Project 2 (40 points)  
 Assigned: Monday, September 3, 2018  
Due: Monday, September 10, 2018, 6:00 PM

Programming Assignment #2 — Game of Life



**CS-2303, System Programming Concepts, A-term 2018**

### Abstract

Write a *C* program that plays the *Game of Life*. Accept as arguments 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 these arrays as arguments to functions

### Before Starting

Read Chapter 5 K&R pertaining to arrays and sections §§7.5–7.7 regarding file access. Pay particular attention to §5.10 about command line access and §§5.7–5.9 about multi-dimensional arrays.

### 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* cell has 0, 1, 4, 5, 6, 7, or 8 occupied neighbors, the organism dies (0 or 1 of loneliness; 4 thru 8 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 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 two configurations. In rare cases, a pattern repeats after an interval of more than two generations. In still other cases (some not so rare), a pattern replaces itself by a fixed offset in one or both dimensions, thereby “flying” off the screen. 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.

Note that, according to the rules, all changes for each generation are considered to take place simultaneously. Unfortunately, your program has to work through the board square-by-square. That is why you always need to have at least two arrays: One holding the state of the board at the beginning of the turn, and one which will be filled in with the state of the board at the end of the turn (which will be the state at the beginning of the next turn). It turns out that you will also need a third array; more about this later.

### Implementing your program

Your program should be called **life**. It needs to do several things:

* Read the arguments to program from the command line.
* Read the initial configuration of the board from an input file.
* Allocate at least three arrays, each large enough to hold one generation of the game. Initialize the first generation with the initial configuration in the approximate center of the board.
* Play the game for as many generations as needed until one of the termination conditions above is met.
* Print out the final configuration, along with a message saying how many generations were played and under what condition the game terminated.

### Program Arguments and Input

The program should be invoked from the command line with the following arguments:

./life *NR NC gens input [print] [pause]*

where

* **NR** and **NC** are unsigned integers indicating the number of elements in the *y* and *x* directions if the grid, respectively. In other words, the number of rows and columns, respectively, in the grid
* **gens** 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.
* **input** is the name of a file containing a sequence of lines, each consisting of a sequence of **'x'** and **'o'** characters, indicating the occupied and unoccupied cells of the initial configuration. An **‘x’** indicates an occupied cell, an **‘o’** or a blank indicates an unoccupied cell. If a line is shorter than the width of the grid, cells to the right are considered unoccupied. If there are fewer lines in the file than the height of the grid, cells below are considered unoccupied. Note: The **‘o’** characters are included to make it easier to see the patterns in the file. When you print the grid, please only print blanks and **‘x’** characters.
* **print** is an optional argument with value of **'y'** or **'n'** indicating whether each generation (including generation 0) should be printed or displayed before proceeding to the next generation. If this item is missing, it defaults to **'n'.** Note: The user does not type the square brackets; they are used to denote that this argument is optional.
* **pause** is an optional argument with value of **'y'** or **'n'** indicating whether a keystroke is needed between generations. If this and/or the print item is missing, it defaults to **'n'**. It is not possible to specify a value for **pause** if you do not also specify a value for **print.**

After interpreting the program arguments, your program must open the input file, read its lines, and initialize the configuration in the approximate center of your board in the *x-* and *y-*dimensions. In other words, the pattern in the file starts in the upper-left corner of the grid; after you read it in, determine the number of lines and the greatest width, and shift the pattern down and to the right as needed to approximately center it.

#### Example patterns

Here is a simple pattern that happens to be a “still life” or steady state:

xx  
xx

That is, the next generation starting from this pattern produces exactly the same pattern. Here is another still life pattern:

oxo  
xox  
xox  
oxo

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

x  
x  
x

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

oxx  
xxo  
oxo

It creates an interesting sequence of generations, including many sub-patterns that come, go, and/or fly off the edge of the board, until it finally reaches a steady state after 1176 generations.

### Allocating your arrays

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

* Use the malloc() or **calloc()** functions to allocate memory from *The Heap* and return a pointer to that memory. This is the most common practice in *C*. We will study it in class shortly.
* In gcc or *C++,* inside a function or compound statement, declare an array whose size is specified by an expression at run time. For example, the following is legal in gcc:

void function(unsigned int x, …) {  
 int A[x], B[x], C[x];  
   
 /\* use arrays A, B, and C \*/  
 …  
} // Function

Unfortunately, this only works for single-dimensional arrays. For a multi-dimensional array, only the first subscript can be determined at run-time. The rest of the subscripts must be compile-time constants.[[1]](#footnote-1) Because this assignment calls for the grid of the Game of Life to be determined at run-time (in both dimensions), you cannot use it.

#### Allocating multi-dimensional arrays at run-time

§5.9 of Kernighan and Ritchie describe how the effect of a two-dimensional array can be achieved by allocating an array of pointers, each pointer of which points to another array of int. Suppose that you wish to allocate an array *B* of nr rows of nc columns each. One way is as follows:

int \*B[ ];  
unsigned int i, j, k;  
  
B = malloc(nr \* sizeof(int \*)); // Allocate array of nr pointers  
if (B) for (i = 0; i < nr; i++) {  
 B[i] = malloc(nc \* sizeof (int)); // Allocate one row  
 if (!B[i]) exit(-1); /\* error \*/  
}

Then the array element of row j, column k, may be accessed as follows:

B[j][k] /\* assuming that j < nr and k < nc \*/

There are other ways of solving the same problem.

*Sample code*

You will find sample code for this assignment on Canvas. The file **Life.zip** can be imported into Eclipse as a makefile project; feel free to use this as a starting point for your own program. It demonstrates how to read the command-line arguments, how to read from the input file, and how to create a two-dimensional array.

The file **Assignment2\_testcases.zip** contains some sample input files. Unzip it into the directory with your executable.

### Playing the Game

To play the game, it is suggested that you set up a function along the lines of the following to play one generation:

void PlayOne (unsigned int nr, unsigned int nc, int Old[ ][ ],

int New[ ][ ]) {  
 /\* loop through array New, setting each array element to

zero or one depending upon its neighbors in Old.\*/

} // PlayOne

This can be called by the function Life using:

PlayOne(nr, nc, A, B);

The result is that PlayOne reads the contents of the first array (argument A) and updates the second array (argument B). Subsequent generations might be played by

PlayOne(nr, nc, B, C);

PlayOne(nr, nc, C, A);

so that the generations cycle among three arrays. Note: this shows how the program would work, conceptually. You should probably not code it exactly this way. Instead, always call PlayOne() using the same variables as arguments, but each turn change which allocated array the variables point to. For example, if you call the arguments p and q, at the start of the program you would set p = A, q = B, spare = C. Then, at the beginning of the loop, you call PlayOne(x, y, p, q). At the end of the loop, to get ready for the next turn, set temp = p, p = q, q = spare, spare = temp. Do you see a pattern here? Think loop invariants – what is true at each point in the loop?

If the pause argument is set to **'y'**, the program should wait for one character of input from the keyboard between calls to PlayOne().

To test for termination conditions, you could adapt PlayOne to return values of zero or non-zero to indicate whether anything has changed. You should 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. When the graders test your program, they will use one or more standard input files containing with typical patterns. The program arguments will match the input files.

### Deliverables

This project must be carried out on the course virtual machine. Your submission must include the following:

* *Two* or more .c files and one or more .h file to implement your game.
* A makefile to build your assignment. The executable program must be called life. The makefile must be able to make any individual .o file or the entire application. It also must be able to make clean and **make docs** (by running doxygen).
* At least one test case that demonstrates that your program works on a non-trivial pattern.
* A document called README.txt, README.pdf, README.doc, or README.docx 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.

Before submitting your assignment, execute make cleanto get rid of extraneous files. Then export your project and your test case to an archive zip file named PA2\_*userName*.zip, where *userName* is replaced by your WPI username (i. e., login ID). Submit that zip file, along with some output from your test cases and your README file, to *Canvas*.

This programming assignment is named PA2*.* Programs submitted after the due date and time will be tagged as late, and will be subject to the late homework policy.

### Grading

This assignment is worth one hundred (100) 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 on a different platform from the one you have been using, just to be sure that it does not blow up and does not contain surprising warnings.

### Additional Notes

Command line arguments in *C* are explained in §5.10 of Kernighan and Ritchie*.* The function prototype of main() is

int main(int argc, char \*argv[]);

The elements of the argv array are strings, which we have not yet studied in this course. The argument argv[0] contains the name of your program, argv[1] is the value NR, argv[2] is the value NC, argv[3] is the value gens, and argv[4] is a string containing the name of the input file. The numeric values can be extracted using the function atoi(). The file name can be used directly in calls to fopen() . Sample usage is shown below:

#include <stdio.h>  
#include <stdlib.h>

FILE \*input;  
int k, nr, nc, gens;

if (argc < 5)  
 /\* report error in command line \*/

nr = atoi(argv[1]);  
nc = atoi(argv[2]);  
gens = atoi(argv[3]);

input = fopen(argv[4], "r");  
if (!input)  
 /\* report unable to open file \*/

/\* continue with print and pause arguments \*/

In Eclipse, you can specify command-line arguments when you create or modify a “Run configuration”.

You can put the input file in the same directory as your source file.

We will cover how to read from a file in class.

Be sure to create a makefile. Use the one from Lab 2 as a guide.

Sample code will be posted on Canvas.

Be sure to use Doxygen header comments on all the functions you write. Here is a big tip: Decide what you want each function to do, then write the header comment, then write the function. This forces you to decide what you want the function to do, before you spend time writing it. If you are unsure, make your best guess – you can always update the comments and code later. Be sure that you can run doxygen to generate the Web pages; we will talk about this in class.

Write and test your program step-by-step, always making sure you have a version which compiles and runs, even if incomplete. Consider writing the input and output functions first; then you can use them while testing the other parts of the program.

1. Kernighan and Ritchie do not allow dynamically-sized arrays at all. However, they do allow arrays with an unspecified size to be passed as arguments to function. In the case of multi-dimensional arrays, only the first subscript may be unspecified; the remaining subscripts must be known at compile time. This is discussed on page 112. [↑](#footnote-ref-1)