Course Outline-CSE 105 (Structured Programming)
Department of Computer Science & Engineering
East West University

Assignment 2: Game of Life

Abstract
Write a *C* program that plays the Game of Life. Accept as inputs the size of the board, the initial configuration, and the number of generations to play. Play that number of generations and display the final configuration of the board.

Outcomes
After successfully completing this assignment, you should be able to:–

- Develop a *C* program that uses two-dimensional arrays
- Allocate memory for the arrays at run time
- Pass arrays as arguments to functions

John Conway's Game of Life
The Game of Life was invented by the mathematician John Conway and was originally described in the April 1970 issue of *Scientific American* (page 120). The Game of Life has since become an interesting object of mathematical study and amusement, and it is the subject of many websites.

The game is played on a rectangular grid of cells, so that each cell has eight neighbors (adjacent cells). Each cell is either occupied by an organism or not. A pattern of occupied and unoccupied cells in the grid is called a *generation*. The rules for deriving a new generation from the previous generation are these:–

1. *Death.* If an *occupied* (having value 1) cell has 0, 1, 4, 5, 6, 7, or 8 occupied neighbors, the organism dies (less than 2 of loneliness; greater than 3 of overcrowding).
2. *Survival.* If an occupied cell has two or three neighbors, the organism survives to the next generation.
3. *Birth.* If an unoccupied (having value 1) cell has precisely three occupied neighbors, it becomes occupied by a new organism.

Examples can be found at http://www.math.com/students/wonders/life/life.html.

Once started with an initial configuration of organisms (Generation 0), the game continues from one generation to the next until one of three conditions is met for termination:

1. all organisms die, or
2. the pattern of organisms repeats itself from a previous generation, or
3. a predefined number of generations is reached.

Note that for some patterns, a new generation is identical to the previous one — i.e., a steady state. When this occurs, termination under condition #2 occurs. In some other common cases, a new generation is identical to the second previous generation; that is, the board oscillates back and forth

between to configurations. In rare cases, a pattern repeats after an interval of two or more generations. In this assignment, you will be responsible for terminating after a steady state is reached or an oscillation of two alternating patterns is reached.

In theory, the Game of Life is played on an infinite grid. In this assignment, your program will play on a finite grid. The same rules apply, but squares beyond the edge of the grid are assumed to be always unoccupied.

Implementing your program

Your program should be called **life.c** or **life.cpp** It needs to do several things:–

- Prompt for input to configure the program. (For extra credit, you may get the input from the command line; see below.)
- Allocate at least three arrays, each large enough to hold one generation of the game. Initialize one generation with the initial configuration in the approximate center of the board.
- Play the game for as many generations as needed according to the input *num_generation*.
- Print out the configuration of all generation

Input

The first line of input should include 2 numerical values to be interpreted as:–

n m

where

- *n* indicating the number of elements in the grid. For example if n=12 it will create a grid of 12×12.
- *m* is the number of generations to play. This value must be greater than zero. The program should halt prior to this number of generations if it determines that the game has reached a termination condition m=0 or all the value of of the grid is zero or we are in a steady state (where no value changes from current generation to the next one).

Following the first line of input should be a sequence of lines consists of a series of '1' and '0', indicating the occupied and unoccupied cells of the initial configuration. This sequence of lines is terminated by an end-of-file indication or by an empty line. *You must place this initial configuration in the approximate center of your board.*

For example, here is a simple pattern that happens to be a "still life" or steady state:–

```
11
11
```

That is, the next generation starting from this pattern produces exactly the same pattern.

Here is another still life pattern:–

```
010
101
101
010
```

The following pattern produces an oscillation between a vertical line of three occupied cells and a horizontal line of three occupied cells

```
1
1
1
```

Likewise, the following pattern is a well-studied one called the *R-Pentomino*.

011
110
010

This creates an interesting sequence of generations, including many sub-patterns that come and go, until it finally reaches a steady state after 1176 generations.

Also you can take input from the file. In that case you file input should look like following

12 1789

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

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

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

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

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

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

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

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

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

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

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

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

Where 12 is the 12×12 grid and 1789 is the number of generation to display. Then all the zeros and ones are value for your grid.

Allocating your arrays

There are two ways in *C* to create an array dynamically at run-time:–

- Use the **malloc()** function to allocate memory from *The Heap* and return a pointer to that memory. Although this is the most common practice in *C*, we will not yet have studied pointers at a suitable depth before this assignment is due. Therefore, you should not use this approach unless you already know what you are doing.
- Inside a function or compound statement, declare an array whose size is specified by a variable with a value determined at run time. For example, the following is legal in *C*:–

```
void Life(unsigned int n) {
    int A[n][n], B[n][n], C[n][n];
    /* use arrays A, B, and C to play the game */
    …}
```

When the function **Life** is entered, it allocates the three arrays on the stack based on the integer parameter ***n.***

Playing the Game
To play the game, you should set up a function such as

void PlayOne (unsigned int n, int Old[][], int New[][]) {
/* loop through array New, setting each array element to zero or one depending up its neighbors in Old.*/

} // PlayOne

This can be called by the function **Life** using:–

```
PlayOne(n, A, B);
```

The result is that **PlayOneGen** reads the contents of the first array argument and updates the array of the second argument. Subsequent generation might be called by

```
PlayOne(n, B, C);

PlayOne(n, C, A);
```

so that the generations cycle among the three arrays you allocated.
To test for termination conditions, you could adapt **PlayOneGen** to return values of zero or non-zero to indicate whether anything has changed. You could also construct another function to compare two arrays, returning zero if they are the same and non-zero if they are different, for example.

Testing
You should test your Game of Life with several initial conditions, including patterns that you find on the web (Wikipedia or http://www.math.com/students/wonders/life/life.html). When the graders test your program, they will use one or more standard input files comprising one initial line followed by several pattern lines.

Deliverables
You must provide the following:–

- **life.c/cpp** files.
- At-least 3 sample input and output files (if you are using files then just send your input and output files of different combination of input).
- At least one test case that demonstrates that your program works on a non-trivial pattern.
- A document called **README.txt**, or **README.pdf**, or **README.doc** summarizing your program, how to run it, and detailing any problems that you had. Also, if you borrowed all or part of the algorithm for this assignment, be sure to cite your sources *and* explain in detail how it works.

Programs submitted after 11:59 pm (BD time i.e. GMT +6) on due date (December 1) will be tagged as late, and will be subject to the penalty.

This assignment is worth thirty (30) of your points. *Your program* must *compile without errors in order to receive* any *credit.* It is suggested that before your submit your program, compile it again and be sure that it does not blow up or contain surprising warnings.

In case of problem you can knock us at lab time or mail us.
Md. Shamsujjoha
Faculty member, Department of Computer Science & Engineering, East West University
Email: dishacse@yahoo.com