Developer Document

Nia Tatrishvili

November 27, 2022

1 Why this structure?

The program uses a combination of *two-dimentional* dynamically allocated *arrays*, *structs*, singly *linked lists* and *enums*. With these structures, the program works faster, mre efficiently and is more intuitive for the writer and for the reader of the code. The whole code is divided into small functions and the functions are made in a way that modifying the code for the future code requires less effort and thought. The used structures are:

1.1 Board

The board is a structure that consists of

- the width of the board as an integer
- the height of the board as an integer
- the content of the board as a two-dimentional array

The reasons for creating this structure are that

- a two-dimensional array was necessary to make the neighbour-counting process manageable
- a dynamic 2-dimentional array can not be treated without knowing its size at every step
- giving three parameters to every function (*width*, *height* and *a two-dimentional dynamic array*) is redundant

1.2 boardType

The BoardType enum consists of

- *ReadFromFile*
- *Randomized*
- *Error*

The reasons for creating this enumerator are that

- the user needed to choose the kind of board they wished to use
- this information was needed in multiple different functions passing this information as a number or a character might have been
  - not intuitive
  - easy to make a mistake
  - difficult for the writer to fix errors
  - difficult for the reader to understand the idea

1.3 BoardList

The BoardList structure consists of

- a pointer to a *board* - the stored data
- a pointer to another BoardList - *next*

The reasons for creating this structure is that

- the program needed to compare information of several boards to each other
- despite this information being in the files, it was uncomfortable and difficult for the computer to read and process this information in a manageable manner
- using dynamic arrays would require too much unnecessary reallocation of the memory

2 Functions

2.1 gameOfLife:

- this function takes no parameters and:
  - shows user the basic rules of the game (by calling the *displayRules* function)
  - asks user for the type size and type of the board that they would like to play with (by calling the *ChooseStartInstruction* function)
  - allocates space for the desired board
  - fills the desired board with the desired data (by calling the *fillBoard* function)
  - simulates life with the inputted board (by calling the *simulation* function)
  - takes the number of the cycles returned frm the simulation and stores it in *cycle*
  - clears the allocated space for the *board*
  - returns *cycle*

2.2 displayRules:

- this function takes no parameters and returns nothing
- it only shows user the basic rules of the game

2.3 fillBoardRandom:

- The idea of this function is to fill the board with pseudo-random dead or alive cells.
- It takes a pointer to a *board* as a parameter and has no return value
- It goes through the board grid and calls the function rand() to randomize an integer. Based on the remainder this number has on 7, it writes either empty space or @ symbol in the cell space (dead or alive respectively)

2.4 fillBoardFromFile:

- The idea of this function is to fill the board with dead or alive cells that the user inputted in the file.
- It takes a pointer to the *board* as a parameter and returns whether closing the file (where we read the information from) was successful
- It goes through the board grid and writes the next character from the board in it. It skips every character except space and @

2.5 printBoard:

- The idea of this function is to print a given board to the console
- It takes the *board* as a parameter and has no return value
- To make the board more aesthetically appealing, it draws horizontal and vertical lines around the outputted data. Then,
  - if it finds an empty space, it outputs an empty space
  - but if it finds @, it outputs a half-filled in ASCII symbol (once again, to make it more aesthetically appealing)

2.6 saveBoardInFile:

- The idea of this function is to save a given board in a given file
- this function takes parameters:
  - *board* pointer to the board
  - *fp* pointer to the file, where we want to write info (the file is already opened in "w" or "a" type, and this does not concern our function;
- and it returns whether the file closed successfully
- The function goes through the board elements and prints each character into the file. it skips line after every row is finished.
- After this is done, the program closes the file

2.7 cellNextState:

- The idea of this function is to calculate the next state of the cell based on its and its neighbours' states
- it takes parameters:
  - *current* (the current character)
  - *neighbourCount* (the number of alive neighbours)
- and it returns the character that should be stored in this cell for the next iteration of the board
- The function returns @ if
  - the current cell is @ and the *neighbourCount* is 2 or 3,
  - there are exactly 3 neighbors (no matter if current cell is alive or not)
- returns empty space character otherwise

2.8 nextBoard:

- The idea of this function is to calculate the next state of the whole *board* and write it back to the given *board*
- It takes the current board as a parameter and has no return value
- The function creates a new board called *temp*,
- allocates memory for it
- goes through all cases where a neighbouring cell is alive and counts them with *neighbourCount*
- gives the cellstate and *neighbourCount* to the *CellNextState* function and writes the result in the corresponding cell in *temp*
- copies *temp* to the initial *board*

2.9 aliveCellCount:

- The idea of this function is to count the population of current board
- it takes the current board as a parameter and returns it population
- the function iterates through the whole board row by row and adds 1 to the *aliveCellsCount* every time it comes across and alive cell
- then returns this number

2.10 boardsAreEqual:

- The idea of this function is to compare two boards and tell us whether they are the same board or not
- it takes pointers to two boards *board1* and *board2* as parameters
- it returns
  - 1 if the boards are equal
  - 0 if the boards are not the same
- the function first checks if the sizes of the boards are the same
- then checks if their populations are the same
- if these are not a problem, it iterates through the whole board row by row and checks if both boards have a same symbol in the same cell
- if it comes across a difference, it returns 0 and exists the function

2.11 chooseStartInstruction:

- The idea of this function is to ask the user for instructions and give these instrucitons back to GameOfLife
- It takes the initial empty board as a parameter
- And it returns the type of the board that the user wants to have
- the function
  - asks the user to enter a valid width of the board and does not move on until it is given one
  - asks the user to enter a valid height and does not move on until it is given one
  - stores the height and width in the *board*
  - asks the user whether they will input the board from the file, or want it to be randomized
  - returns corresponding enum element

2.12 fillBoard:

- The idea of this function is to fill the board with the cells according to user's input
- It takes two parameters
  - a *board* with empty space allocated
  - an *instruction* that tells us the type of the board
- And it returns
  - 1 if reading from the file was successful
  - 0 if reading from the file was unsuccessful
- if the instruction is to

2.12.1 *ReadFromFile*

the program opens the input board in read mode and calls *fillBoardFromFile* function with parameters *board* and *fp* (our file)

2.12.2 *Randomized*

the program randomizes the world, opens the input board in write mode and calls *saveBoardInFile* to write *board* in input_board then closes the file

2.13 addBoardToList:

- The idea of this function is to add a given board to the given list of boards
- It takes two parameters
  - a *newBoard* to be added to the list
  - a pointer to the *head* of the BoardList
- And has no return value
- The function
  - first creates a node of struct BoardList and allocates memory for it and all its components
  - then stores the given *newBoard* in it and sets its *next* to *NULL*
  - then traverses through the list to get to the end of it
  - checks if we are in the beginning of the list and if we
    - are not, it adds the *newBoard* to the existing list
    - are, it puts the data that our node is pointing to inside the place where the data that the given head points to

2.14 freeBoardList:

- The idea of this function is to clear all the memory that was allocated for the list of boards
- The function
  - creates a temporary pointer *temp* to go through the list
  - stores the *next* BoardList element into a new pointer
  - frees every space that was allocated for the BoardList element
  - moves temp to the *next* one
  - repeats the loop until we come to a NULL element

2.15 simulation:

- The idea of this function is to simulate the game of life given a start position of the board
- It takes the initial *board* and returns the size of the cycle (if it exists)
- The function
  - creates a BoardList called *head* to later store the boards
  - allocates space for the *head*
  - opens the archive.txt and the cycle.txt with w mode to clear all the data from old programs
  - counts the alive cells in the current board
  - and iterates maximum 100 iterations:
    - adds our current board to the list
    - checks if the current board is equal to any of the boards from the history
    - prints the current board
    - adds it t the archive

    - the population of current board is more or equal to the maximum recorded population, adds the current board in the archive of maximum populations
    - prints the population
    - checks for the cycle and if finds it
      - clears the memory from all the unnecessary garbage
      - exits the function, returning the length of the cycle
    - waits for one second
    - calls *nextBoard* function to change the board for the next iteration
    - counts the new number of alive cells
    - increments the counter of the loop

  this process continues until there is a loop, or until 100 iterations are made
  - then, the program clears the memory from the garbage and returns zero

2.16 printFile:

- The idea of this function is to print the whole contents of the file, but change every @ with a half-filled-cell symbol
- It takes the name of the file to read and returns whether the reading of the file was successful
- The function
  - first opens the file in read mode
  - then checks if there was a problem in this process
  - then reads the contents of the file and
    - prints if the read symbol was @
    - prints the read symbol itself otherwise
    - closes the file

2.17 showStatistics():

- This function is called after the whole simulation of the game of life has ended.
- The role of the function is to show the statistics and characteristics of the game that the user has just played.
- The function takes the length of the cycle as a parameter and has no return type
- It calls the *chooseExitInstructions* function and stores the inputted instruction in *inst* variable. Then, according to the **inst**, it calls the suitable function:
  - *showMax*: calls the *printMaxBoard* function
  - *showCycle*: calls the *printCycle()* function
  - *ExitGame*: exits back to *main*
  - *ErrorInstruction*: calls the *invalidInstructionError* function
- the loop will iterate until the user chooses *ExitGame* instruction. Then, we go back to main.