-life-random 6 5 .25 1 print
9 live cells at start (fraction 0.250000)
Initial board:
. 1 . 1 1 1
. . . . 1 .
. 1 . . . .
. . . . . .
1 . . . . .
. . 1 . . 1

7 live cells after step 1 (fraction 0.194444)
Board after step 1:
. . . 1 1 1
. . 1 1 1 1
. . . . . .
. . . . . .
. . . . . .
. . . . . .

(smaller fraction gives fewer live cells)
```

```
[bmassing@dias02]$ ./game-of-life-random 100 5 .5 10
```

```
5141 live cells at start (fraction 0.514100)
2647 live cells after step 1 (fraction 0.264700)
2453 live cells after step 2 (fraction 0.245300)
2437 live cells after step 3 (fraction 0.243700)
2265 live cells after step 4 (fraction 0.226500)
2181 live cells after step 5 (fraction 0.218100)
2127 live cells after step 6 (fraction 0.212700)
2001 live cells after step 7 (fraction 0.200100)
1873 live cells after step 8 (fraction 0.187300)
1852 live cells after step 9 (fraction 0.185200)
1784 live cells after step 10 (fraction 0.178400)
```

```
[bmassing@dias02]$ ./game-of-life-random 100 5 .25 10
2497 live cells at start (fraction 0.249700)
2789 live cells after step 1 (fraction 0.278900)
2338 live cells after step 2 (fraction 0.233800)
2382 live cells after step 3 (fraction 0.238200)
2243 live cells after step 4 (fraction 0.224300)
2137 live cells after step 5 (fraction 0.213700)
2158 live cells after step 6 (fraction 0.215800)
2145 live cells after step 7 (fraction 0.214500)
2028 live cells after step 8 (fraction 0.202800)
2035 live cells after step 9 (fraction 0.203500)
1899 live cells after step 10 (fraction 0.189900)
```

HINTS:

- For parsing command-line arguments, remember that to get a numeric value you should call `strtol` or `strtod`, and to compare two strings you need `strcmp` rather than `==`.
- You should probably start from `game-of-life.c` and adapt it, calling functions from `game-of-life-util.c` like it does. What you need to do is:
  * Add code to parse the new command-line arguments.
  * Replace the function that reads the initial configuration from a file with one that generates it "randomly".
  * Add a function to count the number of "live" cells.
  * Change the code that always prints the board so that instead it always prints the count of live cells but only prints the board if "print" was requested.

3 Honor Code Statement

Include the Honor Code pledge or just the word "pledged", plus *at least one of the following* about collaboration and help (as many as apply).[3] Text *in italics* is explanatory or something for you to

[3] Credit where credit is due: I based the wording of this list on a posting to a SIGCSE mailing list. SIGCSE is the ACM's Special Interest Group on CS Education.

fill in. For programming assignments, this should go in the body of the e-mail or in a plain-text file `honor-code.txt` (no word-processor files please).

- This assignment is entirely my own work. *(Here, "entirely my own work" means that it's your own work except for anything you got from the assignment itself — some programming assignments include "starter code", for example — or from the course Web site. In particular, for programming assignments you can copy freely from anything on the "sample programs page".)*
- I worked with *names of other students* on this assignment.
- I got help with this assignment from *source of help — ACM tutoring, another student in the course, the instructor, etc. (Here, "help" means significant help, beyond a little assistance with tools or compiler errors.)*
- I got help from *outside source — a book other than the textbook (give title and author), a Web site (give its URL), etc.. (Here too, you only need to mention significant help — you don't need to tell me that you looked up an error message on the Web, but if you found an algorithm or a code sketch, tell me about that.)*
- I provided help to *names of students* on this assignment. *(And here too, you only need to tell me about significant help.)*

4 Essay

Include a brief essay (a sentence or two is fine, though you can write as much as you like) telling me what about the assignment you found interesting, difficult, or otherwise noteworthy. For programming assignments, it should go in the body of the e-mail or in a plain-text file `essay.txt` (no word-processor files please).