Lab Assignment 7—What's the Point?
Using C Pointers

Overview:

In this lab you will get practice using C pointers. Specifically, you will write a program that implements a 1-D, 122-generation cellular automata (CA) simulation, sending each generation out the UART to be displayed on a remote communications terminal. To perform the UART communication, you'll use subroutines from a UART library provided by your instructor, and you will make use of C pointers to read and write elements in the CA data array.

Background:

A CA simulation applies a set of "black/white" coloring rules to individual cells in an array of characters. For this lab the array will be a **1-D, 61-character array** initially containing **30 spaces on either side of a single `'*'` character**, where the `'*'` character represents black and the space character represents white. The simulation proceeds by applying the **Wolfram Rule 26** set of coloring rules to each cell (character) in the current array to produce a "next generation" array. The Wolfram Rule 26 rule set only considers the color of **three adjacent cells**—the current cell and its immediate left/right neighbors. (**Important point**: The first and last cells of the array are always white (i.e. spaces), and transformation rules never change their color.)

Table 1 below contains the eight transformation rules that make up Wolfram Rule 26. Be aware that each new generation must be created from the previous generation *in toto*, i.e. **changes due to the rules are not applied to the current array**. The rules only describe the creation of the *next* generation of the array.

Table 1. Wolfram Rule 26 rule set for a 1-D cellular automata

Rule	Next generation cell color
1	If a cell is black and so are both its neighbors, make it white
2	If a cell is black and so is its left neighbor, but not its right neighbor, make it white
3	If a cell is white and both its neighbors are black, make it white
4	If a cell is white, its left neighbor is black, and its right neighbor is white, make it black
5	If a cell is black, its left neighbor is white, and its right neighbor is black, make it black
6	If a cell is black and both its neighbors are white, make it white
7	If a cell is white, its left neighbor is white, and its right neighbor is black, make it black
8	If a cell is white and so are both its neighbors, make it white

Since the rules describe all possible black/white states involving three cells, there are 2^3=8 rules per rule set. And since there are eight possible states per set, there are at most 2^8=256 such rule sets (of which Wolfram Rule 26 set is one). Be aware that not all rule sets generate productive or interesting future generations, and all are dependent on the initial coloring of the cellular array. Once your program is finished, experiment with different starting conditions. It will prove quite interesting! (Even better—implement additional rule sets and get some bonus points.)

See http://mathworld.wolfram.com/ElementaryCellularAutomaton.html for much more on cellular automata.

Lab Procedure:

Begin by accepting the Lab07 assignment. After the remote assignment repo has been created for you, open **Git Shell** and navigate to your `D:\EECE3624Labs` solution directory. Enter the following command to clone the remote `Lab07` assignment repo to your local storage device.

```
D:\EECE3624Labs> git clone <URL-to-remote-assignment-repo> Lab07
```

Add the newly created `Lab07` project to your `EECE3624Labs` solution using **Atmel Studio** and remember to set `Lab07` as the startup project. Double-click the `Lab07.c` file to see the starter code. And don't forget to **cite my code as your starting point** in your report.

In Solution Explorer you'll notice the `Lab07` *Libraries* folder contains a `libUARTLibrary.a` file. This is a **pre-compiled library** of basic serial communication functions you must use when transmitting the CA simulation patterns to the serial terminal program (PuTTY) for display. In order to use these functions, you'll need to add a `#include UARTLibrary.h` declaration at the top of your code. To learn more about the functions available in `libUARTLibrary.a`, double-click the `UARTLibrary.h` file in Atmel Studio and study its contents. It all there…

Finally, design and implement your program. The following pseudocode describes how your code should behave.

```
Create a 61-character array (must be created inside main())
Initialize the array (via subroutine)
Initialize the UART
Loop 122 times: (main)
      Display the current array (via subroutine)
      Update the current array (via subroutine)
```

Your simulation must display **122 generations** and must use **subroutines** to make changes to the array (i.e. initializing, displaying, and updating). Here are prototypes for those subroutines. **Yours must be *exactly* the same.**

```
void initializeArray(unsigned char *cp);
void displayArray(unsigned char *cp);
void updateArray(unsigned char *cp);
```

Notice that **a single unsigned character pointer is passed to each subroutine**. Since the array is only visible inside `main()`, the array is "out of scope" to the subroutines and only accessible via the passed-in pointer. **Do not change this behavior**.

After declaring a 61-character array in `main()`, call `initializeArray()` to fill it with 30 spaces, one star, and 30 more spaces. This can be done easily using the `strcpy()` function to fill the array. This uses everything as pointers (NOTE: Remember arrays in C are really pointers).

Function protytype:

```
char* strcpy(char* destination, const char* source);
```

Use example:

```
char str1[20] = "C programming";
char str2[20];
// copying str1 to str2
strcpy(str2, str1);
```

Since `strcpy()` uses **two signed character pointers** to access the source and destination arrays. Most importantly, the source pointer (the second argument) **can actually be a literal string**, so you can "hardcode" your initial string directly inside `initializeArray()`. *NB:* Using `strcpy()` requires that you `#include` the `string.h` library in your program. See the `string.h` library documentation for more information. Google is your friend here…

This is a **one-week lab**. You must **compile and demonstrate your working code** in lab on **Friday, October 28th**. When compiling, **there must be no compiler warnings**! As usual, submit your lab report via Canvas and your code via GitHub. Follow **good code design practices** (use of constants, #defines, subroutines, etc.) and make sure your code is **well organized**, **efficient**, and **properly formatted**. And of course, have fun!

Bonus opportunity: Create an infinite loop structure that **displays a menu** of rule sets to choose from, **prompts the user** to choose a set, **accepts their choice**, **performs the simulation** using their selected rule set, and **then repeats the process**. In your code comments, state which rule set(s) you implemented. Also, add a parameter to `updateArray()` to specify the particular rule to use. Do this well and you can pick up a lot of bonus points!