Ministry of Education of Republic of Moldova
Technical University of Moldova
CIM Faculty
Anglophone Department

Report

on APPOO

Laboratory Work Nr. 1

Performed by st. Gr. FAF-131 **Boldescu A.**

Verified by lect.univ **Drahnea V.**

Chișinău 2016

Laboratory Work № 1

Topic:

Develop simple implementation of Conway's Game of Life Automaton.

Task:

- Will use one OOP language. Otherwise you have to prove that your language have at least polymorphism This language should be cross platform, at least to be compiled on MAC
- You are not allowed to use external libs. (At all)
- Field are infinite.
- Your application will run in console.
- Your application will take -i(--input) argument with preset file path. Preset file will contain state of automaton which will be loaded in your app as first generation.
- Your application will take -n(--number-of-iterations) argument for number of required generation to be evaluated. After evaluation the final state of automaton will be outputted to console directly.
- Your application will take -o(--output) argument as output file path. If no output indicated then show final state in console. Output should contain all alive cells and not be bigger that is needed to include them(rectangle small as it is possible).
- Input and output state files format is an array of binary values.

  Eg.:

  00000
  00100
  00010
  01110
  00000

Code snippets C#:

Language which I used is C# you can compile it on MAC OS using Monodevelop and Xamarin Studio

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Text;

namespace Game_of_Life
{
    class Game_of_Life
    {
        int[,] generation;
        int[,] lastGeneration;


        int generationCount;

        int width;
        int height;

        public int GenerationCount
        {
            get { return generationCount; }
        }
        public Game_of_Life(int[,] newGrid)
        {
            generation = (int[,])newGrid.Clone();
```

```
            generationCount = 1;

            width = generation.GetLength(1);
            height = generation.GetLength(0);

            lastGeneration = new int[height, width];
        }

        private int Neighbours(int x, int y)
        {
            int count = 0;

            // Check for x - 1, y - 1
            if (x > 0 && y > 0)
            {
                if (generation[y - 1, x - 1] == 1)
                    count++;
            }

            // Check for x, y - 1
            if (y > 0)
            {
                if (generation[y - 1, x] == 1)
                    count++;
            }

            // Check for x + 1, y - 1
            if (x < width - 1 && y > 0)
            {
                if (generation[y - 1, x + 1] == 1)
                    count++;
            }

            // Check for x - 1, y
            if (x > 0)
            {
                if (generation[y, x - 1] == 1)
                    count++;
            }

            // Check for x + 1, y
            if (x < width - 1)
            {
                if (generation[y, x + 1] == 1)
                    count++;
            }

            // Check for x - 1, y + 1
            if (x > 0 && y < height - 1)
            {
                if (generation[y + 1, x - 1] == 1)
                    count++;
            }

            // Check for x, y + 1
            if (y < height - 1)
            {
                if (generation[y + 1, x] == 1)
                    count++;
            }

            // Check for x + 1, y + 1
            if (x < width - 1 && y < height - 1)
            {
```

```
                if (generation[y + 1, x + 1] == 1)
                    count++;
            }

            return count;
        }

        public void WriteNeighbours()
        {
            for (int y = 0; y < height; y++)
            {
                for (int x = 0; x < width; x++)
                    Console.Write("{0}", Neighbours(x, y));
                Console.WriteLine();
            }
        }
        public void ProcessGeneration()
        {
            int[,] nextGeneration = new int[height, width];

            lastGeneration = (int[,])generation.Clone();

            generationCount++;

            for (int y = 0; y < height; y++)
            {
                for (int x = 0; x < width; x++)
                {
                    if (Neighbours(x, y) < 2)
                        nextGeneration[y, x] = 0;
                    else if (generation[y, x] == 0 && Neighbours(x, y) == 3)
                        nextGeneration[y, x] = 1;
                    else if (generation[y, x] == 1 &&
                            (Neighbours(x, y) == 2 || Neighbours(x, y) == 3))
                        nextGeneration[y, x] = 1;
                    else
                        nextGeneration[y, x] = 0;
                }
            }

            generation = (int[,])nextGeneration.Clone();
        }
        public void DrawGeneration()
        {
            for (int y = 0; y < height; y++)
            {
                for (int x = 0; x < width; x++)
                    Console.Write("{0}", generation[y, x]);
                Console.WriteLine();
            }
            Console.WriteLine();
        }

        public int AliveCells()
        {
            int count = 0;

            for (int y = 0; y < height; y++)
                for (int x = 0; x < width; x++)
                    if (generation[y, x] == 1)
                        count++;

            return count;
        }
```

```
public static Dictionary<int,object> ReadFromFile()
    {
        string input = File.ReadAllText(@"h:\input.txt");
        int linecount = File.ReadAllLines(@"h:\input.txt").Length;
        int colcount = ((input.Split(' ').Length-1)/linecount)+1;
        Dictionary<int, object> result_values = new Dictionary<int, object>();
        result_values.Add(1, linecount);
        result_values.Add(2, colcount);
        int i = 0, j = 0;
        int[,] result = new int[linecount,colcount] ;
        foreach (var row in input.Split('\n'))
        {
            j = 0;
            foreach (var col in row.Trim().Split(' '))
            {
                result[i, j] = int.Parse(col.Trim());
                j++;
            }
            i++;
        }
        result_values.Add(3, result);
        // Console.WriteLine("{0} {1} {2}", input,linecount,colcount);
        return result_values;
    }
    static void Main(string[] args)
    {

        string filename = @"h:\output.txt";
        int nr_of_lines = (int)Game_of_Life.ReadFromFile()[1];
        int nr_of_cols = (int)Game_of_Life.ReadFromFile()[2];
        StreamWriter sw = new StreamWriter(filename, true);
        int[,] grid = (int[,])Game_of_Life.ReadFromFile()[3];

        Console.WriteLine("Enter maximum number of Generations");
        int generationLimit = int.Parse(Console.ReadLine());
        Game_of_Life lifeGrid = new Game_of_Life(grid);
         Game_of_Life.ReadFromFile();

        Console.WriteLine("Generation 0");
        sw.WriteLine("------------------");
        sw.WriteLine(("Generation " + " " + "0"));
        for (int i = 0; i < nr_of_lines; i++)
        {

            for (int j = 0; j < nr_of_cols; j++)
            {
                sw.Write(lifeGrid.generation[i, j] + " ");
            }
            sw.WriteLine();
        }

        lifeGrid.DrawGeneration();
        Console.WriteLine();
        while (lifeGrid.AliveCells() > 0)
        {

            string response;

            Console.WriteLine();
            Console.WriteLine("Generation {0}", lifeGrid.GenerationCount);
            sw.WriteLine("------------------");
            sw.WriteLine(("Generation "+" "+ lifeGrid.GenerationCount));
            lifeGrid.ProcessGeneration();
```

```
                lifeGrid.DrawGeneration();

                for (int i = 0; i <nr_of_lines ; i++)
                {

                    for (int j = 0; j <nr_of_cols ; j++)
                    {
                        sw.Write(lifeGrid.generation[i ,j]+" ");
                    }
                    sw.WriteLine();
                }
                Console.WriteLine();

                if (lifeGrid.generationCount==generationLimit)
                {
                    Console.WriteLine("Max Generaration reached");
                    Console.ReadKey();
                    sw.Close();
                    break;

                }
                if (lifeGrid.AliveCells() == 0)
                {
                    sw.Close();
                    Console.WriteLine("Every one died!");
                    Console.ReadLine();
                }
                else
                {
                    Console.WriteLine("Press <Enter> to contiune or n<Enter> to
quit.");

                    response = Console.ReadLine();

                    if (response == "n" || response == "N")
                    {
                        sw.Close();
                        break;
                    }
                }
            }
        }
    }
}
```

Explanation of functions&variables

```
        int[,] generation;
        int[,] lastGeneration;


        int generationCount;

        int width;
        int height;
```

The first variable contains the state of the current generation of cells. The second holds the state of the last generation of cells. Next is counter that contains the number of generations that have been iterated. The last two contain the width and the height of the generation.

Since the count variable is private we need a propriety to access it.

```
public int GenerationCount
 {
     get { return generationCount; }
 }
```

Next I implemented the constructor. I made it so that the user would have to send a 2-dimensional array of integers. A 1 would indicate a live cell and 0 would indicate a dead cell.

```
public Game_of_Life(int[,] newGrid)
 {
     generation = (int[,])newGrid.Clone();

     generationCount = 1;

     width = generation.GetLength(1);
     height = generation.GetLength(0);

     lastGeneration = new int[height, width];
 }
```

The constructor creates the generation by cloning the array -generation 0. It sets the counter for the first generation. Then it stores the width and height of the grid using the GetLength method of the array. Then it creates an empty grid that will be used to store the state of the last generation.

I created a Neighbors method that is passed the coordinates of the cell and count the number of neighbors the cell has.

The code creates a variable to hold the number of neighbors the cell has and then checks each neighbor, making sure that a neighbor that is outside of the grid is not verified. In order the following coordinates are v (x - 1, y - 1) (x, y - 1), (x + 1, y - 1), (x - 1, y), (x + 1, y), (x - 1, y + 1), (x, y + 1) and (x + 1, y + 1). I put in comments to show which neighbor is being verified.

```
 private int Neighbours(int x, int y)
 {
     int count = 0;

     // Check for x - 1, y - 1
     if (x > 0 && y > 0)
     {
         if (generation[y - 1, x - 1] == 1)
             count++;
     }

     // Check for x, y - 1
     if (y > 0)
     {
         if (generation[y - 1, x] == 1)
             count++;
     }
```

```
            // Check for x + 1, y - 1
            if (x < width - 1 && y > 0)
            {
                if (generation[y - 1, x + 1] == 1)
                    count++;
            }

            // Check for x - 1, y
            if (x > 0)
            {
                if (generation[y, x - 1] == 1)
                    count++;
            }

            // Check for x + 1, y
            if (x < width - 1)
            {
                if (generation[y, x + 1] == 1)
                    count++;
            }

            // Check for x - 1, y + 1
            if (x > 0 && y < height - 1)
            {
                if (generation[y + 1, x - 1] == 1)
                    count++;
            }

            // Check for x, y + 1
            if (y < height - 1)
            {
                if (generation[y + 1, x] == 1)
                    count++;
            }

            // Check for x + 1, y + 1
            if (x < width - 1 && y < height - 1)
            {
                if (generation[y + 1, x + 1] == 1)
                    count++;
            }

            return count;
        }
```

Next is function for getting a neighbors of a certain cell but it is not used in the program and was constructed for evaluation purpouse

```
public void WriteNeighbours()
        {
            for (int y = 0; y < height; y++)
            {
                for (int x = 0; x < width; x++)
                    Console.Write("{0}", Neighbours(x, y));
                Console.WriteLine();
            }
        }
```

Now we reached the core of the class, the ProcessGeneration function

```
public void ProcessGeneration()
        {
            int[,] nextGeneration = new int[height, width];

            lastGeneration = (int[,])generation.Clone();

            generationCount++;

            for (int y = 0; y < height; y++)
            {
                for (int x = 0; x < width; x++)
                {
                    if (Neighbours(x, y) < 2)
                        nextGeneration[y, x] = 0;
                    else if (generation[y, x] == 0 && Neighbours(x, y) == 3)
                        nextGeneration[y, x] = 1;
                    else if (generation[y, x] == 1 &&
                            (Neighbours(x, y) == 2 || Neighbours(x, y) == 3))
                        nextGeneration[y, x] = 1;
                    else
                        nextGeneration[y, x] = 0;
                }
            }

            generation = (int[,])nextGeneration.Clone();
        }
```

At each step in time, the following rules are checked (for every cell in current generation:

1. Any live cell with fewer than two live neighbors dies, as if caused by under-population.
2. Any live cell with two or three live neighbors lives on to the next generation.
3. Any live cell with more than three live neighbors dies, as if by over-population.
4. Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

In the draw function each cell is written row by row with a WriteLine after each row. In the AliveCells method it just adds one to the count if there is a cell that is alive and returns the value.

```
public void DrawGeneration()
        {
            for (int y = 0; y < height; y++)
            {
                for (int x = 0; x < width; x++)
                    Console.Write("{0}", generation[y, x]);
                Console.WriteLine();
            }
            Console.WriteLine();
        }

        public int AliveCells()
        {
            int count = 0;

            for (int y = 0; y < height; y++)
                for (int x = 0; x < width; x++)
                    if (generation[y, x] == 1)
                        count++;

            return count;
        }
```

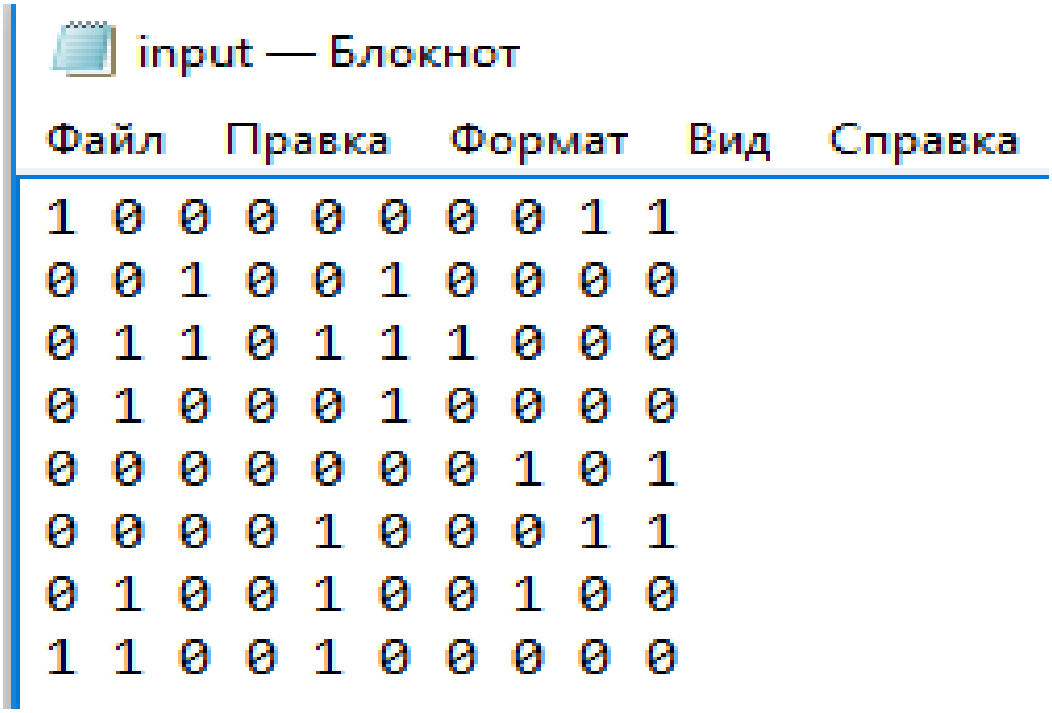
And the main method

NOTE: Change the input and output file path before compiling from Program.cs file

```
static void Main(string[] args)
        {

            string filename = @"h:\output.txt";
            int nr_of_lines = (int)Game_of_Life.ReadFromFile()[1];
            int nr_of_cols = (int)Game_of_Life.ReadFromFile()[2];
            StreamWriter sw = new StreamWriter(filename, true);
            int[,] grid = (int[,])Game_of_Life.ReadFromFile()[3];

            Console.WriteLine("Enter maximum number of Generations");
            int generationLimit = int.Parse(Console.ReadLine());
            Game_of_Life lifeGrid = new Game_of_Life(grid);
             Game_of_Life.ReadFromFile();

            Console.WriteLine("Generation 0");
            sw.WriteLine("------------------");
            sw.WriteLine(("Generation " + " " + "0"));
            for (int i = 0; i < nr_of_lines; i++)
            {

                for (int j = 0; j < nr_of_cols; j++)
                {
                    sw.Write(lifeGrid.generation[i, j] + " ");
                }
                sw.WriteLine();
            }

            lifeGrid.DrawGeneration();
            Console.WriteLine();
            while (lifeGrid.AliveCells() > 0)
            {

                string response;

                Console.WriteLine();
                Console.WriteLine("Generation {0}", lifeGrid.GenerationCount);
                sw.WriteLine("------------------");
                sw.WriteLine(("Generation "+" "+ lifeGrid.GenerationCount));
                lifeGrid.ProcessGeneration();
                lifeGrid.DrawGeneration();

                for (int i = 0; i <nr_of_lines ; i++)
                {

                    for (int j = 0; j <nr_of_cols ; j++)
                    {
                        sw.Write(lifeGrid.generation[i ,j]+" ");
                    }
                    sw.WriteLine();
                }
                Console.WriteLine();

                if (lifeGrid.generationCount==generationLimit)
                {
                    Console.WriteLine("Max Generaration reached");
                    Console.ReadKey();
                    sw.Close();
                    break;

                }
```

```
                if (lifeGrid.AliveCells() == 0)
                {
                    sw.Close();
                    Console.WriteLine("Every one died!");
                    Console.ReadLine();
                }
                else
                {
                    Console.WriteLine("Press <Enter> to contiune or n<Enter> to
quit.");

                    response = Console.ReadLine();

                    if (response == "n" || response == "N")
                    {
                        sw.Close();
                        break;
                    }
                }
            }
        }
```

It uses ReadFromFile function to read the generation 0 it's width, and length (nr of rows and columns from file as well as the first generation grid) after that it asks for user to type the maximum number of generations to display and write to output file then it displays generation 1 and writes it to file and waits for the user to press Enter if he wants to continue but if he presses n and the Enter the application is closed also if every cell from grid dies the application closed. Also in output file you will find every outputted generation starting from the generation 0.

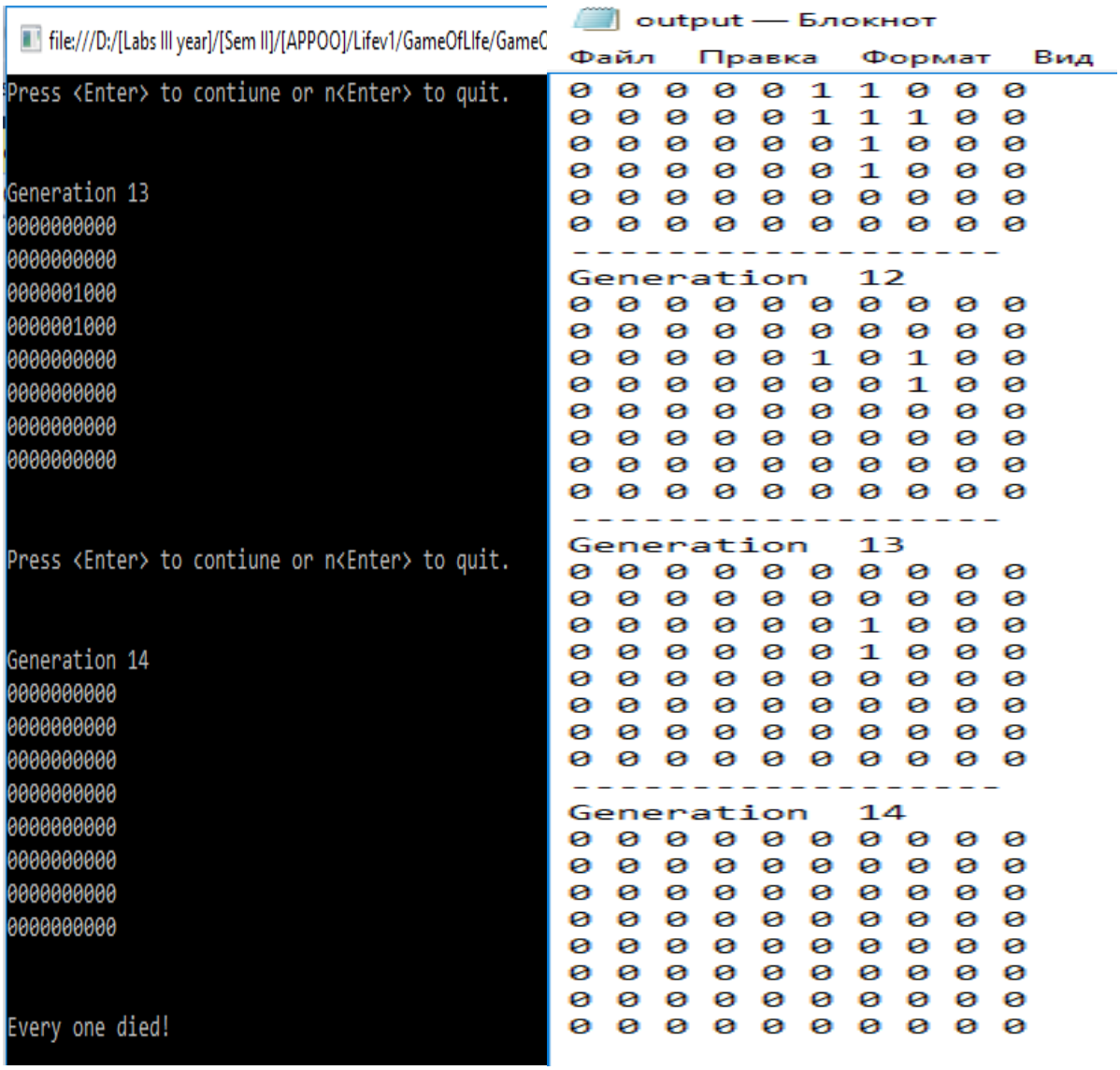


```
Enter maximum number of Generations
3
Generation 0
1000000011
0010010000
0110111000
0100010000
0000000101
0000100011
0100100100
1100100000



Generation 1
0000000000
0011111000
0111101000
0110110000
0000000001
0000000101
1101110010
1100000000


Press <Enter> to contiune or n<Enter> to quit.


Generation 2
0001110000
0100001000
0000001000
0100110000
0000000010
0000100001
1110100010
1110100000


Max Generaration reached
```

output — Блокнот

Файл Правка Формат Вид Справка

```
----------
Generation 0
1 0 0 0 0 0 0 1 1
0 0 1 0 0 1 0 0 0 0
0 1 1 0 1 1 1 0 0 0
0 1 0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 1 0 1
0 0 0 0 1 0 0 0 1 1
0 1 0 0 1 0 0 1 0 0
1 1 0 0 1 0 0 0 0 0
----------
Generation 1
0 0 0 0 0 0 0 0 0 0
0 0 1 1 1 1 1 0 0 0
0 1 1 1 1 0 1 0 0 0
0 1 1 0 1 1 0 0 0 0
0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 1 0 1
1 1 0 1 1 1 0 0 1 0
1 1 0 0 0 0 0 0 0 0
----------
Generation 2
0 0 0 1 1 1 0 0 0 0
0 1 0 0 0 0 1 0 0 0
0 0 0 0 0 0 1 0 0 0
0 1 0 0 1 1 0 0 0 0
0 0 0 0 0 0 0 0 1 0
0 0 0 0 1 0 0 0 0 1
1 1 1 0 1 0 0 0 1 0
1 1 1 0 1 0 0 0 0 0
```



Conclusion: In this laboratory work I learned how to implement a simple Conway's Game of Life console app in C#.