CS120 - Computer Science I
Assignment #9   Fall 2014

Due: Monday November 10th

The purpose of this assignment is to give you some experience using two dimensional arrays. You will also use a library to aid in the display of your output.

In 1970, British mathematician John Conway proposed the Game of Life, a "cellular automata" game. Each position in a 2-dimensional grid represents a living "cell," which lives or dies in the next generation according to some simple rules involving each cell's eight neighbors (N, NE, E, SE, S, SW, W, NW):

1. A living cell with fewer than two neighbors dies due to loneliness.
2. A living cell with more than three neighbors dies due to overpopulation.
3. A living cell with two or three neighbors lives to the next generation
4. A cell is born if surrounded by exactly three living neighbors.

For this program, you are going to simulate Conway's Game of Life. Using a 2-D array of char, you will compute the next generation of the game, then display it. A living cell can be represented by a '1' in the position, and a space otherwise. You should perform the calculation for the next generation in a separate array, and once finished with the entire calculation, display it.

The result of each new generation is best displayed on the screeen when it does not scroll after each line. Therefore we will use a class called `CursWin`. This class provides capabilities very similar to the standard iostream cout class. It includes numerous functions (methods) - the complete documentation for the class is provided in the class files themselves. The main addition is that the user can "move" to a specific place on the screen before outputting values. This allows the user to place text anywhere on the screen. In our case, this will allow us to "draw" the game of life in place, rather than having it scroll off the screen as we output.

The main features that you will need are as follows:

1. You will need to #include "CursWin.h" in your main program.
2. To make an output window, declare something like:

   ```
   CursWin outw(0,0,20,70);
   ```

   This declares a Curses window with upper left (0,0) and lower right (20,70) [(row,col)]
3. Then write to the window, use it like you would a normal I/O stream, ie:

   ```
   outw << "Hello, world!" << cendl;
   ```

   The "normal" stream commands are included, prepended with a 'c', such as "cendl" above.
4. To move to a specific location within the window, use the Cmove(r,c) macro:

   ```
   outw << Cmove(10,20) << "This is the center of the screen" << cflush;
   ```

5. Something that is more important with CursWin than with iostream is the use of cflush (which will output the text without appending a new line) or cendl. If you do not use cendl or cflush, your output will NOT get to the screen until the program encounters one of these.
6. When you compile your program, you need to include both the Curswin.cpp file as well as the standard curses library:

   ```
   g++ assign9.cpp CursWin.cpp -lcurses
   ```

7. The program should be able to read an initial configuration from a file. This files contains the initial arrangement of alive and dead cells as a list of 900 (30x30 "world") of 1's and 0's. When the program starts it should ask the user if they want a random configuration or a configuration from a file. If they pick random random configuration, the program should generate 1's and 0's (alive and dead cells) randomly. If the user picks to load an initial configuration from a file, the program should ask for a file name, and the program should open that file and read in the initial configuration. Two sample files are given on the course schedule.

In addition to the proceeding required work, the following additions can be done for extra credit. You may do as many or as few of these as you want. Please include comments at the beginning of the program listing which of the extra credit problems you did.

1. Have the program also read the size of the "world" from the configuration. I.e. the file should begin with two integers defining the width and height of the "world". The program should read these first and create arrays, loops, etc. accordingly.
2. Have the program stop if it enters a constant configuration, i.e. if the configuration doesn't change between iterations. This will require comparing two successive "world" states to see if they are indentical (i.e. have exactly the same 0's and 1's). If two successive states are identical it means that the program has entered a steady-state and won't change any more.
3. Have the program stop if it enters a steady-state with a period of two, i.e. if the configuration starts to repeat between on every other iteration. This will require comparing "world" states that are one state apart to see if they are identical (i.e. have exactly the same 0's and 1's).
4. Go on-line and find an alternative set of rules, either ones using only two states (alive and dead) or ones using three or more states. Change the program to use the alternative rules and test it. In the comment at the beginning of the program list the new rules and describe how the program behaved with them.