

**DT555B Programming in C**

* **Lab 4**

Samuel Jeffman

2024-12-09

[1. Introduction 1](#_Toc184829075)

[2. Design 2](#_Toc184829076)

[Grade 5 Task 3](#_Toc184829077)

[Description 3](#_Toc184829078)

[Top-down design 3](#_Toc184829079)

[Flow chart 6](#_Toc184829080)

[Pseudo code 14](#_Toc184829081)

[Test cases 15](#_Toc184829082)

[3. Implementation and Test 18](#_Toc184829083)

[Grade 5 Task 1 18](#_Toc184829084)

[Description 18](#_Toc184829085)

[4. Results and discussion 21](#_Toc184829086)

[5. References 22](#_Toc184829087)

# 1. Introduction

The objectives of this lab were to gain knowledge on how to design more advanced applications using file i/o, arrays and pointers. I chose Task 3 grade 5, advent of code, since it seemed like it had interesting problems to solve and, in the end, I was not disappointed.

# 2. Design

One major change I decided on when designing my algorithms was to move away from AlgoBuild and try to find something a bit more modern and less locked down. You can for instance apparently only edit an AlgoBuild project on a specific computer. Meaning if I start building a project on my desktop computer I can’t continue editing the same project on my laptop computer.

Anyways, I found Flowgorithm. A very similar tool like AlgoBuild but, with a bit more modern user interface and it also supports file i/o which turned out to be very useful in this lab.

## Grade 5 Task

### Description

As usual I started by trying to understand the problem. How all the elves can move is a little bit tricky and if you get something wrong here you can get easily lost and spend hours debugging something you don’t really have to. But the simplified rules are as follows:

1. Check 360-degrees for neighbours.
2. No neighbours, no move.
3. No neighbours in a specific direction, in a specific “order”, suggest move.
4. Check for collisions, not allowed to move if another elf wants to move there.
5. Perform moves.
6. Shuffle “order”.
7. Repeat

I did trip up on a specific part of the rules, but luckily, I caught it quite early, so it didn’t cause too much trouble. I initially didn’t understand that the elves are permitted to move, at least in theory, in any direction endlessly. This exposes a design flaw in my implementation as the grid in my case has a finite size. I managed to, however, work around this by adding a factor to the grid size which, due to time constraints, isn’t exposed at runtime.

### Top-down design

1. Start application
2. Load initial state
   1. Read from file
      1. Read all characters on all lines from "input.txt"
      2. Store as 2-d char array, "grid". Eg. grid[30][12] = '#'
   2. Init 3-d int array, "proposedMove", 2-d array of same size as 2.1.2, where each element is an array containing 2 elements and will hold the parent objects proposed move. Eg. ary[30][12][0] = 29, ary[30][12][1] = 12 for a move “North”
   3. Init 2-d int array, "numOfProposedMoves", of same size as 2.1.2, for storing if its ok to move to this position. Eg. ary[30][11] == 1, if its ok to move. More than 1, not ok to move
   4. Set inital direction order
      1. Save in array, "moveOrder" [N,S,W,E]
   5. Set search order from current position
      1. Save as 2-d int array, x and y.
         1. NW:[-1,-1],N:[-1,0],NE:[-1,1],E:[0,1],SE:[1,1],S:[1,0],SW:[1,-1],W:[0,-1]
   6. Init int, "numOfRounds"
3. User input number of rounds
   1. Ask how many rounds to rounds
      1. If no input or negative input
         1. Store as -1 in int variable, "numOfRounds"
      2. If correct positive input
         1. Store in int variable, "numOfRounds"
4. 3.9 Start new round
   1. 3.9.1 Init int, "numOfMoves"
5. Check for neighbours
   1. Iterate char array, row-by-row
      1. Iterate char array, char-by-char
         1. Check for neighbour
            1. Use 2-d int array, from 2.5.1

If no neighbour

Go to next char

If neighbour

Propose move, using step order

Save move to array 2.2

Add 1 to array 2.3

1. Check proposed moves
   1. Iterate array 2.2 row-by-row
      1. Iterate array 2.2 col-by-col
         1. Iterate array 2.2 coord-by-coord
            1. If -1

continue

* + - * 1. If >= 0 and if array 2.3 == 1, move position in array 2.1.2 to new position from array 2.2

Add 1 to "numOfMoves" 3.9.1

* 1. Reset arrays 2.2 and 2.3

1. Re-order steps
   1. Shuffle array 2.4.1
2. Start new round
   1. If no moves performed or when number of rounds reached
3. Stop
   1. When no moves performed or when number of rounds reached
      1. Calculate empty ground tiles('.') in smallest rectangle where all # fits
         1. Init ints topRow, rightCol, bottomRow, leftCol
            1. Start from [0][0] in a spiral pattern

Go through top row

If char is #, save row to topRow

Go through right edge column

If char is #, save col to rightCol

Go through bottom row in reverse

If char is #, save row to bottomRow

Go through left edge column in reverse until x=x+1

If char is #, save col to leftCol

If all four sides of recatngle has been found, move on.

* + - * 1. Print result

Init int dotCount

Iterate grid row-by-row, from topRow to bottomRow

Iterate grid char-by-char, from leftCol to rightCol

If char is .

Add 1 to dotCount

Print dotCount to screen

* + - * 1. Exit application or go to 2

### Flow chart

Initially I decided to just make simplified flowcharts in AlgoBuild. Basically, just print the steps from the top-down design, as I had thought the algorithms would be a bit too complicated to build mainly since I couldn’t read the initial state from file in AlgoBuild. But then I decided that’s no fun and not very challenging so luckily, I found Flowgorithm which also supports file i/o. So, I decided to have a go at converting the top-down design into working algorithms in Flowgorithm.

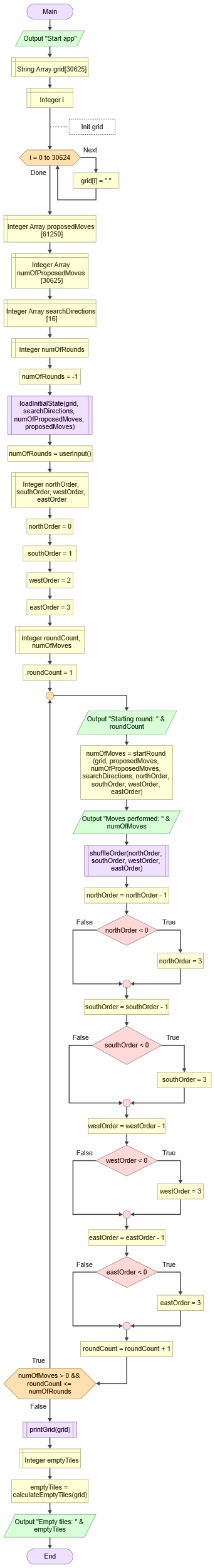
However, I soon ran into a drawback in Flowgorithm which is that it doesn’t support multi-dimensional arrays. Which of course can be worked around by flattening the arrays into 1-d. It’s just more complicated to keep track of the index. But I made it work.

I worked the algorithms step by step, building them mostly as separate functions and in the end I managed to actually solve the first part of the Advent of code challenge, using only Flowgorithm. This was maybe not the actual intended result, but it meant at least I understood the rules of the challenge and I was on the right track. I ran the Flowgorithm for about an hour to complete 10 rounds and get a correct result on the first try. But, before I got to that point, I had done extensive debugging and was confident I had ended up with a correct result.

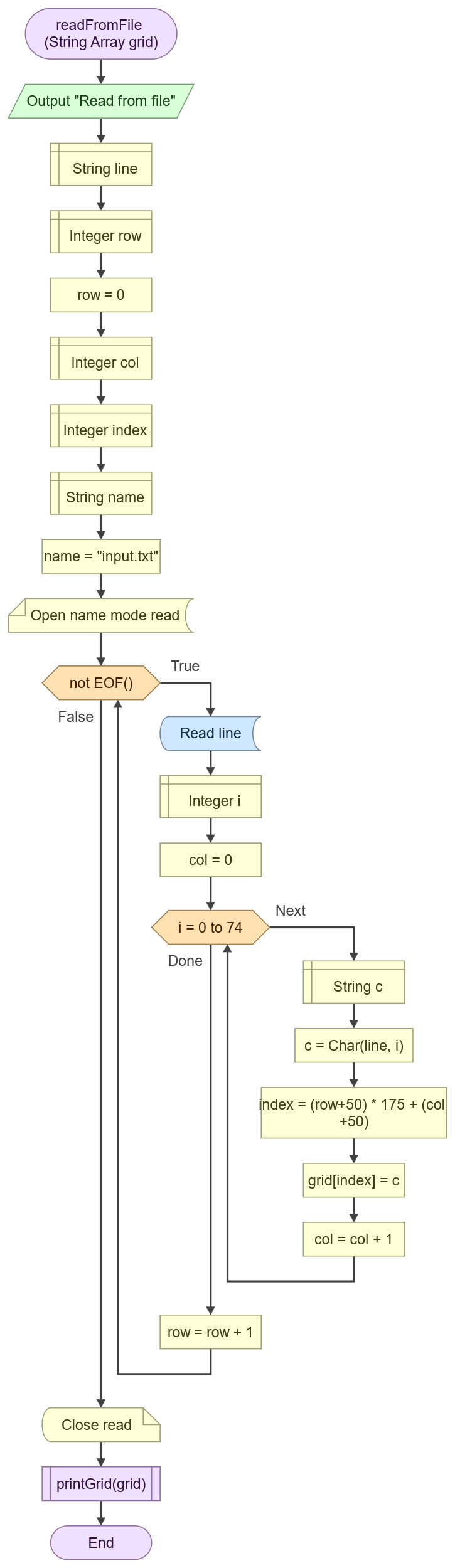
I, of course, also did the implementation in C and I got the privilege of seeing how much faster it is at computing this kind of task. As a bonus I also implemented enough to solve step 2 of the Advent of code challenge. A task which would have taken Flowgorithm roughly 100 hours to complete…

Some of the functions in the flowcharts got a little bit too big, but they were refactored into smaller functions when implemented in C.

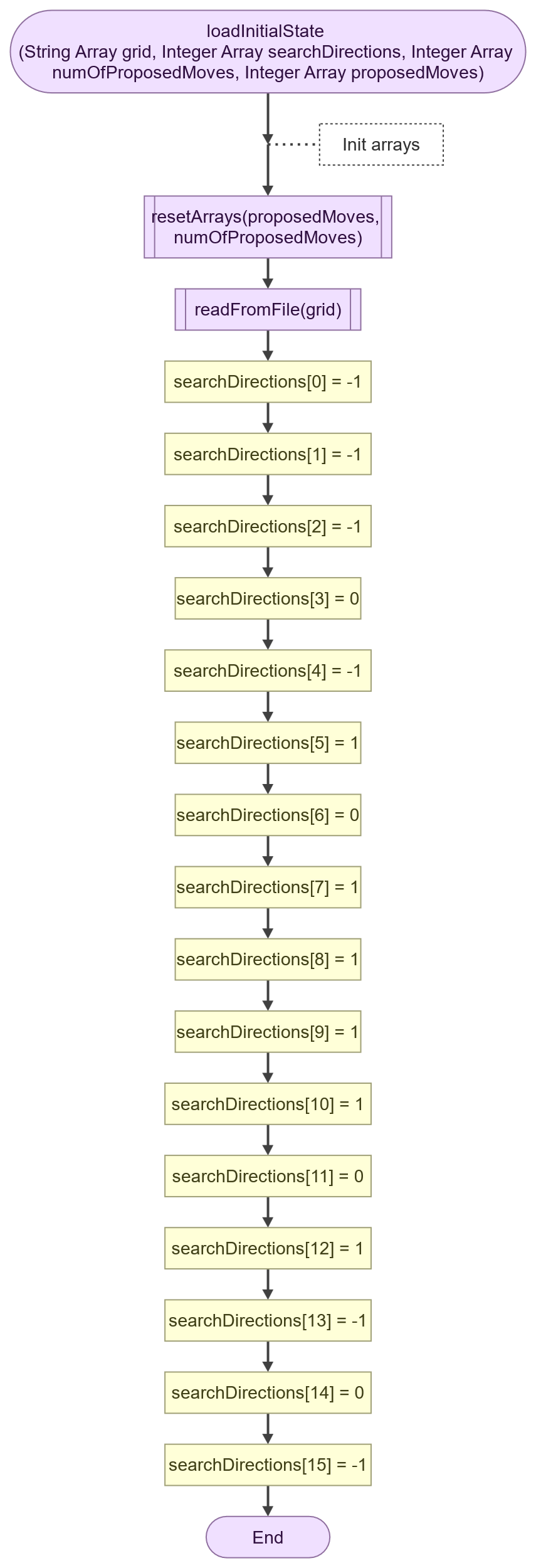
#### Main



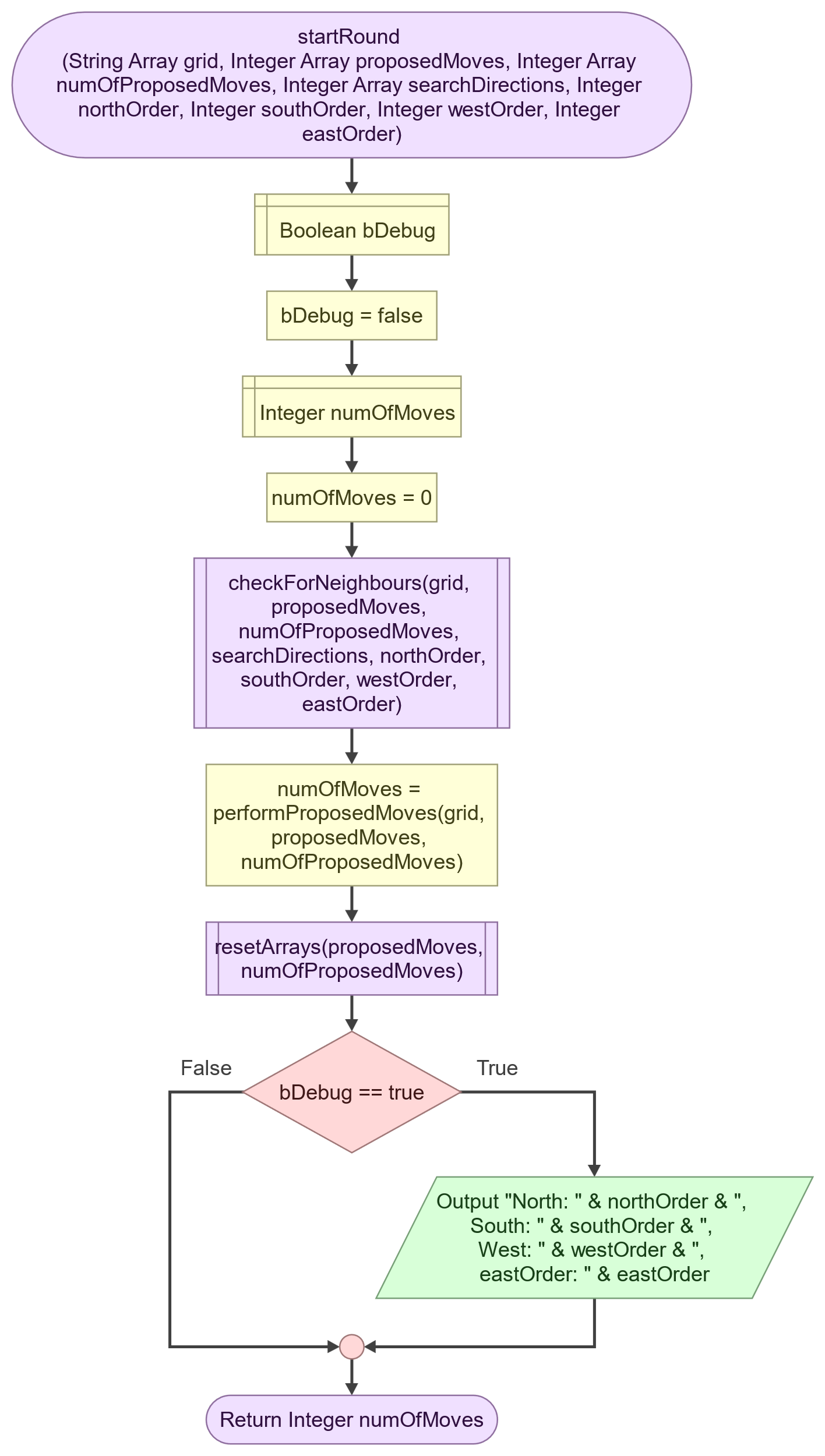
#### ReadFromFile



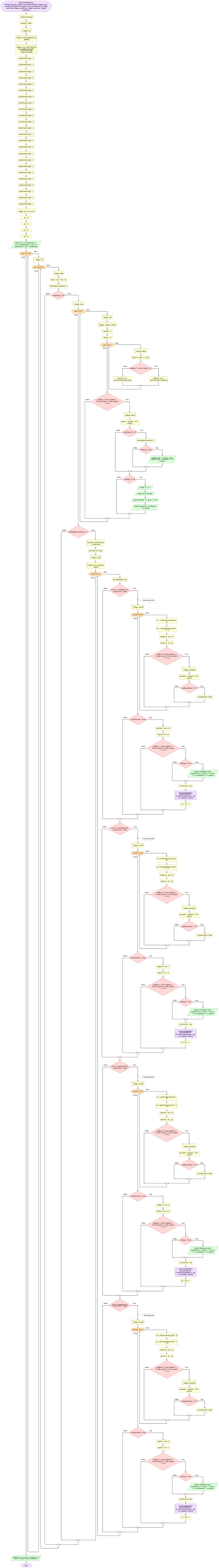
#### LoadInitialState



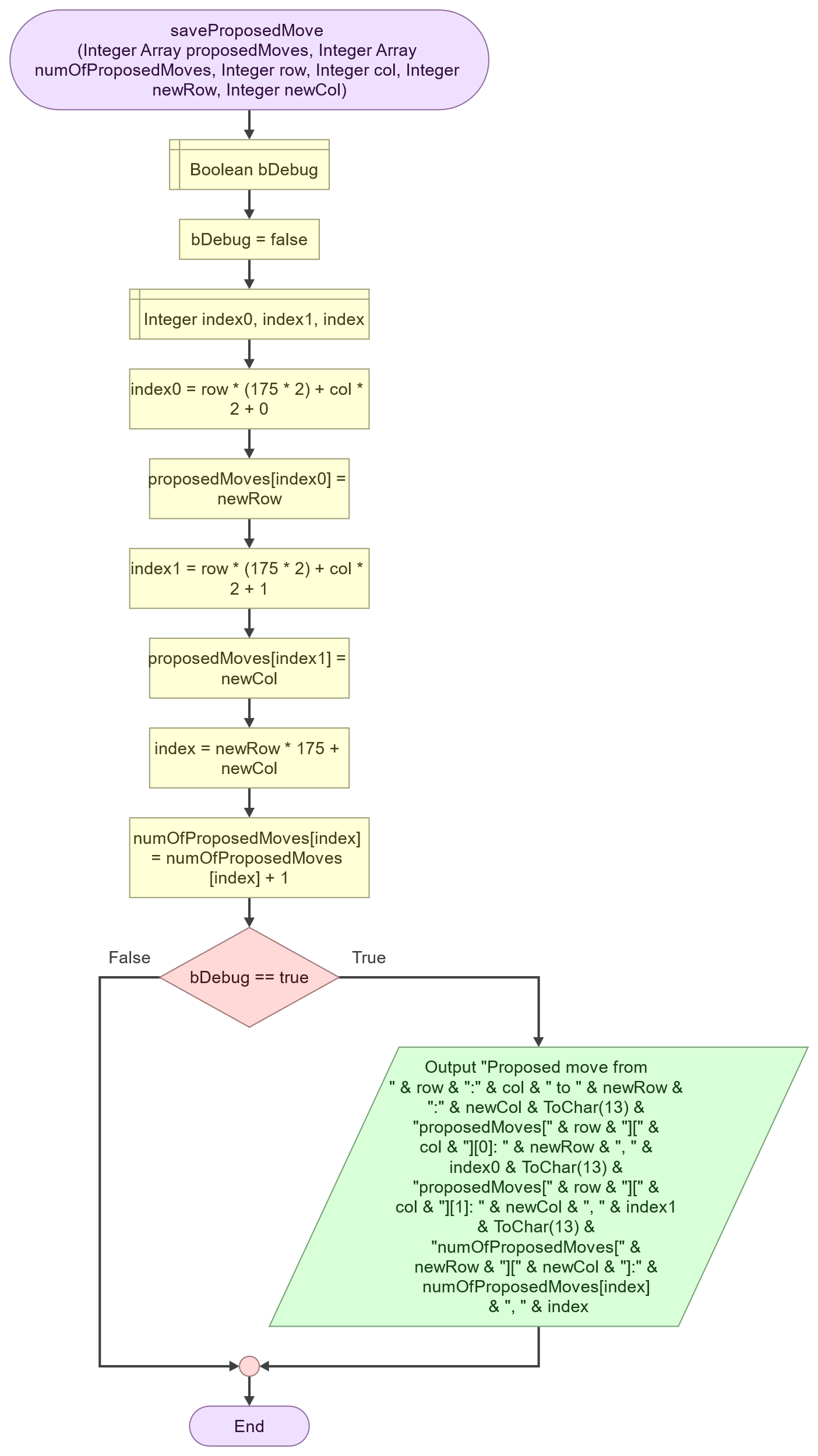
#### StartRound



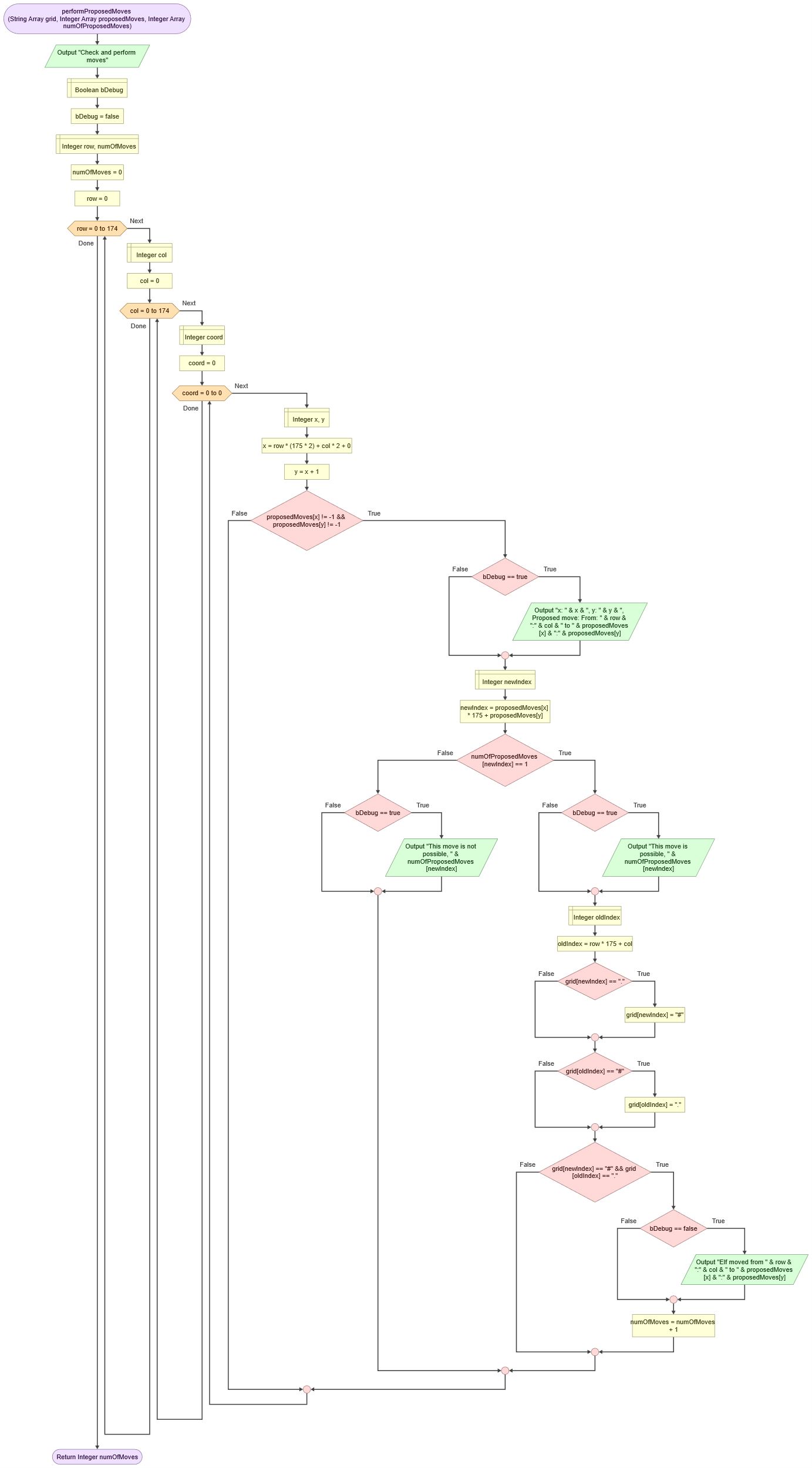
#### CheckForNeighbours



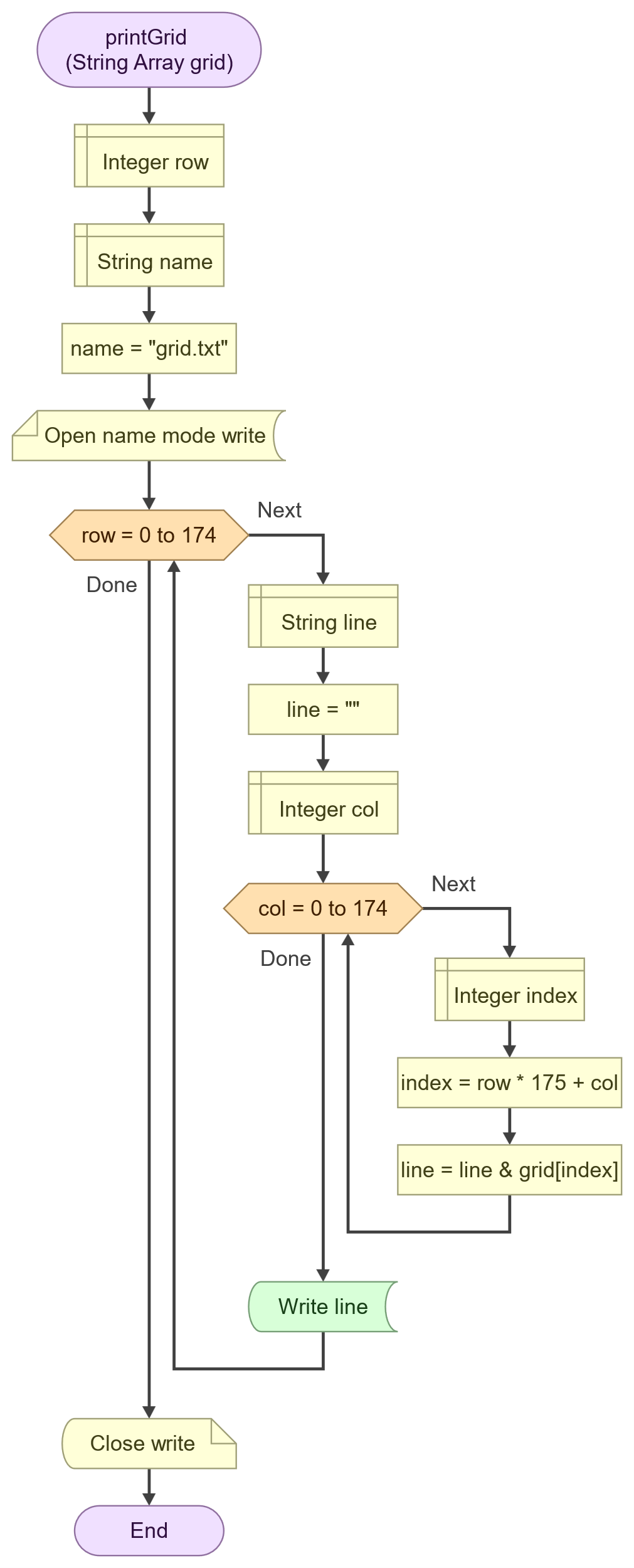
#### SaveProposedMove



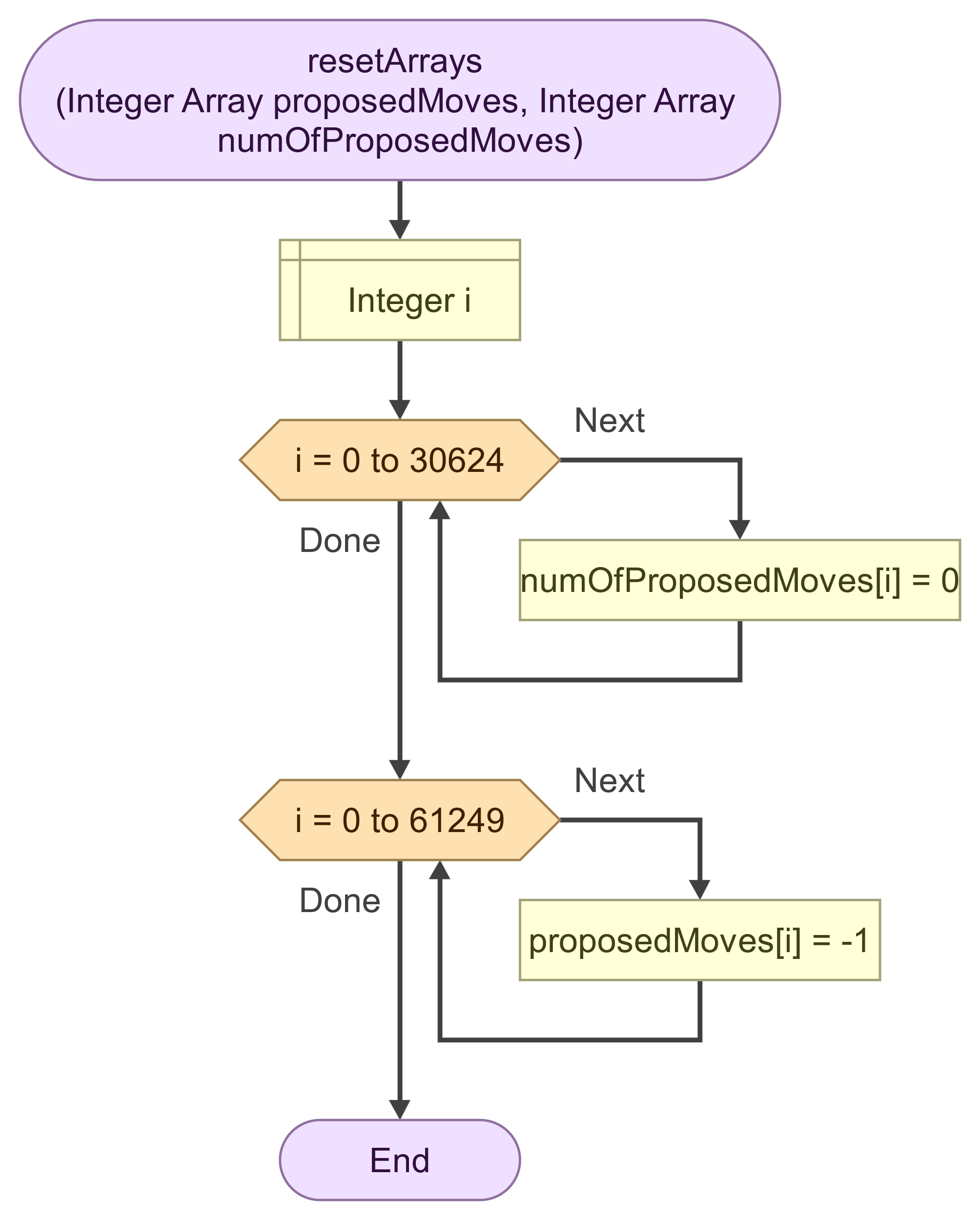
#### PerformProposedMoves



#### PrintGrid



#### ResetArrays



### Pseudo code

Flowgorithm also generates pseudo code, but I chose not to add is here as I don’t really put much emphasis on it. I am more sold on the concept of flowcharts.

### Test cases

The code is mostly tested using assertions to check that each function does what it is supposed to do. It was also very helpful in the fact that I basically knew which state the output should be in after 10 rounds. For the input I got from Advent of code I was supposed to have 4302 empty tiles inside the smallest rectangle. So basically after each change in the code I could run 10 rounds and check that I still got the same result of “4302 empty tiles”. After I completed step 2 of the challenge I also knew that it needed 1025 rounds to get to a point were no moves were made. So I could also use this a as a reference point whenever I did future changes in the code.

#### Different size input

I did tests using different size input, both small and large, and decided to make the minimum grid size 75x75. So even if your input is only one row with 10 characters, the grid size will still be 75x75. This to ensure enough space even if your input is very small, since a factor of 3 won’t give you much space to move if your input is only one row.

Also non-square and irregular shaped input was tested.

#### Special characters

Characters which aren’t # or . are accepted in the input file, but will just be ignored by the application.

#### Memory leaks.

Since I am utilizing dynamic memory allocation, I did a few tests to see that memory usage is stable after each round. The way the application is set up all the needed memory should be allocated directly on startup. Meaning memory usage between each round should ideally not vary at all. I did this test by drastically increasing the size of the input data, as much as 10 MB at one point. This has the effect of the application running much slower, since each round takes a lot more time to compute, and the application as a result runs a lot longer. I studied the memory usage of the application using ProcessExplorer and no real change was detected.

I wanted to implement some form of memory allocation counter, to keep track of allocation/deallocation. But I ran out of time and had to leave that feature on the drawing board.

#### Assertions

Most of the functions in *functions.c* were tested using assertions. Assertions were implemented in a separate file, tests.c. They are only run if the macro *DEBUG* is defined in main.c.

A few examples of the test cases used:

int numOfMoves = startRound(simData);

assert(numOfMoves == 756 && "testStartRound(): Number of moves should be 756"); //756 moves after round 1

shuffleOrder(simData);

numOfMoves = startRound(simData);

assert(numOfMoves == 581 && "testStartRound(): Number of moves should be 581"); //581 moves after round 2

int emptyTiles = checkEmptyTiles(simData);

assert(emptyTiles == 2871 && "testCheckEmptyTiles(): Empty tiles should be 2871"); // There should be 2871 empty tiles in the initial state data

# 3. Implementation and Test

## Grade 5 Task 1

### Description

As always, it’s a lot more fun when it comes time to start implementing the solution in code, when the design work is solid. You already have a clear understanding of the problem and how to solve it. However, there still were challenges to overcome, and as usual it involved pointers. This time with the added addition of dynamic memory allocation.

There was however one thing that made implementation a whole lot easier and that was the addition of the use of structs. It almost makes C feel a little bit object-oriented. I’m glad I had a look at the module on structures before I started implementing the code for this lab. It makes it so much easier to just pass a pointer to an instance of a struct as an argument to a function, instead of relying on separate variables or even worse, global variables.

Another challenge would be to work with multi-dimensional arrays and how to allocate memory for them.

And finally, how to handle file input and output.

Implementation went smoothly using the flowcharts as a template and the first version of the code was implemented closely to how the flowcharts turned out. I feel it is easier to optimize the code from there.

The short version of how the code is implemented would be application starts, input file is read, memory is allocated to match the size of the input data and the input file is stored in a 2-d array with a data type of char. Then the application asks for input from the user on how many rounds to run.

The application then starts by checking all the # cells in the grid, row-by-row and column-by-column, if there is no neighbor in any direction there is no need to do anything. If a neighbor is found in a specific direction a move is suggested according to which direction is currently first in line. The suggested move is stored in separate 3-d int array.

When the checking is done all the suggested moves are performed, but only if it can be done without two # colliding with each other.

If no moves were performed or amount of rounds has reached the limit, the application ends and prints some data to screen and the current grid is stored to file in **C:\ElfSim\output.txt**. Otherwise, a new round starts.

#### Structs

It almost feels like cheating when working with structs in C. It sort of abstracts away whatever is inside the struct and I don’t really have to consider if it’s a pointer or a pointer to another pointer or another struct. I don’t know, it might be a trick of the mind, but it sure felt like something clicked in to place when I added the struct concept in my code. It makes the code much cleaner also, because you usually only have to refer to an instance of a struct instead of a bunch of separate variables.

I collected all that is needed inside an instance of a pointer to a struct called *SimulationData*, which is a typedef struct, which in turn makes it cleaner when creating an instance of this struct. The struct contains all that is needed to run one or multiple rounds of the simulation. It stores the current state of the grid in a member called *grid.* It also stores data on suggested moves in a member called *proposedMoves.* It also stores a few other things like the size of the grid, number of moves and so on.

The initial version of SimulationData was quite large, containing many separate members. This was refactored into multiple smaller structs that where then made mambers of *SimulationData*.

typedef struct

{

char\*\* grid;

int\*\* numOfProposedMoves;

int\*\* searchDirections;

int\*\*\* proposedMoves;

GridInfo gridInfo;

Directions directions;

Orders orders;

Size fileSize;

unsigned long iterations;

} SimulationData;

#### Memory allocation and void\*

The initial version of the code looked quite inefficient and contained a lot of duplicated code, regarding memory allocation for the various arrays. A lot of work was done to create generic functions to allocate memory for the various types of arrays. With the help of void\* you can create functions which return pointers for any type. In this case I need to return pointers to arrays of both type int and char. And they also needed to be one, two and three dimensional. And I ended up with three functions which facilitate all those needs,   
*void\* allocate1DArray(int length, size\_t elementSize, void\* initialValue)*,   
*void\*\* allocate2DArray(int rows, int cols, size\_t elementSize, void\* initialValue)* and   
*void\*\*\* allocate3DArray(int rows, int cols, int depth, size\_t elementSize, void\* initialValue)*. Where the 3D function calls the other two in turn. I was really satisfied with this construct.

#### File I/O

I feel that file read and write usually can be quite tricky, since you normally have to consider quite a few scenarios where you may need to handle extra whitespace and/or special characters. In my case, to save time, I chose to trust the source and just read all the characters as they are presented in the file. And also, any characters which aren’t # or . will just be ignored by the code.

There were a few things that tripped me up. One thing being I had a hard time getting the exe file to read the file as if it was stored parallel to the exe file itself. I tried a few things that didn’t work and ended working around the issue by storing the input in a static path, **C:\ElfSim\input.txt**. The same path is also used to store the output file, output.txt, which contains the state of the simulation after the simulation ends.

Another thing I had troubles with initially was null termination, ‘\0’, which I added unnecessarily when reading the file. It caused some strange things to happen in certain areas of the grid. But luckily, I caught the issue quite early on.

# 4. Results and discussion

I was a bit skeptical I would be able to pull this off in time. But as the design work wore on, I grew more confident. And a major part of the work is, as usual, understanding the problem. “What is it we are trying to solve here?”. And a big part of it also goes out to put down proper design work and lastly the implementation and testing really to a certain extent becomes the easy part. It’s so much more fun coding when there is actual groundwork done, instead of just diving straight in to the code, getting stuck and just banging your head against the wall.

I’m properly satisfied with how this project turned out. The satisfaction I get when seeing how blazingly fast C is compared to Flowgorithm, is almost worth the ticket alone. Flowgorithm completed step 1 of the challenge in about one hour. My application reaches the same result in 0.194 seconds. Step 2 is completed in 19.44 seconds. Maybe it’s not a fair comparison, but it’s still entertaining.

So, if I take anything away from this lab it’s that design work is important and C code compute stuff mind-blowingly quickly.

En bild som visar text, skärmbild, Teckensnitt

Automatiskt genererad beskrivning

En bild som visar text, skärmbild, Teckensnitt

Automatiskt genererad beskrivning

# 5. References

-, -. (-). *lecture 6-1 Pointers and arrays [Pdf]*.Canvas.  
 https://hkr.instructure.com/courses/7307/modules/items/380354

-, -. (-). *lecture 7 Structures [Pdf]*.Canvas.  
<https://hkr.instructure.com/courses/7307/modules/items/380364>

Geeks for geeks. (2024, 11 Oct.). *Void pointers*.  
<https://www.geeksforgeeks.org/void-pointer-c-cpp/>

Geeks for geeks. (2024, 11 Oct.). *Structures in C*.  
<https://www.geeksforgeeks.org/structures-c/>

Geeks for geeks. (2024, 11 Oct.). *Typedef in C*.  
<https://www.geeksforgeeks.org/typedef-in-c/>

Geeks for geeks. (2024, 11 Oct.). *Multi-dimensional arrays in C*.  
<https://www.geeksforgeeks.org/multidimensional-arrays-in-c/>

Geeks for geeks. (2024, 11 Oct.). *Dynamic memory allocation in C*. <https://www.geeksforgeeks.org/dynamic-memory-allocation-in-c-using-malloc-calloc-free-and-realloc/>