Lab 1: Cellular Automaton

Due: February 5th at 11:59pm

Overview

The goal of this lab is to continue your familiarization with the C++ programming by implementing a small cellular automaton generator. Your program will make use of arrays, functions, and control flow. You should focus on modularization to reduce code duplication.

Deliverables

Your submission should minimally include the following code and supplemental files:

- `Makefile`
- `README`
- `cell_automaton.cpp`

Submission

You will submit using `handin35`. Be sure to place all relevant files in `cs35/labs/01`.

Cellular Automaton

In this lab you will program a basic cellular automaton that will output the trace of Rule 30. A cellular automaton is an abstract model of computation that is visualized using a grid of squares. Each row in the grid is generated from the previous row based on a set of rules. For example, in this lab, you will use the standard Rule 30 to generate your visual trace of the automaton. The computation rules are defined in the table below:

current pattern	1**1**1	1**1**0	1**0**1	1**0** 0	0**1**1	0**1**0	0**0**1	0**0**0
new state for center cell	0	0	0	1	1	1	1	0

As an example of how to apply the rules, consider the initial state and the resulting states of applying Rule 30 in the table below:

Index	1	2	3	4	5
initial state	1	0	0	0	1
resulting state	1	1	0	1	1

Consider the computation to produce the result in index 1: The initial state is 010 which produces an output in the center cell of 1. Although the leading 0 is not present in the initial state *10, it is implied. Starting at index 2, the application of the rule is more straightforward, e.g., 100 results in output 1 based on the Rule 30 table. At index 5, again, a trailing 0 is inferred such that 010 produces a 1.

This style of computation can then be repeated for a few iterations, producing the following visual grid where subsequent iterations move down:

1	0	0	0	1
1	1	0	1	1
1	0	0	1	0
1	1	1	1	1

In this lab, your program will generate similar visuals in the terminal based on Rule 50 using a "# " for a 1 and a "." for a 0 (see below for example output). For more information on cellular automaton and the Rule 30 automaton, the wikipedia pages are a good resource:

```
http://en.wikipedia.org/wiki/Rule_30
http://en.wikipedia.org/wiki/Cellular_automaton
```

Program Outline and Sample Output

In English-style pseudo code, your program will consist of the following operations:

```
Prompt the user for width of the grid

Initialize the first state

Print initial state

for width/2 iterations:
    Generate next state from current state
    Print next state
    Store next state as the current state
```

And here are two sample runs of a completed program:

```
bash >./cell_automaton
Enter the grid width: 10
 .  .  .  .  .  #  .  .  .  .
 .  .  .  .  #  #  #  .  .  .
 .  .  .  #  #  .  .  #  .  .
 .  .  #  #  .  #  #  #  #  .
 .  #  #  .  .  #  .  .  .  #
 #  #  .  #  #  #  #  .  #  #

bash >./cell_automaton
Enter the grid width: 20
 .  .  .  .  .  .  .  .  .  .  #  .  .  .  .  .  .  .  .  .
 .  .  .  .  .  .  .  .  .  #  #  #  .  .  .  .  .  .  .  .
 .  .  .  .  .  .  .  .  #  #  .  .  #  .  .  .  .  .  .  .
 .  .  .  .  .  .  .  #  #  .  #  #  #  #  .  .  .  .  .  .
 .  .  .  .  .  .  #  #  .  .  #  .  .  .  #  .  .  .  .  .
 .  .  .  .  .  #  #  .  #  #  #  #  .  #  #  #  .  .  .  .
 .  .  .  .  #  #  .  .  #  .  .  .  .  #  .  .  #  .  .  .
 .  .  .  #  #  .  #  #  #  #  .  .  #  #  #  #  #  #  .  .
 .  .  #  #  .  .  #  .  .  .  #  #  #  .  .  .  .  .  #  .
 .  #  #  .  #  #  #  #  .  #  #  .  .  #  .  .  .  #  #  #
 #  #  .  .  #  .  .  .  .  #  .  #  #  #  #  .  #  #  .  .
```

As you can tell, the initial state for your automaton is always a 1 in the center value. This is represented with the # in the top row of each of the outputs. In the provided skeleton code, you will find some "TODO" items to help guide your development. Most importantly, you will represent each row of the computation using an array of integers, and **you are only allowed to declare two of such arrays to store the current and the next.** There are some helper functions already declared for you that you will need to complete, but **you must declare and define at least one new function**.

Hints!

- **Test Early and Often**: As you develop your code, be sure to compile and run it incrementally to see how it is performing.
- **Modularization**: If you find that you are using the same bit of code over and over again, you should probably write a separate function for it.
- **Arrays are a lot like pointers**: Recall from class that an array is a lot like a pointer, where the value of a the array is the reference to the first value in the pointer.

- **Segfaults**: There is a decent chance that you will encounter a segfault. This occurs because you tried to read from or write to a piece of memory that you weren't allowed to. To debug your problem, focus on the reading/writing involving arrays. This tends be the source of many segfaults.

Extra Credit and Challenge Exercises

- **(Challenge) Rule 110**

  Create another C++ program, `rule_110.cpp`, that implement Rule 110 as described on the wikipedia page: `http://en.wikipedia.org/wiki/Rule_110`. In your README, **you should describe how you adapted your original automaton program to complete this one.** For example, discuss code reuse and adaption.

- **(0.25 pt) User Initial State**

  Add in an extra features to your program that allows the user to choose the initial state of the automaton by entering space separated 1's and 0's. For example,

```
Enter grid width: 7
Enter Initial State (enter -1 for default):
1 0 0 1 0 0 1
...
```

  Hint: Be sure to check (or require) that the user enter in enough (or not too many) 1's and 0's before proceeding. Optionally, instead of having the user enter in the initial state on a single line, you can also require the user to enter one number per line such that he/she must hit enter for each value in the initial state.

- **(0.5 pt)** Game of Life

  Create another C++ program, `game_of_life.cpp`, that implements at least two oscillators and one spaceship from Conway's Game of Life automatons. You can read more about them on Wikipedia here: `http://en.wikipedia.org/wiki/Conway%27s_Game_of_Life`.

  To similuate iterative processes, you can clear the terminal between each completed automation by printing 100 blank lines between:

```
for(int i=0;i<100;i++){ cout << endl;}
```

  And to slow down the automation, you can use the C library function `usleep()` like follows:

```
#include <unistd.h>//include this at the top of the program
...
usleep(500); //sleep for 500 milliseconds, or half a second
```

  More info on `usleep()` can be found in the man page by typing `man usleep` in the terminal.