National University of Computer and Emerging Sciences

School of Computing Fall 2019 Islamabad Campus

Programming Fundamental

Project

(Deadline: 28th November, 2019 10:00 AM)

Submission: You are required to use Visual Studio for the assignment. Combine all your work (solution folder) in one .zip file after performing "Clean Solution". Name the .zip file as ROLL_NUM_A_01.zip (e.g. **19i-0001_Proj.zip**). Submit zip file on slate within given deadline. Failure to submit according to above format would result in deduction of 10% marks.

Comments: Comment your code properly. **Bonus marks (maximum 10%)** will be awarded to well commented code.

Deadline: Deadline to submit assignment is **28th November, 2019 01:00 AM**. Late submission without any deduction will be accepted till **28th November, 2019 01:00 AM**. No submission will be considered for grading outside slate or after **28th November, 2019 10:00 AM**. Correct and timely submission of assignment is responsibility of every student; hence no relaxation will be given to anyone.

Plagiarism: -50% marks in the assignment if any significant part of assignment is found plagiarized. A code is considered plagiarized if **more than 20%** code is not your own work.

Game of Life

In this assignment, you will implement a mathematical game known as "Game of Life" devised by John Conway in 1970. This zero-player game is set on a two dimensional array of cells (potentially infinite in dimensions). Each cell is either alive or dead. The game starts with an arbitrary pattern of cells set to live status. The rules of game are as follows.

If the cell is dead:

1. **Birth**: if exactly three of its neighbors are alive, the cell will become alive at the next step.

If the cell is already alive:

2. **Survival**: if the cell has two or three live neighbors, the cell remains alive.

Otherwise, the cell will die:

3. **Death by loneliness**: if the cell has only zero or one live neighbors, the cell will become dead at the next step.
4. **Death by overcrowding**: if the cell alive and has more than three live neighbors, the cell also dies.

A neighboring cell is any of the eight cells adjacent or diagonally adjacent to the given cell. These carefully chosen rules allow for very interesting evolution of cells starting with even simple patterns.
The life is staged on a two dimensional grid, with each cell at unique (x, y) coordinates. The top left corner has coordinates (0,0) and x increases as one moves right and y increases as one moves down.(Typical choice for graphics devices). Simplest choice for representing the cells would be to use a two dimensional integer array, where each cell contains a simple integer with value=1 for live cell and 0 for dead cell. (The cells may also contain other information like neighbor count etc). To perform updates efficiently, an additional array(unInit-array) can be used (There are many other algorithms, but this is what you will need to implement for this assignment). The unInit-array keeps track of only the

live cells. An entry in the unInit-array points to the corresponding entry in the grid and vice-versa. This reference loop serves as a way to verify that the entries are valid. The figure 1 shows the relationship between the arrays. Attributes like status of a cell, count etc are not shown in the figure, but are necessary for your program. These attributes can be kept in the unInit array rather than grid to save space. Basic operations on this unInit array are as follows.

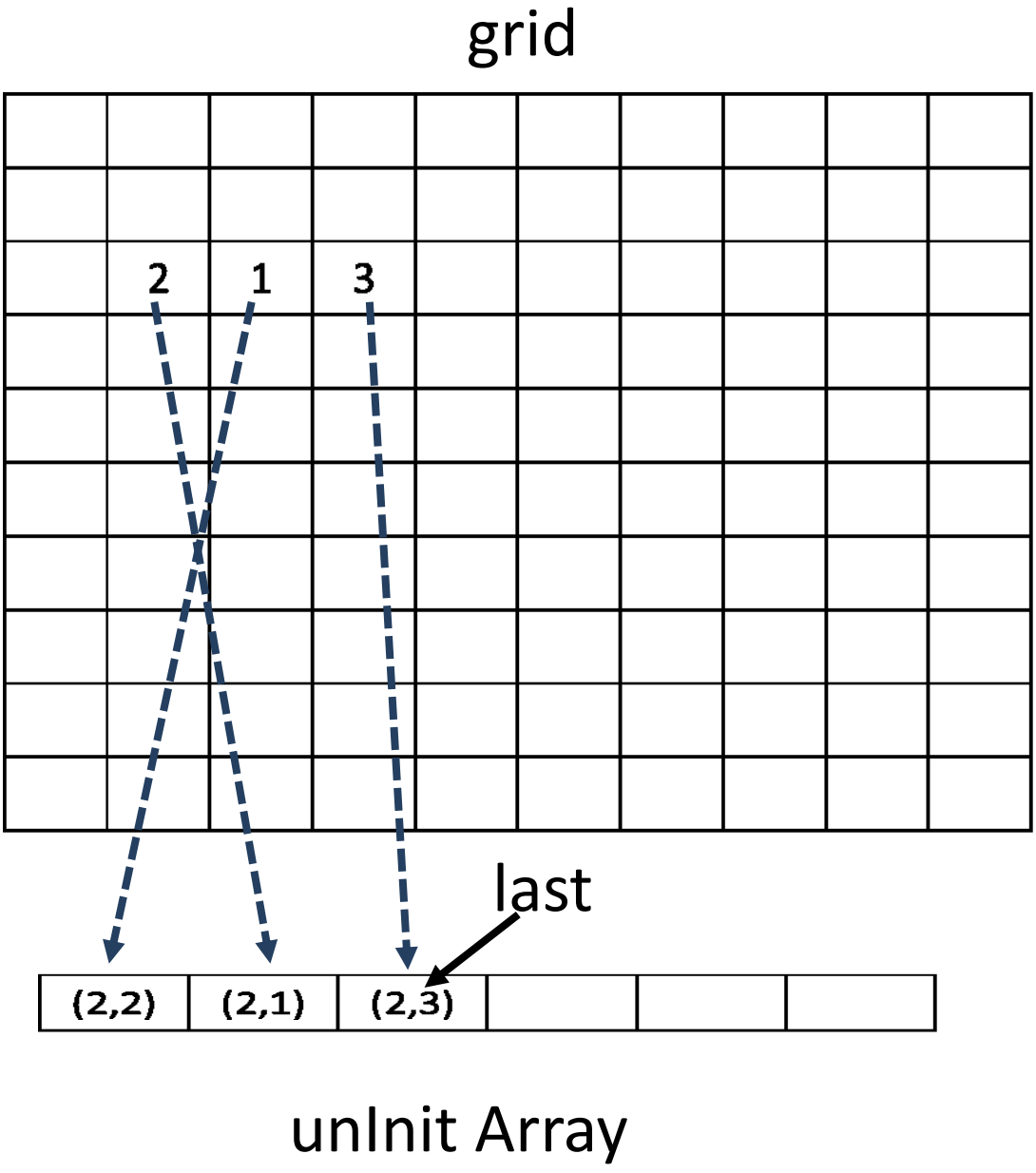

Figure 1. Representation of a blinker object

Insertion: A new element is always inserted at the end and this entry and the corresponding cell in grid are updated to refer each other.

Deletion: Just deleting an element will not work as it leaves a hole in the array. Fortunately it is simple to fill this hole by swapping in the last array element in to this just vacated entry. (We can do this because the order of entries is not important for our purposes). When the swap is done, it is important to ensure that the

reference from the grid to the swapped entry is still correct. As an example, deleting cell (2,2) from state shown in figure 1 leads to the figure 2. Note that grid entry (2,2) may still refer to its location in unInit array (index 1 in this case). However since the element at index 1 in unInit array does not point back to the deleted entry, the grid entry is invalid.

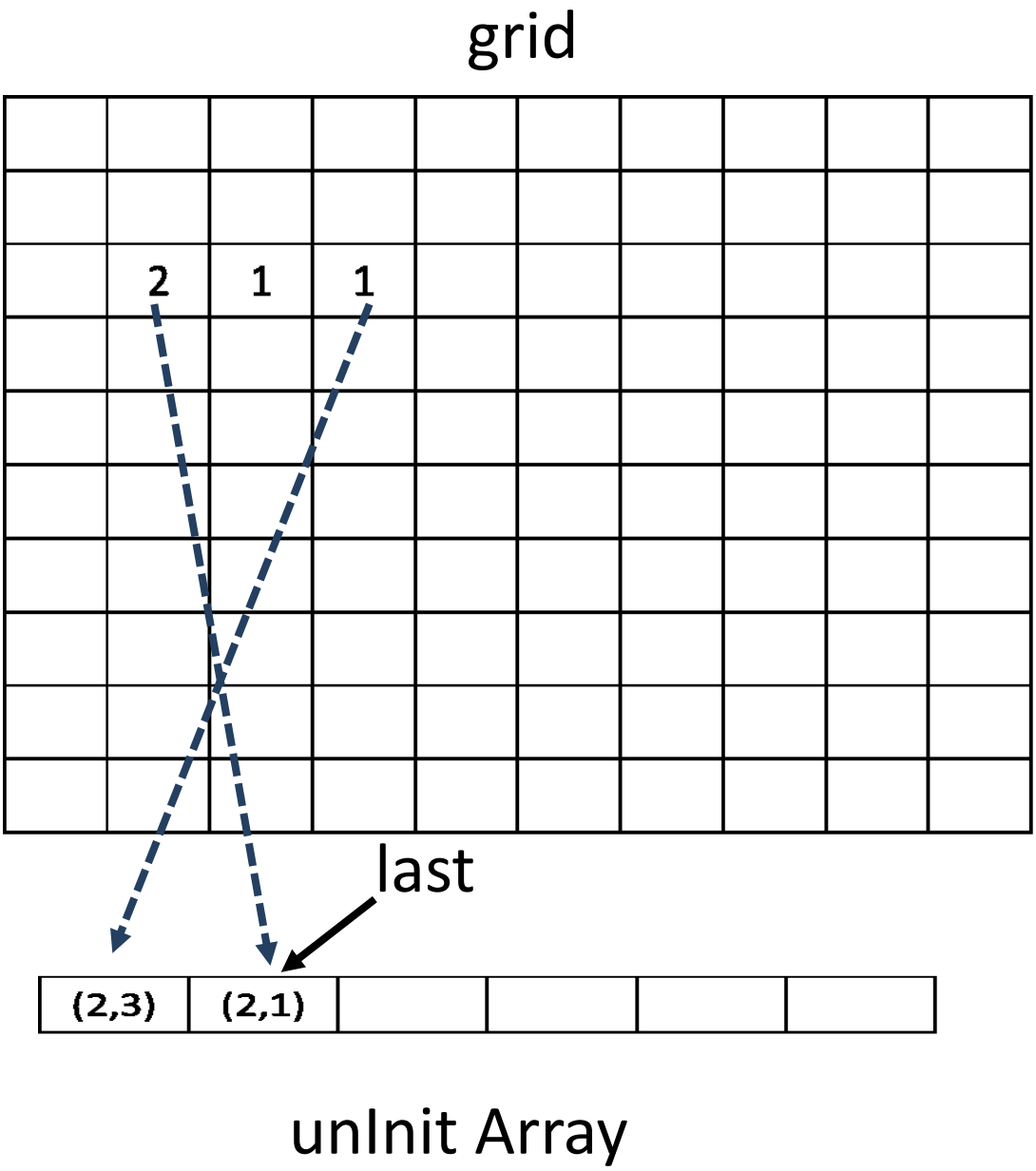

Figure 2. State after deleting cell (2,2) from unInit Array

Various phases involved in the update are as follows. These phases are primarily for descriptive purposes and you may choose to combine them for efficiency or convenience.

Adding neighbors: In this phase, the number of neighbors for each cell will be counted. For each cell, there are eight neighbors. (If the cell has coordinates (x,y), then (x-1,y-1), (x, y-1), (x+1,y-1), (x-1,y), (x+1,y), (x-1,y+1), (x, y+1), (x+1,y+1) are the neighbors.). It could be easier to add the neighbors to a different array, i.e. one array for live cells and one array for their neighbors. Since the game rules for

how alive cell propagate are different than rule for dead cells, this division of cells into different arrays can make things simpler.

Counting live neighbors: Count the number of live cells adjacent to each cell. This live neighbor count is what is needed to decide which cell will propagate to the next generation.

Cell propagation: Once the neighbor counts are known, the status of cells is determined and cells that will stay alive in the next generation are updated. (Using the rules of Life). You can perform this update within same unInit-array (by deleting and inserting as necessary) or you can add propagating cells into a temporary array and then replace the unInit-array.

Birth: if exactly three of cells neighbors are alive, the cell will become alive at the next step.

Display: The cells are drawn onto a Graphical device or to a terminal device. This phase will be kept simple so that programming can focus on the main aspects of the game.

Additional details and advice

Your program should read input from a file. The input file contains how many generations of life you need to simulate as well as the coordinates of the initial pattern. We will provide some examples for you to work with, and you should try to test with more patterns of your own. The input file format is as follows. Only the numbers will be in the file. The comments following '←' below are only explanatory.

10 ← number of generations that you will need to simulate
6 ← number of cells in the object about to be read
30 19 ← (x, y) coords for the various cells of the object
20 20
21 20
22 20
30 20
30 21

Make your algorithm as simple as possible so that you have less of a chance to introduce bugs.
We suggest the following outline:
Implement read/write procedures. Read in your creature from a file and then print it out the grid. Please do not hardcode the name of input file. You will need to process the command line argument appropriately.

Test your insert and delete procedures for the unInit array. If these work correctly, most of the work is done. Modular design will help here.

Check that you are able to correctly add the cell neighbors. Only the ones not already in the array will need to be added. While adding the neighbors, it is easy to compute the counts too. Print out the unInit array to ensure this works correctly.

Now apply the rules correctly and perform update. If you chose to update within the same unInit array, remember that you need to scan backwards as delete procedure moves element at the last to the deleted slot.

Test your procedure on some small creature. The blinker object is made of just 3 cells and is very useful for testing. Test it on some larger creature that we will provide.

As far as printing/displaying is concerned, print a 30x30 board. If a creature wonders off then ignore it. All programs should give a half a page outline of the algorithm you used. Please do start early and discuss any doubt/issues in advance with the Instructor and Professor.

What to submit
Submit your code for game of life. The code documentation should give a half page outline of the algorithm used. Note in particular how the data structure you used helps you in improving run time. The code needs to be well documented so that grader can get a good idea of what each of your procedures do.

Program modularity, clarity, documentation etc along with correct implementation will be considered for grading.

Good Luck!