Michael Nguyen
mnguy181@ucsc.edu
1/28/2021

CSE13S Winter 2021
Assignment 3: The Game of Life
Design Document

PURPOSE

The purpose of this program will be to design functions that will help implement the Game of Life, a zero-player game where its evolution is determined by the initial state. The game will be played on a two-dimensional grid of cells that represents a universe where each cell may either be dead or alive. Each step in time as the game progresses will be considered a generation. In this game, there will be 3 rules at the end of each generation:

1) Any live cell with two or three live neighbors will survive.
2) A dead cell with three live neighbors will become a live cell.
3) All the other cells will die from loneliness or overcrowding.

In this program, I will be creating an abstract data type (ADT) which will serve as an abstraction for the finite 2-dimensional grid of cells called the universe. Although the game was meant to be played on a possibly infinite two-dimensional grid of cells, unfortunately, computers do not have infinite memory. There will be two separate files that contain code in this programming assignment: life.c will contain my implementation of the Game of Life and universe.c will contain my implementation of the Universe ADT. The header file, universe.h will be supplied by the professor and will not be modified. The universe.c file will include constructor, destructor, accessor, and manipulator functions that will be required for the Game of Life including uv_create(), uv_delete(), uv_rows(), uv_cols(), uv_live_cell(), uv_dead_cell(), uv_get_cell(), uv_populate(), and uv_census(). The program will accept different command-line options including a command-line option that will specify the input file to read to populate the universe as well as another command-line option that will specify the output file to print the final state of the universe to. By default, the input will be stdin and the output will be stdout.

TOP LEVEL FOR UNIVERSE ADT

The top level design of my code for universe.c is given by the following pseudocode. Note that more in-depth explanations on the implementation of each function follows and that this is just a brief general outline of how I will approach the programming assignment in regards to implementing the Universe ADT.
Please do note that the following pseudocode was developed through the help of TA Eugene from his lab section on 1/26/2021.

Struct Universe
    Initialize the fields rows, cols, **grid, toroidal

```
uv_create()
        Universe *u = calloc() for memory allocation sizeof(Universe)
        Initialize and set u->rows, u->cols u->toroidal
        u->grid = calloc() for memory allocation sizeof(bool*)
        For loop() to iterate through each row
                u->grid[r] = calloc() for memory allocation sizeof(bool)
        Return
uv_delete()
        For loop() to iterate through each row
                Free the column
        Free the grid
        Free the universe
uv_rows()
        Return u->rows
uv_cols()
        Return u->cols
uv_live_cell()
        Set grid[r][c] to true
uv_dead_cell()
        Set grid[r][c] to false
uv_populate
        Initialize r and c to 0
        While loop(fscanf() is not equal to -1)
                grid[r][c] = true
uv_census()
        Initialize count to 0
        If toroidal
                For loop() iterate from r-1 to r+1
                        For loop() iterate from c-1 to c+1
                                If out of bounds
                                        A = (r1 + rows) % rows
                                        B = (c1 + cols) % cols
                                        If grid[A][B] is alive
                                                Count++
                                Else
                                        If alive
                                                Count++
        Else
                For loop() iterate from r-1 to r+1
                        For loop() iterate from c-1 to c+1
                                If out of bounds
                                        Continue
                                Else
                                        If alive
                                                count++
```

```
uv_print()
        For loop to iterate through each row
                For loop to iterate through each column
                        If live cell
                                fputc('o', outfile)
                        Else
                                fputc('.', outfile)
                Print new line after each column
```

The file universe.c will include the header file "universe.h" which has been provided to us by Professor Darrell Long, and we will not be able to modify this file. The file will also utilize the calloc() operation from the <stdlib.h> library for memory allocation as specified in the lab document. It will also include <stdio.h> library for FILE *.

DESCRIPTION FOR UNIVERSE ADT

Typedef struct Universe

The universe will be abstracted as a struct called Universe using typedef and will include several different fields: rows, cols, grid, and toroidal. The fields rows and columns will be specified when creating a universe, and the grid will be used to implement a 2-dimensional grid that includes all the cells. Because each cell has two states, either dead or alive, we will be representing each cell with a value of false (if dead) or true (if alive). Finally, the program will also be able to specify if the universe is toroidal (true) or flat (false) with the boolean field toroidal. A brief overview of the struct of the Universe is provided in the lab document.

Function *uv_create(int rows, int cols, bool toroidal)

This function serves as the constructor function that creates a Universe. It will accept rows and the number of columns as its parameters which will be used to create the dimensions of the 2-dimensional grid. If the value of toroidal is true, then the universe is toroidal, and if false, the universe will be considered flat. The function will thus return a pointer to a Universe. The implementation of the function will begin with the initialization of Universe *u. The universe will have its memory dynamically allocated using the calloc() operation from <stdlib.h> where the count (or number of elements) will be 1, and the size of the memory will be of size Universe. Then, we will initialize each of the different fields for the universe struct as follows: u->rows will be set to rows, u->cols will be set to cols, u->toroidal will be set to toroidal. In the next step, I will need to initialize the 2-D grid while also dynamically allocating its memory to be the size of the rows and columns specified by the function's parameters. To do this, I will begin by initializing u->grid and dynamically allocate its memory to be calloc(rows, sizeof(bool*)). To ensure that we allocate the proper amount of memory for the number of columns, I will need to utilize a for loop() that ends when r is greater than or equal to the number of rows (where r increments by 1 after each iteration). In this loop, I will dynamically allocate enough memory using calloc() for the number of columns for the grid at u->grid[r]. Finally, the function will return universe u (which is a pointer to the universe constructed).

Function void uv_delete(Universe *u)

This function serves as a destructor function for a Universe, freeing any memory allocated for a Universe by *uv_create(). The general idea is to begin by freeing from the inside first. Thus, I will begin the implementation by utilizing a for loop() that iterates through each row, freeing the memory of each column using free(). When the loop is finished iterating, I will then free the grid, and follow up with freeing the universe u itself.

Function int uv_rows(Universe *u)

This function serves as an accessor function that simply returns the number of rows in the Universe. Basically, it will return u->rows.

Function int uv_cols(Universe *u)

This function, like uv_rows(), serves as an accessor function that simply returns the number of columns in the specified Universe, returning u->cols.

Function bool inbounds(universe *u, int r, int c)

Although not required, this function will return true if the row r and column c lie inside the bounds of the universe. This will be useful for the other functions to ensure that r and c are in bounds. If r and c are out of bounds (less than 0 or greater than or equal to u->rows or u->cols), then the function will return false.

Function bool uv_live_cell(Universe *u, int r, int c)

This function will serve as a manipulator function that simply marks a cell at row r and column c as live (true). I will begin by checking if the row r and column c are in bounds with the function inbounds(). If r and c are in bounds, then set u->grid[r][c] as true. If the row and column are out of bounds, then nothing is changed.

Function bool uv_dead_cell(Universe *u, int r, int c)

This function, like uv_live_cell(), will also serve as a manipulator function that marks a cell at row r and column c as dead (false). I will begin by checking if row r and column c are in bounds with the function inbounds(). If r and c are in bounds, then set u->grid[r][c] as false. If the row and column are out of bounds, then nothing is changed.

Function bool uv_get_cell(Universe *u, int r, int c)

This function returns the value of the cell at row r and column c. If the row and column are not in bounds, then the function will return false. Otherwise, after checking if row r and column c are in bounds with the inbounds() function, I will simply return u->grid[r][c] (true if the cell is alive or false if the cell is dead).

Function bool uv_populate (Universe *u, FILE *infile)

This function's purpose will be to populate the Universe with "row column" pairs read in from the infile, requiring the use of the <stdio.h> library. While the first line of the input file will be used to construct the dimensions of the 2-dimensional grid for the universe, fscanf() will be used to read in the remaining pairs that will populate the universe. To do this, I will first begin by initializing variables r and c set to 0, and follow up with a while() loop that will end when fscanf()

reaches the end of the infile (or is equal to -1). Inside this loop, I will set u->grid[r][c] to true at the specified row-column pair that was read by fscanf(). In addition, if fscanf() reads a pair that is outside the bounds of the universe, I will return false, stating that the universe failed to be populated.

Function int uv_census(Universe *u, int r, int c)

This function's purpose will be to return the number of live neighbors adjacent to the specified cell at row r and column c. I will be needing to keep track of whether or not the universe is toroidal or flat in this function. I will begin by initializing a variable called count that will keep track of the number of live neighbors (set initially to 0). I will then initialize an if statement that checks if u->toroidal is true. If so, then I will need to utilize the % (mod) to find the neighbors in the case that the specified cell is at the edge of the grid. If a neighboring cell is considered to be out of bounds after checking if it is in bounds (through inbounds()), then I will simply use:

(r + rows) % rows
(c + cols) % cols

This will allow us to get all the neighbors if the universe is toroidal. I will utilize a for loop() that will iterate start from r-1 to r+1. Another nested for loop() will be used to get the columns of the neighboring cells starting from c-1 to c+1. Using the formula to find the neighbors in a toroidal universe, I will check if the cell is dead or alive, adding to the count variable if the neighbor is alive. In the case that the universe is flat, I will utilize another nested for loop() except for when checking if the neighbor is out of bounds, we will not utilize the same formula. Instead, we will continue with the iteration without adding to count if a "neighboring" cell is considered to be out of bounds in a flat universe.

Function uv_print(Universe *u, FILE *outfile)

This function will print out the universe to the outfile, indicating that a cell is alive with 'o' and '.' if the cell is dead. I will be using fputc() for this function. Keep in mind that although a universe may be toroidal, I will always be printing out a flat universe. To do this, I will once again be utilizing a nested for loop(). The first for loop() will iterate through each row, ending when int r reaches the number of rows in the grid (where r is 0 and increments by 1). The second for loop() will iterate through each column, ending when int c reaches the number of columns in the grid (where c is 0 and increments by 1 after each iteration). Inside the nested for loop(), if u->grid[r][c] is true, fputc('o', outfile). Otherwise, fputc('.', outfile). This function will also print a newline character after each row is printed.

TOP LEVEL FOR THE GAME OF LIFE

The top level design of my code for life.c is given by the following pseudocode. Note that more in-depth explanations on the implementation of the main() function will follow and that this is just a brief outline of how I would approach implementing the actual rules for each generation for the Game of Life based on the universe ADT.

Define the command-line options
Define the delay of 50000 microseconds for ncurses

```
Int main()
        Initialize ncurses screen as shown in the lab document
        Initialize and set the default values of *infile, *outfile, generations, toroidal, and silence
        While loop() for getopt()
        Error Handle for invalid input file, invalid # of generations
        fscanf() dimensions for universe
        Initialize Universe A and B
        Error Handle if uv_populate() unsuccessful
        Populate universe A using uv_populate()
        For  loop() for each generation up to number of generations
                If ncurses is silenced
                        For loop() for each row
                                For loop() for each column
                                        If number of live neighbors is 2 or 3
                                                If the cell is alive
                                                        It survives
                                                Else
                                                        Cell is dead
                                        If the cell is dead and number of live neighbors is 3
                                                Cell returns alive
                                        If the number of live neighbors is < 2 or >3
                                                Cell is dead
                        swap
                Else
                        Clear the screen
                        Same for loops() as above, except use mvprintw() to print 'o' to ncurses
                        Swap
                        Refresh the screen
                        Sleep for 50000 microseconds
        Close screen
        Close and free all dynamically allocated memory
```

Please note that this file will include the header file "universe.h" for the use of the universe ADT. Libraries used include the <ncurses.h> library for the use of ncurses, <stdio.h> for stdin and stdout, and <stdlib.h> library.

DESCRIPTION FOR GAME OF LIFE

To initialize the ncurses screen, I will be following the example code that was shown in the lab document which requires the use of initscr(). When initializing the default values of the game of life, the lab document specified that by default:

Infile = stdin
Outfile = stdout
Generations = 100
Toroidal will be false

Silence is false

Next, I will need to utilize a while loop() stat ends when the getopt() function is equal to -1. This while loop() was largely inspired by Eugene's code for how to handle command-line options for assignment 2. In this case, -t represents that a universe will be toroidal. -s specifies that ncurses will be silenced. -n generations specifies the number of generations a universe will go through. -i input will specify the input file that fscanf() will read from and -o output will specify the output file to print the final state of the universe to.

When conducting error handling, I will need to check if there is an invalid input file or an invalid number of generations. Thus, I will use fprintf(stderr, "") in order to print error messages and exit the program safely if there are errors in the input read from getopt(). If infile is NULL, then throw an error message and return 0.

Next, I will need to use an initial call to fscanf() to read the number of rows and columns of the universe I will be populating from the specified input. I will then create 2 universes (A and B) utilizing the constructor function uv_create() in accordance to the specified dimensions. Then, I will populate universe A utilizing the function uv_populate() which will read from the input file to populate the universe with live cells at specified row-column pairs. Obviously, if uv_populate returns false, then I will need to throw an error message that states that there was malformed input for the row-column pairs.

Next, I will utilize a for loop() that iterates from generation 0 to the specified number of generations. This for loop() will conduct the Game of Life based on the 3 rules specified in the lab document. Within this for loop(), I will first need to check if the user wanted ncurses is silenced. Either way, nested for loops() for each generation will need to be done except when silenced, there will be no output shown on ncurses.

For each row and column in the specified input, if the number of live neighbors obtained by using uv_census() is 2 or 3, then a live cell will survive (stays true) in universe B. If the cell is dead and the number of live neighbors is exactly 3, then it will return back to life (true) in universe B. However, if the number of live neighbors is less than 2 or greater than 3, then the cell will die (boolean false) in universe B.

At the end of each for loop()/generation, I will need to swap the universes similar to how a person would swap integer values. Universe A will always represent the current state of the universe while universe B will always represent the next state of the universe. To do this, I will need a temp variable.

If the user specified that ncurses is not silenced, then I will need to use clear() from the ncurses.h library, refresh(), and usleep(DELAY) to clear the screen, display universe A, refresh the screen, and then sleep for 50000 microseconds. The nested for loop() to iterate through each row and column will be exactly the same as how it is when ncurses is silenced. The exception is that when there is a live cell, then I will use mvprintw() at the specified row and column and print a character 'o'. Dead cells will only be printed in the final output of the universe. Of course, at the end of each generation, I will still need to swap universe A and B by utilizing a temp variable.

To close the screen of ncurses, I will use endwin(). Afterward, I will print the final state of the universe using uv_print(a, outfile) which will print the universe to a specified output location where live cells are indicated by an 'o' and dead cells are indicated by a '.' (period). Finally, I will close all files using fclose() to close the input and output files, and free all dynamically allocated memory using uv_delete() function which will free the universes properly.

UPDATES

- Updated the pseudocode and description of function uv_print(). At first, I forgot to print a newline character after each row was printed for a universe.
- I forgot to add break statements for each case in the getopt() while loop()
- Added an error handling description for invalid inputs