Krisha Sharma

CSE13s

Winter 2022 Long

01/28/2022

Assignment 4 DESIGN.pdf

Description of Program:

The main task in Assignment 4 is to implement the Game of Life in C programming. The Game of Life, also known as Conway's Game of Life was developed by John Horton Conway in 1970. It is a zero-player game, meaning that its evolution is determined by its initial state, requiring no further input. The specifics of the assignment implementation are as follows:

Files to be included in directory "asgn3":

- `universe.c`
    - Implements the `Universe` ADT.
- `universe.h`
    - Specifices the interface to the `Universe` ADT.
    - This file is provided and *may not* be modified.
- `life.c`
    - Contains the `main()` and *may* contain any other functions necessary to complete the implementation of the Game of Life.
- Makefile
    - `CC = clang` must be specified
    - `CFLAGS = -Wall -Wextra -Werror -Wpedantic` must be specified

    - `make` must build the `life` executable, as it should `make all` and `make life`.
    - `make format` should format all the source code, including the header files
- README.md
    - Text file in Markdown format that describes how to build and run the program, how the program handles erroneous inputs, and any problems encountered while developing the program.
- DESIGN.pdf
    - Describes design for the program thoroughly with pseudocode and visualizations.
- WRITEUP.pdf
    - No WRITEUP is required for this assignment.

General Notes:

- Assignment 4: The Game of Life (also known as Conway's Game of Life) asks us to implement the Game of Life in C programming.
- Understanding the game:
    - The Game of Life should be played on a potentially *infinite* two-dimensional grid of cells that represents a universe.
    - Each cell has two possible states: whether dead or alive.
    - The game progresses through generations, but it can be looked at as "steps in time"
    - There are three main rules for the game:
        - Any *live* cell with two or three live neighbors *survives*.
        - Ant *dead* cell with exactly three live neighbors *becomes a live cell*.

        - All other cells die, either due to loneliness or overcrowding,
- In order to create the universe that this game is played in, an abstraction must be created
- ADT, also known as an *abstract data type*, is something that we will write that will provide the abstraction for a universe, a finite 2-D grid of cells.
    - The gird cannot be infinite because computers work in *finite memory*
    - I will need to create multiple functions that are required for my ADT
        - I need to create the constructor, destructor, accessor, and manipulator functions
        - The universe.h header file is given to us. for the Universe ADT.
- The universe will be abstracted as a `struct` called `Universe`.
    - Use a `typedef` to construct a new type and treat it as opaque, so pretend that you cannot manipulate it directly.
- `Universe.h` declares the new type and `universe.c` defines its concrete implementation
- The `Universe` must contain the following fields: `rows`, `cols`, and a 2-D boolean grid.
        - Since there are 2 states, *alive* and *dead,* then the best choice for representing these states is a bool value.
            - A cell with a `false` in the gird indicates that the cell is dead, likewise, if a cell has the value `true`, that indicates that the cell is alive
- The `Universe` will contain these following functions:
    - Universe pseudo code:

      ```
      struct Universe {
      ```

```
    uint32_t rows;
    uint32_t cols;
    bool **grind;
    bool torodial;
}
```

- `uv_create`: this is the constructor function that creates a Universe.
  - The first two pramenters it accepts are `rows` and `cols`, indicating the dimensions of the underlying bollena grid.
  - The last parameter torodial is a boolean, if the value of `toroidal` is `true`, then the universe is *trodial*
  - The *return type* of this function is going to be of type `Universe *`, meaning the function should return a pointer to a `Universe`.
  - We will use the `calloc()` function from `<stdlib.h>` to dynamically allocate memory.
  - Pseudocode:

```
uint32_t **matrix = (uint32_t **) calloc(rows,
sizeof(uint32_t));
for (uint32_t r = 0; r < rows; r += 1) {
matrix[r] = (uint32_t *) calloc(cols,
sizeof(uint32_t));
}
```

- `uv_delete`: this is the destructor function for the `Universe`

        - This function frees any memory allocated for a `Universe` by the constructor function.
        - This function makes sure that there are no memory leaks.
        - Use `valgrind` to check for memory leaks.
        - INSERT PUSEDO CODE HERE
    - `uv_rows`: this function is an accessor function and returns the number of rows in the sepcificed `Universe`.
        - This is possible, but only inside `universe.c`
        - INSERT PUSEDO CODE HERE
    - `uv_cols`: this function is an accessor function and returns the number of columbs in the specified `Universe`.
        - INSERT PUSEDOCODE HERE
    - `uv_live_cell`: this function is a manipulator function and it marks the cell at row `r` and column `c` as *live*.
        - If the specified row and column lie outside the bounds of the universe, nothing changes.
        - Since we are using *bool*, we assume that *true* means live and *false* means *dead*.
        - INSERT PUSEDOCODE HERE
    - `uv_dead_cell`: this function marks the cell at row `r` and column `c` as *dead*.
        - Like in `uv_live_cell()`, if the row and column are out-of-bounds, nothing changed.
        - INSERT PUSEDOCODE HERE

- `uv_get_cell`: this function returns the value of the cell at row `r` and column `c`.
  - If the row and column are out-of-bounds, `false` is returned
  - Again, *true* means that the cell is alive.
- `uv_populate`: this function will populate the Universe with row-column pairs read in from `infile`
  - This function will require `<stdio.h>` since infile is a `FILE *` (`FILE` is defined in the `<stdio.h>` library)
  - The necessary *include* will be supplied in the `universe.h` for use.
- `uv_census`: this function will return the number of live neighbors adjacent to the cell at row `r` and column `c`.
  - ideally the universe of the game extends to infinity in the $\pm\ x$ and $\pm\ y$ directions; however, we cant have things falling off the edge of our universe.
  - We need to treat our universe (the flat grid) as a flat Earth or a *torus*.
  - If the universe is flat, or non-trodoial, then you should only conside the *valid* neighbors for the count.
  - If the universe is toridial then you sould consider all the neighbors as valid, you simply wrap to the other side.
- `uv_print`: this function prints out to the `outfile`.
  - A live cell is indicated with the character ‘o’ (a lowercase O) and a dead cell is represented with a character ‘.’ (a preiod).
  - We will need to use either `fprintf()` or `fputc()` to print out to the specified `outfile`.

        - You cannot print a trous, so you will always be prininng out the flattened universe.

Error Handling:

- I have not run into any errors yet during this assignment, but as I do, they will be documented in this section.

Citations:

- Throughout this assignment, I am doing high-level pseudocode collaboration with my sister Twisha Sharma (tvsharma). We are bouncing ideas off of each other and generally talked out the best ways to go about implementing the Game of Life.