Assignment 4: *Design*

Sriramya Prayaga

February 3, 2022

Abstract

In this document, I will be describing the design process for Assignment 4. For Assignment 4, we are essentially building John Horton Conway's Game of Life. The game is constructed using two different files: universe.c (and universe.h, the provided universe header file) and life.c. In the file universe.c, there is a universe structure and some functions that create a 2-D boolean matrix on the heap, and populate the cells of the matrix. In the file life.c, the program can take in command-line arguments, and using the matrix created from universe.c, can display the output of The Game of Life using ncurses.

1 Making the Universe

As mentioned above, there are several functions and a structure that defines a universe in the file universe.c. The structure consists of two uint32's that store the number of columns and rows in the universe, a boolean that states if the universe is toroidal, and a pointer to a pointer of the universe matrix grid. A cell can have either of two states: dead or alive, which can be represented as boolean values false or true. A toroidal universe means that the values at the start of a particular array are connected to the values at the end of that array– that is, it can be thought of as a closed loop for both the rows, columns, and diagonals of the matrix.

The following is the pseudocode/structure for these functions.

1.1 Pseudocode

```
Universe *uv_create(uint32_t rows, uint32_t cols, bool toroidal) {

        universe = an instance of the Universe structure

        initialize universe with given rows, cols, and toroidal value

        allocate space using calloc for a column of pointers to rows for
        the grid in a universe

        for each row in universe {
               fill with pointer to number of columns
        }

        initialize universe with value of grid
```

```
          return universe
}
void uv_delete(Universe *u) {
          free all pointers to columns
          free column of pointers to rows
          free instance on universe
}
uint32_t uv_rows(Universe *u) {
          return number of rows in u
}
uint32_t uv_cols(Universe *u) {
          return number of columns in u
}
void uv_live_cell(Universe *u, uint32_t r, uint32_t c) {
           if row and column of cell value (r,c) is in-bound {
                   set value of cell value to true
           }
}
void uv_dead_cell(Universe *u, uint32_t r, uint32_t c) {
           if row and column of cell (r,c) is in-bound {
                   set value of cell value to false
           }
}
bool uv_get_cell(Universe *u, uint32_t r, uint32_t c) {
           if cell (r,c) is in bounds of the universe {
                    return cell value
           }
           else {
                    return false
           }
}
bool uv_populate(Universe *u, FILE *infile) {
           while (fscanf(text file) has not reached the end of file) {
                     if cell (r,c) is in bounds of the universe {
                            call uv_live_cell(u, r, c)
                     }
                     else {
                            return false
                     }
           }
           return true
}
uint32_t uv_census(Universe *u, uint32_t r, uint32_t c) {
           count = 0;
```

```
            if (toroidal == false) {
                      for all neighboring cells (that are not toroidal) {
                             count +=1
                      }
             }
            else {
                      for all neighboring cells (that are toroidal) {
                             count +=1
                             }
            }
}
void uv_print(Universe *u, FILE *outfile) {
            for values in the grid of u {
                      if uv_get_cell == true{
                         fprintf 'o' to designated outfile
                      }
                      else {
                         fprintf '.'
                      }
            }

}
```

1.2 Some notes on the Pseudocode

The universe grid, as can be seen from the pseudocode, is referenced by a pointer to a pointer. The grid uses calloc() to initialize to a grid of all falses. There are also functions

`uv_rows() and uv_cols()`

that return the number of rows and columns in the universe, a function that sets a cell to be "alive",

`uv_live_cell()`

, a function that sets a cell to be "dead"

`uv_dead_cell()`

, and another function that returns the state of the cell

`uv_get_cell()`

. The populate function takes in an input file (can be standard input) which contains the number of rows and columns of the grid/universe on the first line, and, on the subsequent lines, contains the cells which are alive. It uses fscanf() to read the input (which would be written in the pattern of "int (space) int" on each line).

Another thing to note is the print function can print to any output file (including standard output). Depending on whether the value of the cell is false or true (dead or alive), it prints '.' or 'o'.

The delete function de-allocates and frees the heap memory that was used for making the universe. It first frees the data inside the matrix, then the matrix itself, and, finally, the universe.

2 Making Life

Now, we shall discuss the second part of the assignment: life.c. This is the file that implements the rules of the Game of Life, and outputs the result. This game is run for a certain number of user-provided "generations". To be clear, there are rules that dictate the state (alive or dead) of each cell for each generation on the matrix. The three rules for this game are: any alive cell that has two or three live neighbors lives, any dead cell that has exactly three live neighbors lives, and all other cells die in the next generation. These rules are implemented using an if-statement below in my pseudocode.

The file life.c uses getopt() to parse command-line arguments which specify how the results of an instance of The Game of Life should be displayed.

The command-line options for this program are:

- -t : specify a toroidal universe
- -s : specify ncurses to be silenced (only the output of the universe print function should be shown)
- -n generations : this specifies how many generations the program should run for
- -i input : specifies input file to read row-column pairs (default is standard input)
- -o output : specifies output file to print final results of the game (default is standard output)

The pseudocode for this main function in life.c is as shown below.

2.1 Pseudocode

```
initialize all default variables

while (command-line options are given) {
      switch(based on command line options above) {

      for case 't' and 's': store boolean variables of toroidal,
      and silence respectively

      for case 'n': store number of generations in variable

      for cases 'i' and 'o': open input/output files and store with
      pointer variables
      }
}

open given input file to read

read first line of data, store the two values into separate variables 'row' and 'column'

create two different instances (A and B) of the universe using the same 'row' and 'column',
and 'toroidal' variables
```

```
call uv_populate(A) and uv_populate(B)

if silence is false {

        initialize ncurses screen

        for until the number of generations given {
          for all cells in A {
             refresh ncurses screen
             display A
             sleep for 50000 seconds
             check the three Game of Life rules/conditions
          swap pointer of A with pointer of B

        }

        end ncurses
}
else {
for until the number of generations given {
          for all cells in A {
              check the three Game of Life rules/conditions
          swap pointer of A with pointer of B
          }
        }
}
output result of last generation by calling uv_print(A, outfile)

close input file
close output file
free A and B
```

2.2 Some notes on the Pseudocode

In my pseudocode, I often just wrote the value 'cell' when making function calls, however, the actual values that the functions accept are r and c (the row, and column of the cell.) I had written it that way for simplicity purposes.

We need to use ncurses to display the results of The Game of Life (while the appropriate functions are called from universe.c), so we could see how the game plays out visually. As can be seen from above, I initialized ncurses and also ended it. If the silence (-s) command-line option was given, then ncurses would not be initialized, and only the three rules for the game would be implemented in the for-loops. However, if the silence option was not given, then the results of the cells for each generation would be outputted using ncurses.

I also initially created two versions of the grid for the program, so as to keep track of the current and previous generations of the game of life —I used pointers to swap the particular arrays that were passed as parameters.

This program uses fscanf() to read input from a file, and calls the print function in universe.c (which uses fprintf to write the output to an extraneous file, as needed.)

Some important details to highlight here are the fact that I created two different additional functions in the life.c file other than the main function mentioned above. To be clear, I created a function to swap the pointers of the two Universes A and B and a function to implement the three rules of life for each generation. The pseudocode for my swap function is as so:

```
void swap_universe (Universe **A, Universe **B) {
        Store pointer of A in temporary variable
        set pointer A equal to B
        set pointer B equal to the temporary variable
}
```

As can be seen here, I simply swapped the two pointers by passing them to a function that takes in pointers to pointers. This idea is similar to how integers were swapped in the example of the textbook The C Programming Language by Brian Kernighan and Dennis Ritchie. The other function I created has structure similar to what is written below:

```
void life_rules(Universe *A, Universe *B, uint32_t i, uint32_t j) {
             if cell is alive and 2 or 3 neighbors of cell are alive {
                call uv_live_cell(B,cell)
             }
     else if 3 neighbors of dead cell are alive {
                call uv_live_cell(B,cell)
             }
     else {
                call uv_dead_cell(B,cell)
             }
}
```

I chose to put this code in a separate function because both branches of code in my main program that use/don't use ncurses still utilize the same rules of life, and it seemed repetitive to write it twice.

2.3 Error Checking

As I realized improper input could be fed into my life.c file, I created some error checking statements to ensure that my program would not crash with these inputs. I have checked for any row-column values that are less than zero, files that could not be opened, and generation values that are less than zero, for example. I printed an error statement to rveal these errors.

3 Summary

This assignment combines the files universe.c/h and life.c to play The Game of Life. The universe.c file contains a struct and some functions which define what a universe is, and how to populate it. The file life.c takes in command-line arguments that specify things like how many generations the game should be played for and which file has the input for the game. It also contains the actual code for the rules of the game (and can actually play it "live" using ncurses, if the option was selected), and displays the result of the last generation of the game to the desired output file.