Nicolas Corfmat

CSE13S, Winter 2023

ncorfmat@ucsc.edu

Dealing With Dynamic Memory Allocation and Abstract Data Types

Assignment 4: The Game of Life

1 Description of Program

This program simulates “The Game of Life,” an evolutionary game created by mathematician John Horton Conway, that requires no further input beyond an initial state. Essentially, a 2-D grid representing a *universe* consists of either dead or alive cells, whose states are updated as time, or *generations*, progresses. The rules of the game are straightforward. If a live cell has two or three live adjacent cells, it survives. A dead cell becomes live if it has exactly three live neighbors. Lastly, any cell that does not meet this criteria dies, and the cycle continues. The game ends once every last generation has been performed.

2 Breakdown of Conway’s Game of Life

Unlike traditional games that require input from a user, ”The Game of Life” constitutes the idea that once a reality is set, the evolutionary path is fixed and any changes are a result of the game’s initial state. For this version, the user feeds an input file outlining the Universe’s initial layout and updates the Universe after each generation. Conway’s ideas are observed by how no matter how many times we run the same initial Universe, they all take up the same evolutionary path.

3 Making Use of "*ncurses*"

This program must include the ability to visualize the Universe's evolution through time, and what better way than to display each generation than by animating it on a window. To accomplish this, we turn to the *ncurses* library. Similar to the screen you are most likely reading this PDF on, our window follows a structured pipeline in order to print the desired image. The window first clears itself, creates a picture, refreshes itself, and then incorporates a delay between each frame. In the case of ”The Game of Life,” this was the process we had to implement. I began by initializing the screen with the *initscr()* function and then hiding the cursor using *curs_set()*.

Since each generation produces a new image, we must first clear the previous window with *clear()* in order to ensure a clean slate to display the updated Universe. Next, we display the current Universe by depicting live cells with "o's" and dead cells with a period. These can be printed using *mvprintw()*.

Additionally, we must refresh the window by calling *refresh()* and more importantly, include a delay between each image or "frame." Forgetting to include a delay would produce an instant animation, making it almost impossible to observe changes between each frame. For this program, we use the *delay()* function, giving it an input of 50,000 microseconds.

4 Compiling "The Game of Life"

In this program, we created variables representing Universes, however, we know that unlike integers and strings, "Universe" is not a conventional data type, but something known as an abstract data type. By essentially designing our own data type, we can decide what values this abstract data type (ADT) will hold and what operations they can possess. In our constructor, or the area where we initialize the values of our ADT, we create the grid of our Universe. Here is where we must remember to include likely the most important step, and that is allocating memory for our Universe and the grid. In doing so, we request that there is a block of space saved on the heap for the Universe.

Often during the process of testing my program, did I encounter issues with accessing memory. In one case, I failed to de-allocate memory from unused data types. Another common mistake was trying to access memory from the wrong location, consequently raising a "*Segmentation fault (core dumped)*" error. For example, in a function that was tasked to read data from a file, I requested that it read the file and perform tasks with the data contained in it. The obvious mistake here was that I assumed all data files passed to it contained data. This prompted me include a check that exits the function if the file was empty or continue if it consisted of data. Ensuring proper memory was freed and checking whether I was trying to access something that did not exist, were two approaches that helped me fix these issues.

5 Conclusion

Ultimately, this assignment taught me about the importance of assigning memory upon running the program, as it is difficult to predict how much space the program will require. Moreover, I came to understand how abstract data types are not only created, but implemented within other files by linking them and calling their operations. Lastly, I learned how freeing memory is arguably as important as allocating it, since this prevents memory leaks and other issues that might crash the program and lose potentially valuable data.