DT555B Programmering i C

Lab 4 - Task 1 Game of Life

Zandra Hedlund

2023/12 /13

Table of Contents

1. Introduction

The task I chose to work on in this lab is divided into two parts. The first part is about reading and understanding code. The second part is completing the code to make a functioning program.

In software development, teamwork is very common and you often work in a codebase together with other developers. The ability to comprehend and interpret software code is foundational for developers. Code reading involves systematically understanding a program's source code to grasp its functionality. This skill is vital for maintaining and modifying codebases effectively. Additionally, studying exemplary programs through code reading is a potent method for improving programming skills. This task is an opportunity to work on these skills.

Introduction to Conway's Game of Life

The Game of Life, devised by John Conway in 1970, is a zero-player "game," simulating cellular evolution. This game unfolds within a two-dimensional world, comprising individual "cells." At each "generation," these cells exist in one of two states: "dead" or "alive." The game operates based on a defined set of rules governing the evolution of these cells from one generation to the next.

These rules dictate a cell's state in the next generation as a product of its neighboring cells' states in the current generation. Within a two-dimensional framework, a cell's neighbors are those 8 cells vertically, horizontally, or diagonally adjacent to that cell. Conway's rules can be summarized as follows:

1. Any live cell with fewer than two live neighbors dies, as if caused by underpopulation.
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 overcrowding.
4. Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

Task 1-1 Code Reading

Question 1: How is the world represented and defined in the solution?

The world in this program is represented as a grid of cells, each cell being either alive or dead. The world is a two-dimensional array defined in the file lifegame.c

- The world is represented by two static variables: world and nextstates
- world represents the current state of the cells
- nextstates represents the state of cells in the next generation

The size of the world is defined in lifegame.c, with constants WORLDWIDTH and WORLDHEIGHT, and their values can only be accessed from other files by calling the functions get_world_width() and get_world_height().

Each cell can either be DEAD (0) or ALIVE (1). These constants are defined in the header file, meaning they are global and can be reached by all functions in the program, regardless of which file they are in. To draw the world, constants CHAR_ALIVE '*' and CHAR_DEAD ' ' is used, where * symbolizes an alive cell and an empty cell symbolizes a dead cell. These two are defined in lifegame.c.

Functions are provided to initialize the world, get cell states, set cell states, finalize the evolution and output the world's current state. Also, there are functions yet to be implemented for setting the cell's state in the next generation, calculating the cell's state in the next generation and the number of alive neighbors to each cell. There are also prototypes for reading and writing the world's state to and from a file.

Question 2: Top-Down Design Diagram

The top-down design diagram involves understanding how functions relate to one another and their hierarchical structure, starting from the main() function.

```
main()
|
├── initialize_world()
|
├── next_generation()
|    ├── get_next_state()
|    └── num_neighbors()
|     └── get_cell_state()
|
├── finalize_evolution()
|    └── update world[x][y] = nextstates[x][y]
|    └── reset nextstates[x][y] = DEAD
|
└── output_world()
```

This hierarchical representation demonstrates the flow of function calls and their relationships within the program.

1. **main()**
    - Controls program flow, initializes the world, and manages generation evolution.
    - Calls initialize_world() to set up the initial state of the world.
    - Runs a loop for the number of generations, calling next_generation() in each iteration.
    - Calls output_world() after each generation to display the current state of the world.
2. **initialize_world()**
    - Initializes the world to a predefined pattern (glider in this case).
    - Sets up the initial state of the world with cells initialized to DEAD or ALIVE.
3. **next_generation()**
    - Computes the next generation of the world based on Conway's Game of Life rules.
    - Calls get_next_state() and num_neighbors() to determine the states and counts of neighbors for cells.

4. **get_next_state(int x, int y)**
    - Computes the state of a cell at coordinates (x, y) in the next generation using the rules of Conway's Game of Life.
    - Uses num_neighbors(x, y) to calculate the number of alive neighbors.
5. **num_neighbors(int x, int y)**
    - Calculates the number of alive neighbors for a specific cell at coordinates (x, y).
    - Calls get_cell_state(x, y) to retrieve the state of neighboring cells.
6. **get_cell_state(int x, int y)**
    - Retrieves the state (DEAD or ALIVE) of a cell at specified coordinates (x, y).
7. **finalize_evolution()**
    - Updates the world state to the next generation and resets all next generation states to DEAD.
    - Updates the current world with the next generation's state.
    - Resets the next generation states to DEAD after updating the world.
8. **output_world()**
    - Outputs the current state of the world to the console.

Question 3: Critical reflection on this design of the solution

The functions implemented in lab4-task1.c are not prototyped in the headerfile, meaning that to call them from a function in another file, they need to be prototyped inside that file. The same goes for defining constants and including libraries. It’s more practical to keep all library inclusions, definitions of constants and function prototypes within the headerfile.

If WORLDWIDTH and WORLDHEIGHT were defined in the headerfile instead of in lifegame.c, the functions get_world_width() and get_world_height() wouldn’t be needed and the values could be accessed directly using the constants instead.

The same way, the static global variables world and nextstates, could have been declared global instead of static, giving them the same functionality, but being reachable by all functions within the program.

In lab4task1.c, the order of the functions makes it necessary to prototype them at the top of the file. I would place the main at the bottom of the file and the functions above it in the following order:

1. get_next_state() - calls num_neighbors()
2. num_neighbors() - calls get_next_state()
3. next_generation() - calls get_next_state()
4. main() - calls next_generation()

Since get_next_state() and num_neighbors() call each other, one of them would have to be prototyped at the top of the file, since they can't both be below the other. If they were all prototyped in the headerfile, the functions could be written in any order in the c-files, but without headerfile, the function prototype of the function being called, needs to be above the function calling it.

It would also help to have a Makefile to compile the files together. For bigger projects, that would be necessary.

Task 1-2 Implementing the Evolution

Question 4: What does the specifier "static" mean? Why is it used?

The static keyword in C has several implications when applied to variables and functions:

Static variables

1. **File Scope:** When a variable is declared as static at the global scope (outside of any function), it limits the visibility of that variable to the file where it is declared.
2. **Lifetime:** static variables have static storage duration, meaning they retain their values throughout the entire execution of the program.

In lifegame.c:

- static int world[WORLDWIDTH][WORLDHEIGHT]
- static int nextstates[WORLDWIDTH][WORLDHEIGHT]

These two-dimensional arrays are declared outside any of the functions, which makes them global variables, but when variables are declared as global within a file by using the static keyword, they are limited in scope to that particular file. They are visible and accessible only within the lifegame.c file, allowing other functions within the same file to manipulate them, but preventing access from functions defined in other files.

When a global variable is initialized, its value persists throughout the execution of the program until the program terminates or until explicitly modified within the code. When a variable is declared as static, its value persists across function calls.

As the arrays retain their values between function calls, it allows for tracking the state of the world across different function executions. This is crucial in implementing the Game of Life's rules, as the state of cells in one generation affects the next.

Static functions

- **Function Scope:** When a function is declared as static, its visibility is limited to the file in which it is declared. It cannot be accessed or used by functions in other files. It's common practice to declare specific helper functions static. Static functions don't need to be prototyped in the header file. None of the provided functions are declared as static.

Question 5: Why is the two-dimentional array nextstates[ ][ ] required?

The two-dimensional array nextstates[ ][ ] serves a crucial purpose,a separate space for the computation and storage of the next generation's state. This array is essential because it helps maintain the state transition without altering the current state of the world during the computation of the next generation. nextstates[ ][ ] acts as a buffer or temporary storage for computing the next generation based on the current state of the world (world[ ][ ]). It holds the calculated states of cells for the next generation without modifying the current state.

Using a separate array for the next generation allows the current generation world[ ][ ] to remain consistent throughout the calculations. Once all cells' next states have been computed, this information can then be used to update the current generation as a whole.

Question 6: What symbolic constants are defined in lifegame.h? Why do we define these symbolic constants?

In the lifegame.h header file, two symbolic constants are defined. These constants represent the states of cells within the world.

```
#define DEAD 0
#define ALIVE 1
```

Purpose of Defining Symbolic Constants

Symbolic constants like DEAD and ALIVE are used to improve code readability, maintainability, and to provide a meaningful representation of states within the program. By assigning these symbolic constants to the actual values used to represent cell states (0 for dead and 1 for alive), the code becomes more understandable and easier to maintain. If the values for some reason needed to be updated, they only need to be changed in one place.
Using these symbolic constants instead of direct numeric values throughout the codebase helps by making the code more self-explanatory compared to using numerical values directly in the code. This can be particularly helpful when other developers read or work with the code, reducing the chances of misunderstanding or unintentional misuse of numeric values.

In summary, defining symbolic constants like DEAD and ALIVE in the lifegame.h header file provides a clear and descriptive representation of cell states within the Game of Life, enhancing code readability and maintainability.

2. Design

get_next_state()

The get_next_state() function is designed to determine the state of a specific cell in the next generation based on the rules of Conway's Game of Life. This function takes the coordinates (x, y) of the cell as input and calculates its state in the next generation according to the specified rules.

Algorithm and pseudocode

1. **Neighbor Calculation**
    - alive_neighbors variable is assigned the count of live neighbors for the cell at position (x, y). This count is determined using the num_neighbors(x, y) function.
2. **Determining Next State**
    - **If the current cell is alive (get_cell_state(x, y) == ALIVE)**
        - If the count of alive_neighbors is less than 2 or greater than 3, the cell will be dead in the next generation (return DEAD). This simulates underpopulation or overcrowding.
        - If the count is 2 or 3, the cell remains alive in the next generation (return ALIVE).
    - **If the current cell is dead**
        - If the count of alive_neighbors is exactly 3, the cell becomes alive in the next generation (return ALIVE). This simulates reproduction.
        - Otherwise, the cell remains dead in the next generation (return DEAD).

int get_next_state(x, y)

```
alive_neighbors = num_neighbors(x, y)
IF get_cell_state(x, y) == ALIVE
        IF alive_neighbors < 2 OR alive_neighbors > 3
                Return DEAD
        ELSE
                Return ALIVE
        END IF
ELSE
        IF alive_neighbors == 3
                Return ALIVE
        ELSE
                Return DEAD
        END IF
ENDIF
END FUNCTION
```

num_neighbors()

The num_neighbors function aims to calculate the number of live (or "alive") neighboring cells surrounding a specific cell located at coordinates (x, y). It iterates through the neighboring cells within the vicinity of the given cell and tallies the count of live cells based on their states.

Algorithm and pseudocode

1. **Initialization**
    - Initialize variables: count to track the number of live neighbors, and iterators i and j.
2. **Iterating through Neighbors**
    - Use nested loops to iterate through a 3x3 grid surrounding the specified cell (x, y).
    - Skip the center cell (current cell) by employing a continue statement.
    - Calculate the coordinates of each neighboring cell (neighbor_x and neighbor_y) relative to the current cell.
3. **Checking Validity and Counting Live Neighbors**
    - Verify if the calculated neighbor coordinates fall within the boundaries of the world.
    - If valid, retrieve the state of the neighboring cell using get_cell_state.
    - Increment the count variable if the neighboring cell is alive (get_cell_state returns ALIVE).

```
int num_neighbors(x, y)
count = 0
FOR i from -1 to 1
        FOR j from -1 to 1:
            IF i == 0 AND j == 0
                CONTINUE
                END IF
            neighbor_x = x + i
            neighbor_y = y + j
            IF neighbor_x >= 0 AND neighbor_x < get_world_width() AND
neighbor_y >= 0
  AND neighbor_y < get_world_height()
                count += get_cell_state(neighbor_x, neighbor_y)
              END IF
        END FOR
END FOR
Return count
END FUNCTION
```

next_generation()

The next_generation function orchestrates the process of computing the state of every cell in the subsequent generation based on the rules of Conway's Game of Life. It iterates through each cell in the world, determines its state in the next generation using get_next_state, and updates the world accordingly using set_cell_state.

Algorithm and pseudocode

1. **Iteration Through Cells**
    - Utilize nested loops to traverse every cell in the world.
    - For each cell, compute its state in the next generation using get_next_state.
    - Update the state of the cell in the next generation using set_cell_state.
2. **Finalization**
    - Upon completion of iterating through all cells, finalize the evolution by calling finalize_evolution. This step resets the world's next generation states to prepare for the subsequent iteration.

void next_generation()

```
FOR x from 0 to get_world_width()
        FOR y from 0 to get_world_height()
            next_state = get_next_state(x, y)
            set_cell_state(x, y, next_state)
    END FOR
END FOR
finalize_evolution()
END FUNCTION
```

main()

The main function orchestrates the simulation of Conway's Game of Life by initializing the world and iterating through multiple generations, computing and displaying the state of the world in each iteration. It also prompts the user to continue to the next generation by waiting for the ENTER keypress.

Algorithm and pseudocode

1. **World Initialization**
    - Invokes initialize_world() to set up the initial state of the world.
2. **Evolutions**
    - Initiates a loop that iterates NUM_GENERATIONS times.
    - In each iteration:
        - Calls next_generation() to compute the states of cells for the subsequent generation.
        - Displays the current state of the world on the console using output_world().
        - Prompts the user to continue to the next generation by printing a message and waiting for the ENTER key using getchar().

```
int main(void)
initialize_world()
FOR n from 0 to NUM_GENERATIONS
        next_generation()
        output_world()
        OUT "Press ENTER to continue..."
        getchar()
END FOR
Return 0
END PROGRAM
```

3. Implementation and Test

The implementation was done by putting together the newly implemented functions and the already implemented ones and trying to run the program. It took me a while to figure out how to access the defined constants in lifegame.c, until I realized I could use the functions get_world_width() and get_world_height(). I got some bugs from the beginning, but it was mainly typos. The instructions were clear and the function descriptions and prototypes was very helpful in getting an understanding of the problem and what needed to be done.

Since the program doesn't take any external input except pressing, the testing was performed running the program and checking that the output updated according to the expected pattern during the set number of generations. I tried pressing other keys before the Enter key, but it didn't cause any problems. The program continued to output the world as expected once the Enter key was pressed again, even when several characters was typed before it.

4. Results and discussion

The program compiles and seems to behave as expected. I think this was a fun exercise and it felt good being able to puzzle the pieces together to make a functioning simulation. It's an important skill to be able to read and understand code, as well as writing code and testing it. I think that this lab was a good way of practicing those skills. It was also interesting to learn more about Convey's Game of Life and implement it into code. It would be interesting to try

to use the same logic and apply it to hardware. For example, using led lights that switches on and off depending on the cell states.

5. References

- Linux system manual for functions puts, abort, getchar, fprintf, printf, command line argument “man <function name>”
- Static variables: https://www.geeksforgeeks.org/static-variables-in-c/
- Global variables: https://www.geeksforgeeks.org/global-variables-in-c/
- The lectures provided on Canvas
- VS Code to write and run the code