GAME OF LIFE

Fordham University

Department of Computer and Information Science

CISC 2200: Data Structures

Instructor: Prof. H. M. Ammari

Students: Destin Piagentini, Kevin Yonamine

Fall 2018

Table of Contents

Introduction

In the Game of Life, developed by a mathematician named Conway, cells are susceptible to reproduce, die, or survive. The action of each cell is determined by a set of rules. The cells are on a square grid of arbitrary size (a size determined by the program's user). Therefore, each cell is surrounded by eight other cells, unless the square is on a boundary. The rules for each cell are as follows:

1. Survive: The cell survives to the next generation if it has two or three adjacent cells that are alive.
2. Die: The cell dies by overpopulation if it has four adjacent cells that are alive. The cell dies by isolation if there is one or fewer adjacent cells that are alive.
3. Reproduce: Each dead cell that is adjacent to exactly three cells that are alive gives birth and the cell is alive for the next generation.

An important factor to consider is that all births and deaths occur at the same time during a generation, meaning each cells' behavior is determined by the state of the game at the start of the round, not as the round changes as the turn proceeds.

Design and Solution

Our Process - How We Arrived At Our Design

Our approach to this program was that of a divide-and-conquer methodology. Our first steps in creating this program was creating the class interface that would hold all our member functions and private data members (the class interface, however, would be added to as the need for more functions and data members became evident over time). From the start we knew we needed essential pieces of the class such as the pointer that would be dynamically allocated to a 2-D array in the constructor to hold the statuses of all the cells, or essential functions such as game over checking and printing the board, just to name a few.

Our next step was to create the class's constructors. Initially we had a default constructor that would initialize the board to a 10 x 10 square, and initialize all data members in accordance with this default configuration. The second constructor reads in two int values to be used as the board's height and width dimensions. The board and variables are initialized in accordance with the desired dimensions. A copy constructor was also created later on so the class adheres by the C++ rule of the "Big Three" since we have two pointers as data members in the class (the game board and the adjacent-count board). We also made a destructor to break down the 2-D array when the object went out of scope as part of the "Big Three" rule as well.

After the constructors were in place, we focused on developing the most basic algorithms needed before we could move on to the actual gameplay. The functions that got the most spotlight in the early stages of our development were the CheckGameOver function and the function that would count the adjacent cells to any given cell, and return the number. It was

critical for us to have these features in place so that once we began sculpting the gameplay we could test it immediately and have instant feedback for our bugs.

Once these features were in place, we had the ability to go on to craft the function that would update the board each turn, UpdateGame. We struggled with how to approach the fact that all births and deaths happen at the same time. We knew this meant that we could not apply the game rules to versions of the cells that were in the process of being changed. Ultimately, we decided to create an identical 2-D array within the function (called AdjacentCellsGrid) to hold the number of adjacent cells each cell had. The number of cells adjacent to any given cell would be held in a table similar to the game-board, so the number held at any given space corresponded to a cell on the game-board in that same space. This array (and a function that passed this new array in by reference and gathered the data after it was constructed in UpdateGame) was then able to "snapshot" the data of the cells at the beginning of the round. While the cells would change as UpdateGame cycled through them, the crucial data it needed would remain constant. Once this method was applied, the handling of rules was simple as it compared adjacent-cell counts and the alive/dead status of cells to determine what game rule to employ.

Lastly, we turned our attention to the "main" function, and made sure a basic, yet sensical game-loop was set up so that we were able to test the game. After a bit of debugging, the game was in a working state. While it was an achievement for us to have the game in a working state, this was merely "halftime" for us, as we had to focus on polishing both our algorithms and our front-end user experience.

For polishing the backend, we made a few refinements to our code to make sure our code was running efficiently. At this point we added a copy constructor, an overloaded assignment operator, and modified our destructor (which we had since the beginning). The major change in

this stage was making the adjacent-count 2-D array a part of the class interface, and not constructed and destructed every single time UpdateGame was run. Ultimately, that act of constantly constructing and deconstructing every turn is simply not as efficient as having that board within the class and simply updating every turn. As a result, every class dealing with that array needed to be reworked. Instead of reading in the array as a parameter, the functions now needed to draw from it from the class. The construction and deconstruction process of this array was also moved to all constructors and the destructor respectively.

We also, at this point, decided to embrace the method of allocating an extra row/column on each side of the GameBoard and AdjacentCellsGrid arrays. Previously, we created an equal-size replica of the table dimensions, but we noticed in the function used to count adjacent cells that there was a lot of comparisons between variables “i” and “j” to ensure that if one of the cells was on the edge or corner of the board, the algorithm would not include index numbers less than 0 or greater than the board’s height/width. In the end, however, we noticed that this called for a lot of comparisons each pass of the for-loop. To make matters worse, this function was called once for each cell per turn to count the adjacent alive cells. In the end, creating an array with an extra row/column on each side and thus making no cells live on the edge of the array meant we were able to eliminate those comparisons and increase efficiency.

For polishing the user experience, we finalized the dialogue interactions in the “main” function, as well as making the table of cells a lot more readable with typography that presented the cells in a table-like drawing. We also added a variable (and its accessor) to keep track of how many turns have passed to show at each round and at the game-over screen. We also made the game able to be replayed once it was over. Finally, we added to UpdateGame and CheckGameOver to include a case handler for a game with unchanging cells. The functions

utilize an enum GameOverCondition defined outside of the class in GameofLife.h to determine which GameOver case is met (either the cells are all unchanging or are all dead) so that the "main" function can print the appropriate error message. To determine whether cells were unchanging, we declared a new boolean GameChanged as part of the class to serve as a flag in UpdateGame. If the board did not change, GameChanged would evealuate to false. The CheckGameOver function then uses this value to determine whether a game-over resulted in an unchanging board or not. The CheckGameOver function was updated to pass-by-reference a enum object of type GameOverCondition. In "main", a temporary GameOverCondition enum object is called with the function so that "main" can identify the game-over case.

A Brief Explanation of Each Function (Algorithm) and How It Works

int main()

The "int main()" function is the function that handles the user-end experience, as well as processing what to do (game-over/run a new turn/quit the program) based on game status and/or user input. The "int main()" function runs at O(n) where "n" is the amount of turns performed across all games (remember, "main" supports multiple play-throughs).

The "main" function begins by explaining the rules of the game to the user. Next, input from the user is used to instantiate an object of class "GameofLife" which will call the constructor that uses the two dimensions to create a board with those dimensions. Next, "main" prints the board and begins the turn phase of the game. At any time the user can manually exit, but if they don't the board is continually updated each turn and printed to the user (along with the turn count). The function then checks to see if the game is over, passing in an GameOverCondition enum variable to record which case is met (if any cases are met). If it is, a game-over screen is printed based on the correct case (are the cells dead or stagnant,

unchanging). The “main” function also utilizes the fact that if the board does not change, the turn count wont increment to know not to print the new board in the same sense as the advancement of a typical turn, instead saying “The board did not change” before the printing a game over message. If the game is ever evaluated to be over, or the user quits, the program asks the user if they want to start with a different board. If so, the user is brought back to the start where input is asked and the process repeats. If the user declines to run again, the “main” function (and, therefore, the program) ends.

GameofLife() [Default Constructor]

This default constructor should theoretically never be used, but is a good practice to include. This constructor creates game board and adjacent-cell count 2-D arrays based on default dimensions of 12 x 12 (the game board will be 10x10, 2x2 for the other empty rows/columns). The BoardHeight and BoardWidth private data members are set to twelve each, the GameBoard array is allocated and stuffed with random true/false values (excluding the lining edges, which are all set to false), and the AdjacentCellsGrid is allocated but not updated. The TurnCount private data member is also initialized to 0.

GameofLife(const int &, const int &)

This constructor creates an object based on a given dimension size for the table (+ 2 for empty rows/columns), which, in the main program, will be the user input. The BoardHeight and BoardWidth private data members are set to their respective values each, the GameBoard array is allocated and stuffed with random true/false values (excluding the lining edges, which are all set to false), and the AdjacentCellsGrid is allocated but not updated. The TurnCount private data member is also initialized to 0. This constructor runs at O(xy) where “x” is the user-entered BoardHeight and “y” is the user-entered BoardWidth. This order is derived from the for-loop

needed to allocate the two 2-D arrays alongside the for-loops that assign randomly generated values to GameBoard.

GameofLife(const GameofLife &)

This constructor creates an object based on a duplicate object of the same type. The BoardHeight and BoardWidth private data members are set to be the values of the duplicate object's BoardHeight and BoardWidth variables respectively, and the GameBoard and AdjacentCellsGrid arrays are allocated and stuffed with identical values to the respective corresponding arrays in the duplicate object. The TurnCount private data member is also initialized to the duplicate's value. This constructor runs at O(xy) where "x" is the BoardHeight and "y" is the BoardWidth of the duplicate object. This order is derived from the for-loops needed to allocate the two 2-D arrays alongside the for-loops that assign values to GameBoard and AdjacentCellsGrid.

~GameofLife()

This is the class's destructor for when an object goes out of scope. The destructor is needed as part of the "Big Three" of C++ due to the presence of dynamically allocated pointers. The destructor's intended job is to free up the dynamically allocated 2-D arrays. The destructor performs this task by cycling through the first dimension of the array, and using the "delete[]" command for each. Finally, now that the pointers act like a one-dimensional arrays, we can use the delete command to free them. The destructor runs at O(n) where "n" is the BoardHeight dimension, or the amount of rows the for-loop needs to cycle through to delete.

GameofLife & operator= (const GameofLife &)

This overloaded assignment operator copies an object of the same type's values to the invoking object's data members. The BoardHeight and BoardWidth private data members are set

to be the values of the duplicate object's BoardHeight and BoardWidth variables respectively, and the GameBoard and AdjacentCellsGrid arrays are deconstructed, reallocated and stuffed with identical values to the respective duplicate arrays. The TurnCount private data member is also initialized to the duplicate's TurnCount value. This constructor runs at O(xy) where "x" is the BoardHeight and "y" is the BoardWidth of the duplicate object. This order is derived from the for-loops needed to deconstruct the two 2-D arrays alongside the for-loops that reallocate and assign values to GameBoard and AdjacentCellsGrid.

void PrintBoard() const

This function includes the typography and for-loops needed to print a well-designed, clean looking board to the user. A lot of typography techniques are employed to get the borders perfectly aligned. The function also translates a boolean's true/false value to the chars 'A' or 'D' representing alive or dead for a better user experience. This function runs at O(xy), where "x" is the BoardHeight variable and "y" is the BoardWidth. This order is derived from the double-nested for-loops needed to cycle through each element of the array.

bool CheckGameOver(GameOverCondition &) const

This function cycles through the 2-D GameBoard array to determine whether all cells are dead and it is game over, or if the cells are unchanging between rounds. The function records the evaluated case (stagnant cells or all dead cells) to a variable passed in of type GameOverCondition (enum). To check the stagnation of cells, the function checks the value of the GameChanged flag evaluated in UpdateGame. If GameChanged is false (the cells are unchanging), then the game is over and GameOverCondition is recorded in accordance. If the game is not over from stagnation, the function then checks if all cells are dead. To determine whether all cells are dead, the function declares a boolean variable called "AllDead" to represent

whether all the cells are dead, and initialize it to true as we assume everything is dead until proven otherwise. We cycle through the GameBoard, and the "AllDead" variable will only change if we detect an alive cell. If we detect an alive cell, we set "AllDead" to false and skip to the end of the loop since we have all the information we need, and there is no need in spending time and resources on pointless comparisons. The enum variable is also updated in accordance with this result. This function runs at a worse-case scenario of O(xy) where "x" is BoardHeight and "y" is BoardWidth. This scenario is when all the cells are dead, and we did not find an alive cell so we did not end up bypassing any section of the loop. Best-case scenario is when the first cell in GameBoard is alive, so the function runs at O(1).

void UpdateGame()

This function is used to advance a turn in the game's timeline. This function first resets the flag variable that will check whether the game has changed or not to false. If any births/deaths are performed, this flag will become "true" to signal that the game is still changing.. Then, the function calls the function that will update the AdjacentCellsGrid and thus take a "snapshot" of the current board before changes are applied. Next, the function cycles through each cell in play (not the ones on the edge that are not supposed to effect the game) and decides what to do. If the cell is alive and has more than 3 or less than 2 adjacent alive cells, it dies. If the cell is dead and is surrounded by exactly 3 adjacent alive cells, it is brought back to life. Living cells adjacent to 2 or 3 alive cells have no change, and thus require no instruction. At the end, if GameChanged is false we do not update the turn counter. Though we technically ran through another turn to detect whether the game changed or not, the game was truly over after the last time the board changed, so we keep the turn counter as it was then. This function runs at O(xy),

where “x” is the BoardHeight variable and “y” is the BoardWidth. This order is derived from the double-nested for-loops needed to cycle through each element of the array.

int GetTurnCount() const

This is an accessor function for the TurnCOunt variable so that the main function is able to produce the turn count to the user. This function runs at O(1).

int CountAdjacentCells(const int &, const int &)

This function is meant to count and return the number of adjacent cells to any given cell. The integers that are passed into the function are the coordinates of the cell whose surroundings we will search. First the function declares a temporary variable which will keep count of the adjacent cells (and will ultimately be what we return). Next, the coordinates given are tested to make sure they are valid. If they are invalid (cells in the empty extra rows/columns are deemed invalid as they would go off the array), we return 0. If the given coordinates is a valid cell, we continue.

When we search the 9 cell area of the given cell and all cells surrounding it, we will include the given cell in this search so that we don’t need to add unnecessary comparisons to check if we need to skip it or not each loop through. Instead, we will eliminate these comparisons and simply include the given cell in the search. Since the cell can not be adjacent to itself, we deduct one from the total adjacent cell count. If the given cell is dead, however, it will not count itself, so before we cycle through the 9 cell search area, we added an if-statement to make sure that the given cell adds one to the total whether it is alive or dead so we can simply return our total – 1. After this if-statement we cycle through the 9-cell search area. This includes the column left of the given cell, the same column of the given cell, and the column right of the given cell for the row above the cell, the same row of the cell, and the row below the cell. This

function runs at O (3^2) or O(9) because we will search 9 cells no matter the input's location on the board.

void UpdateAdjacentCellsGrid()

This function will use CountAdjacentCells to create a "snapshot" of the game board before changes are made. This array will hold the number of adjacent cells that the cell in the corresponding location on the GameBoard array has. This is needed in the UpdateGame function so that AdjacentCellsGrid can provide a reference for how the board looked before changes were started for that turn. The function operates by cycling through the coordinates of each cell in play on the game's board (not the extra cells lining the board) and assigning the CountAdjacentCell to that index of AdjacentCellsGrid. This function runs at O(xy), where "x" is the BoardHeight variable and "y" is the BoardWidth. This order is derived from the double-nested for-loops needed to cycle through each element of the array.

void PrintAdjacentCellsGrid() const

This function includes the typography and for-loops needed to print a well-designed, clean looking board of AdjacentCellsGrid for debugging purposes. A lot of typography techniques are employed to get the borders perfectly aligned. Each cell on this grid contains the number of living cells adjacent to that cell. This function runs at O(xy), where "x" is the BoardHeight variable and "y" is the BoardWidth. This order is derived from the double-nested for-loops needed to cycle through each element of the array.

General Overview of the Program Loop – When and What Functions Are Used

When executed, the program first provides an introduction to the reader explaining the rules and any relevant info. Then, the program jumps into getting the user's desired board

dimensions. Once those board dimensions are entered, the program instantiates a class object based on those dimensions, and a gameboard as well as an adjacent-count board used in internal calculations is created with +2 rows and columns to account for the empty cells lining the edge of the board, improving efficiency in the CountAdjacentCells function. In the game board, each cell is randomly generated based on a 50% true/false approach (unless it lies on the edge and is supposed to be empty, where it is then declared as dead so as not to show up in adjacent counts). Note that the adjacent-count 2-D array grid is allocated but not loaded with values because when the UpdateGame function is run, it will update that array anyway. In other words, it would pointlessly "snapshot" the same board configuration twice despite there being no change. At this point, the starting board is displayed to the user and the game has officially begun.

Everything after the starting board is continuously looped until either all cells are dead or the user decides to exit by typing "Q" for quit. At the start of each turn, the program asks the user to enter any key to continue, or 'Q' to exit. If any non-exit key is entered, the UpdateGame function will run.

The UpdateGame function updates the adjacent-count "snapshot" array, and cycles through all cells on the game board (does not cycle through the empty extra rows/columns because those cells are not a part of the game) to decide their fate based on the game's rules. The function that updates the adjacent-count array (AdjacentCellsGrid) cycles through each position on the game board and uses the CountAdjacentCells function to count the number of adjacent cells on that position of the board and load that number into each space on the adjacent-count 2-D array. The CountAdjacentCells scans the 9-cell area surrounding and including the given cell, and calculates the "alive/dead" status of each cell and returns the total – 1 to account for and negate the given cell who's surroundings are being searched (if the cell is dead it will still

contribute + 1 to the total count). The UpdateGame then uses the AdjacentCellsGrid to decide the fates of each cell. Cycling through all the cells of the game in play, it compares the adjacent count of that cell with its alive/dead status and decides what to do based on the game rules. Either the cell will remain unchanged, it will die (via overpopulation or isolation), or it will come back to life (adjacent to exactly three cells).

Once the UpdateGame function has finished deciding the fate of each cell, the board is once again printed to the user along with the turn count. The board is then checked to make sure the game isn't over (all cells dead). If all cells are dead, the game alerts the player and exists the do-while loop cycling through the turns. If the game is not over, the turn loops again, and the next thing the user sees is another message allowing them to either continue or quit the program. If a user gets a game-over, or the program is manually quit, the program allows the user to start over with a different board specification (and repeat the same process minus the initial rules explanation) or quit the program entirely.

How We Know It Works

We've employed several debugging techniques during the course of the development of this program to ensure the program was producing accurate results. By far the most useful debugging tool was being able to print AdjacentCellsGrid just below the GameBoard, so we can ensure that a) AdjacentCellsGrid is producing accurate results and b) that the correct changes were being made to each cell based on the number of adjacent living cells each turn. Additionally, since a copy constructor and overloaded assignment operator was included as part of the "Big Three" of C++ but never used in the program, we had to set up specific cases to make

sure they worked properly. We've attached screenshots proving all of this in the "Test" section of this document.

Implementation

Note: some comments or code may be edited from its original format to be more readable in a word processor (namely, Microsoft Word).

LifeBoardGame.cpp

```
// Destin Piagentini & Kevin Yonamine
// September 20, 2018
// Professor Ammari - Data Structures
// Fordham University
//Copyright 2018 Destin Piagentini, Kevin Yonamine

// LifeBoardGame.cpp : This file contains the 'main' function. Program execution begins
and ends there.


//References
//1: https://stackoverflow.com/questions/42335200/assignment-operator-overloading-
returning-void-versus-returning-reference-param

#include <iostream>
#include <cstdlib> //Needed for random generator
#include <ctime> //Needed for rndom generator
#include "GameofLife.h"

int main() {

	srand(time(NULL)); //Seed the random generator.

	GameOverCondition Reason; //Variable for game-over case handling

	int Height, Width; //The variables where the user's input will be loaded into.

	char UserInput; //Will be used to determine whether to continue or quit.

	//Introduction, instructions, and gathering of input:

	std::cout << "Welcome to the game of life!\n\n";

	std::cout << "\nIn this game, a board of 'cells' (the height and width of which
	are decided by you) will be randomly generated.";
	std::cout << "\nInitially, some cells will be dead and some cells will be alive.";
	std::cout << "\nThe game will keep progressing through rounds aslong as there is
	at least one cell alive.\n";
	std::cout << "\nEach round, the following rules will be applied:";
	std::cout << "\n\n1. Living cells adjacent to exactly 2 or 3 alive cells will
	remain unchanged.";
	std::cout << "\n2. Living cells adjacent to less than 2 or more than 3 living
	cells will die.";
```

```
std::cout << "\n3. Dead cells adjacent to exactly three alive cells will come back
to life.";
std::cout << "\nNote: all births and deaths occur at the same time during a
generation.";
std::cout << "\nNote: On the game's board, 'A' represents an alive cell and 'D'
represents a dead cell.\n";

do {

	std::cout << "\nPlease enter the board dimensions of the game.\n";
	std::cout << "Height: ";
	std::cin >> Height;
	std::cout << "Width: ";
	std::cin >> Width;
	std::cout << "\n";

	GameofLife GameInstance(Height, Width); //Instantiate the game and board
	with the given input data.

	std::cout << "'A' represents an alive cell, 'D' represents a dead cell.\n";

	std::cout << "Starting board: " << std::endl << std::endl;
	GameInstance.PrintBoard();

	do { //Will keep runing the game until user quits or all cells are dead.

		std::cout << "Press any key followed by 'Enter' to continue, or type
		'Q' followed by 'Enter' to quit: ";
		std::cin >> UserInput;

		if (UserInput != 'Q' && UserInput != 'q') { //If the user does not
		want to exit, we run a turn.

			int TurnCountBeforeTurn = GameInstance.GetTurnCount();
			//Needed to check if the board remained the same so we dont
			print again.

			GameInstance.UpdateGame(); //Update each turn.


			if (TurnCountBeforeTurn != GameInstance.GetTurnCount()) {
			//If the turn counts are the same, the board hasnt changed (so
			game over). If the turn counts are different, the game is
			continuing.

				std::cout << "\nResults After Turn " <<
				GameInstance.GetTurnCount() << ": \n\n";

				GameInstance.PrintBoard(); //Show user updated board.

			} else { //If the game is over due to stagnation, we want to
			tell the user and show it.

				std::cout << "\n";

				GameInstance.PrintBoard(); //Show the user the board
				did not change.
```

```
                        std::cout << "The board did not change.\n"; //Tell the
                        user.

                }
        }

        //No check for Userinput != 'Q' because there should be no change to
        the board from last turn.
        //If there was a game over last turn, it would've shown last turn.

        if (GameInstance.CheckGameOver(Reason)) { //If game over, print a
                                                  game-over screen.

                if (Reason == DEAD) {

                        std::cout << "Game over! All cells died after " <<
                        GameInstance.GetTurnCount() << " turns!\n";

                } else if (Reason == STAGNANT) {

                        std::cout << "\nGame over! The board is unchanging from
                        round to round, and no further changes to the board can
                        be made!\n"; //Board does not print so we add line
                        space at beginning to separate from previous line.

                }

        }

    } while (!GameInstance.CheckGameOver(Reason) && (UserInput != 'q' &&
    UserInput != 'Q')); //Repeat until game is over or user quits.

    //Ask to play again.

    std::cout << "\nType 'Y' or 'y' followed by 'Enter' to play again, or any
    other key and 'Enter' to exit: ";
    std::cin >> UserInput;

} while (UserInput == 'y' || UserInput == 'Y'); //If the user wants to play again
we repeat.

return 0;

}
```

GameofLife.h

```
//Copyright 2018 Destin Piagentini, Kevin Yonamine

//The class interface for "GameofLife" class.

#pragma once

enum GameOverCondition {STAGNANT, DEAD};
```

```
class GameofLife {

	public:

	//Constructors/Destructors
	GameofLife();
	GameofLife(const int &, const int &);
	GameofLife(const GameofLife &); //Copy constructor
	~GameofLife();

	//Overloaded Assignent Operator
	GameofLife & operator = (const GameofLife &);

	//Public functions
	void PrintBoard() const; //Prints the current board.

	bool CheckGameOver(GameOverCondition&) const; //Checks if all cells are dead, thus
	                                              ending the game.

	void UpdateGame();

	//Accessor function for the TurnCount data member.
	int GetTurnCount() const {

		return TurnCount;

	}

	private:

	//Data Members
	int BoardWidth; //Board's width length

	int BoardHeight; //Board's height length

	bool ** GameBoard; //Two-Dimensional array that will become the game's board.

	int ** AdjacentCellsGrid; //Two-DeimensionalArray that will "snapshot" the cells
	adjacent cell count at the beginning of a turn before changes are made.

	int TurnCount; //Keeps track of how many turns have passed.

	bool GameChanged; //Keeps track of whether the game is stagnot or not. If no
	changes between rounds then game is over.

	//Helper Functions

	int CountAdjacentCells(const int &, const int &) const; //Counts adjcent cells to
	any given location.

	void UpdateAdjacentCellsGrid(); //Updates the adjacent count 2-D array to reflect
	new changes.

	void PrintAdjacentCellsGrid() const; //Prints the AdjacentCellsGrid. **FOR
	DEBUGGING PURPOSES**

};
```


GameofLife.cpp

```
//Copyright 2018 Destin Piagentini, Kevin Yonamine

//The class implementation for the "GameofLife" class.

#include <iostream>
#include <cstdlib>
#include <ctime>
#include "GameofLife.h"


//Default constructor. If there are no parameters given.
GameofLife::GameofLife() {

	BoardHeight = 12; //General, default board dimensions.
	BoardWidth = 12;

	GameBoard = new bool *[BoardHeight]; //Dynamic allocation of the first dimension
	of the array.

	AdjacentCellsGrid = new int *[BoardHeight]; //We dont need extra space for the
	outside rows here.

	for (int i = 0; i < BoardHeight; i++) {

		GameBoard[i] = new bool[BoardWidth]; //Dynamic allocation of the second
		dimension of the array, creating the columns for each row.

		AdjacentCellsGrid[i] = new int[BoardWidth]; //We dont need extra space for
		the outside rows here.

		for (int j = 0; j < BoardWidth; j++) { //We can start to fill the newly
		created columns with values. We need to wait before we start counting
		adjacent cells.

			if (i == 0 || i == BoardHeight - 1 || j == 0 || j == BoardWidth - 1)
			{ //If the cell in question is on the outside of the user's
			requested dimensions (if in the empty rows/columns surrounding the
			game board).

				GameBoard[i][j] = false;
				AdjacentCellsGrid[i][j] = 0; //Fill outside with a 0 count of
				adjacent cells.

			} else {

				GameBoard[i][j] = (rand() % 2); //Initialized to random 0/1,
				which is false/true for a boolean.

			}

		}

	}
```

```
	//Do not need to update AdjacentCellsGrid as the UpdateGame function will do this.
	It would be an extra call for no reason.

	TurnCount = 0; //No turns have started, so TurnCount is zero.
	GameChanged = false; //Initialize to false (no rounds yet).


}


//Constructor that reads in the user's desired height/width dimensions for the board and
creates the board in accordance
//int Height: How high the board will be (first dimension of the array).
//int Width: How wide the board will be (AKA how many columns, the second dimension of
the array).
GameofLife::GameofLife(const int & Height, const int & Width) {

	BoardHeight = Height + 2; //Set the board's height and width to the user's desired
	lengths.
	BoardWidth = Width + 2; //Plus 2 for the empty rows around the outside.

	GameBoard = new bool * [BoardHeight]; //Dynamic allocation of the first dimension
	of the array.
	AdjacentCellsGrid = new int * [BoardHeight]; //We dont need extra space for the
	outside rows here.


	for (int i = 0; i < BoardHeight; i++) {

		GameBoard[i] = new bool[BoardWidth]; //Dynamic allocation of the second
		dimension of the array, creating the columns for each row.
		AdjacentCellsGrid[i] = new int[BoardWidth]; //We dont need extra space for
		the outside rows here.

		for (int j = 0; j < BoardWidth; j++) { //We can start to fill the newly
		created columns with values. We need to wait before we start counting
		adjacent cells.

			if (i == 0 || i == BoardHeight - 1 || j == 0 || j == BoardWidth - 1)
			{ //If the cell in question is on the outside of the user's
			requested dimensions.

				GameBoard[i][j] = false;
				AdjacentCellsGrid[i][j] = 0; //Fill outside with a 0 count of
				adjacent cells.

			}
			else {

				GameBoard[i][j] = (rand() % 2); //Initialized to random 0/1,
				which is false/true for a boolean.

			}

		}

	}
```

```
	//Do not need to update AdjacentCellsGrid as the UpdateGame function will do this.
	It would be an extra call for no reason.

	TurnCount = 0; //No turns have started, so TurnCount is zero.
	GameChanged = false; //Initialize to false (no rounds yet).


}

//Copy Constructor
//Part of the Big Three of C++.
//Needed so, if the copy constructor is used, the "GameBoard" pointer does not point to
same address in memory.
//In other words, only values copied.
//const GameofLife & Duplicate: The object who's values will be copied to the new object.

GameofLife::GameofLife(const GameofLife & Duplicate) {

	BoardHeight = Duplicate.BoardHeight; //Copy original object's values.
	BoardWidth = Duplicate.BoardWidth; //Already + 2 from duplicate.

	GameBoard = new bool *[BoardHeight]; //Dynamically allocate first dimension in a
	location that is not being used, i.e. not the first objects memory location.
	AdjacentCellsGrid = new int*[BoardHeight];

	for (int i = 0; i < BoardHeight; i++) {

		GameBoard[i] = new bool[BoardWidth]; //...Same thing for the second
		dimension.
		AdjacentCellsGrid[i] = new int[BoardWidth];

		for (int j = 0; j < BoardWidth; j++) {

			GameBoard[i][j] = Duplicate.GameBoard[i][j]; //Since the '[]'s
			dereference the array, we are copying values here NOT locations.
			AdjacentCellsGrid[i][j] = Duplicate.AdjacentCellsGrid[i][j];

		}

	}

	TurnCount = Duplicate.TurnCount; //Copy the turncount's values.
	GameChanged = Duplicate.GameChanged;


}

//Destructor
//Called when the object goes out of scope.
//We need this to delete the two-dimensional dynamic arrays when we are done with them,
and free up the space in memory.

GameofLife::~GameofLife() {

	for (int i = 0; i < BoardHeight; i++) { //First we delete the columns of each row.

		delete[] GameBoard[i];
```

```
			delete[] AdjacentCellsGrid[i];

		}

		delete[] GameBoard; //Finally, we delete the rows themselves.
		delete[] AdjacentCellsGrid;

}

//Overloaded assignment operator
//Needed because of the presence of a pointer in the class's interface. (Part of the C++
Big Three).
//const GameofLife & Duplicate: The object who's values will be copied.
GameofLife & GameofLife::operator = (const GameofLife & Duplicate) {

		//First we free up our 2-D arrays so we can reconstruct them.

		for (int i = 0; i < BoardHeight; i++) { //First we delete the columns of each row.

			delete[] GameBoard[i];
			delete[] AdjacentCellsGrid[i];

		}

		delete[] GameBoard; //Finally, we delete the rows themselves.
		delete[] AdjacentCellsGrid;

		BoardHeight = Duplicate.BoardHeight; //Copy original object's values.
		BoardWidth = Duplicate.BoardWidth;

		GameBoard = new bool *[BoardHeight]; //Dynamically allocate first dimension in a
		location that is not being used, i.e. not the first objects memory location.
		AdjacentCellsGrid = new int*[BoardHeight];

		for (int i = 0; i < BoardHeight; i++) {

			GameBoard[i] = new bool[BoardWidth]; //...Same thing for the second
			dimension.
			AdjacentCellsGrid[i] = new int[BoardWidth];

			for (int j = 0; j < BoardWidth; j++) {

				GameBoard[i][j] = Duplicate.GameBoard[i][j]; //Since the '[]'s
				dereference the array, we are copying values here NOT locations.
				AdjacentCellsGrid[i][j] = Duplicate.AdjacentCellsGrid[i][j];

			}

		}

		TurnCount = Duplicate.TurnCount; //Copy the turncount's values.
		GameChanged = Duplicate.GameChanged;


		return *this; //Return a reference to the same object.

}
```



```
//A public member function that will be used to print the current game's board to the
screen, called in "main".
//Includes typography for readability by the user.
//Const: Is not intended to change any values, just print them.
void GameofLife::PrintBoard() const {

	//All i/j = 1 and BoardHeight/BoardWidth - 1 to account and not print empty
	rows/columns.

	//Generation of table top border.

	//We should have a two char deficit (on account of the beginning "| "), but the
	table ends in an invisible space char, so only one "-" is needed to allign
	everything.
	std::cout << '-';

	for (int j = 1; j < BoardWidth - 1; j++) {

		std::cout << "----"; // D/A char + " | " = four char spaces each run
		through

	}

	std::cout << '\n'; //Extra line for row start.

	//Now we go down the rows to generate the actual table.

	for (int i = 1; i < BoardHeight - 1; i++) {

		std::cout << "| "; // "| " for table typography.

		for (int j = 1; j < BoardWidth - 1; j++) {

			if (GameBoard[i][j]) { //If this cell's value is "true", print it as
			'A' (Alive) for readability.

				std::cout << "A | ";

			} else { //If this cell's value is "false", print it as 'D' (Dead)
			for readability.

				std::cout << "D | ";

			}
		}

		//We should have a two char deficit (on account of the beginning "| "), but
		the table ends in an invisible space char, so only one "-" is needed to
		allign everything.
		std::cout << std::endl << '-';

		for (int j = 1; j < BoardWidth - 1; j++) {

			std::cout << "----"; // D/A char + " | " = four char spaces each run
			through

		}
```

```
			std::cout << '\n'; //Extra line for readability and to seperate each row.

		}

		std::cout << '\n'; //Extra line for readability.

}

//A private member function that will be used to print the current game's adjacent cell
count board to the screen, called in "UpdateGame".
//This function exists for debugging purposes only.
//Includes typography for readability by the user.
//Const: Is not intended to change any values, just print them.

void GameofLife::PrintAdjacentCellsGrid() const {

		//Generation of table top border.

		//All i/j = 1 and BoardHeight/BoardWidth - 1 to account and not print empty
		rows/columns.

		//We should have a two char deficit (on account of the beginning "| "), but the
		table ends in an invisible space char, so only one "-" is needed to align
		everything.
		std::cout << '-';

		for (int j = 1; j < BoardWidth - 1; j++) {

			std::cout << "----"; // D/A char + " | " = four char spaces each run
			through

		}

		std::cout << '\n'; //Extra line for row start.

		for (int i = 1; i < BoardHeight - 1; i++) {

			std::cout << "| "; // "| " for table typography.

			for (int j = 1; j < BoardWidth - 1; j++) {

				std::cout << AdjacentCellsGrid[i][j] << " | "; //Print the adjacent
				cell count for that cell.

			}

			std::cout << std::endl << '-'; //We should have a two char deficit, but the
			table ends in an invisible space char, so only one "-" is needed to allign
			everything.

			for (int j = 1; j < BoardWidth - 1; j++) {

				std::cout << "----"; //number + " | " = four spaces each run

			}

			std::cout << '\n'; //Extra lines for visibility
```

```
	}

	std::cout << '\n'; //Extra lines for visibility

}

//This is the function called every "turn" to decide the fates of the cells on the board,
and advance the game.
//Handles cases for death and birth (surviving doesn't require any instruction).
//If adjacent count 2 or 3 and alive, that cell survives. If the adjacent count is below
2 or above 3, that cell dies.
//If a dead cell is surrounded by exactly 3 cells (adjacent count = 3), that cell comes
back to life.
void GameofLife::UpdateGame() {

	GameChanged = false;

	UpdateAdjacentCellsGrid(); //Take a "snapshot" of the board's adjacent count
	before we make changes.

	//PrintAdjacentCellsGrid(); //Debugging - User shouldnt see this function run, but
	useful for debugging and having the program show us its process.

	for (int i = 1; i < BoardHeight - 1; i++) { //Cycle through each cell and decide
	their fate. Skip empty rows/columns.

		for (int j = 1; j < BoardWidth - 1; j++) {

			//Death
			if ((AdjacentCellsGrid[i][j] > 3 || AdjacentCellsGrid[i][j] < 2) &&
			GameBoard[i][j] == true) { //If alive and adjacent count is more
			than 3 or less than 2.

				GameBoard[i][j] = false; //Cell dies
				GameChanged = true;


			}

			//Birth
			if (AdjacentCellsGrid[i][j] == 3 && GameBoard[i][j] == false) { //If
			dead and surrounded by excatly three adjacents.

				GameBoard[i][j] = true; //Cell comes back to life.
				GameChanged = true;


			}

		}

	}

	if (GameChanged) { //If the game has changed at all (and, thus, a new round has
				passed).

		TurnCount++; //New turn so we update the count.
```

```
	}

}

//This is a private Helper function used to count the cells adjacent to a given cell at
any given time, and returns that number.
//const int & VertSpot: The row of the inspected cell (who's adjacents will be counted).
//const int & HorzSpot: The column of the inspected cell (who's adjacents will be
counted).

int GameofLife::CountAdjacentCells(const int & VertSpot, const int & HorzSpot) const {

	int AdjacentCount = 0; //Variable that will ultimately be returned.

	//Must use temporary variables because we need the originals to compare against
	for for-loop exit.
	int i, j;

	//Check i and j are valid spots of the array. Cells outlining the edges are
	considered invalid.

	if (VertSpot >= BoardHeight - 1 || VertSpot < 1 || HorzSpot >= BoardWidth - 1 ||
	HorzSpot < 1) { //Checks to ensure the locations given are valid. If not, returns
	0 adjacent cells.

		return 0;

	}

	if (GameBoard[VertSpot][HorzSpot] == false) {

		//To avoid having to check if the cells in the search area are the given
		cell each time, we simply deduct one to our final total count to account
		for the given cell.

		//This, however, will only work if the given cell is alive. Therefore we
		need to count the cell whether its dead or not for this method.

		//Therefore, we count the dead cell before the for-loop to avoid having a
		comparison each time for the given cell. This reduces total comparisons
		needed.

		AdjacentCount++;

	}

	//Example: Do not check if j is above the range

	for (i = VertSpot - 1; i <= VertSpot + 1; i++) { //Cycles through the row above,
	to the row of, to the row below.

		for (j = HorzSpot - 1; j <= HorzSpot + 1; j++) { //Cycles to the column
		left of, to the column of, to the column right of.

			//Execution order:
			//Upper left cell, cell above, upper right cell
			//Cell left of, same cell (which is ignored), cell right of
			//Lower left cell, cell below, lower right cell
```

```
                        //Do not need to check for valid locations due to outside cells.

                        if (GameBoard[i][j] == true) { //Checks to make sure the cell is
                        alive. Will count given cell, but we reduce by 1 at the end to
                        negate this.

                                AdjacentCount++; //If alive and not the given cell, we have an
                                adjacent!

                        }


                }

        }

        return AdjacentCount - 1; //Return the accumulated count.

}

//This is a private helper function used to update the AdjacentCellsGrid, which records
the number of cells adjacent to any given cell at any given instance.
//The 2-D array stores the number in a table where the actual cell is for practicality.
AKA this table is a mirror of the board with numbers for each cell's adjacents.

void GameofLife::UpdateAdjacentCellsGrid() {

        for (int i = 1; i < BoardHeight - 1; i++) { //Cycle through all cells except those
        on the outside.

                for (int j = 1; j < BoardWidth - 1; j++) {

                        AdjacentCellsGrid[i][j] = CountAdjacentCells(i, j); //Uses the
                        CountAdjacaentCell helper function to get an accurate count.

                }

        }

}

//This function will scan the board for any alive cells.
//If all are dead, there is no possible way for the game to continue.
//This is called in "main" to know when to stop the game.

bool GameofLife::CheckGameOver(GameOverCondition& Reason) const {

        bool AllDead = true; //This will only be changed if we find an alive cell. If this
        does not change, we know all cells are dead.

        if (GameChanged == false) { //If the game hasn't changed we know

                Reason = STAGNANT;

                return true;

        } else {
```

```
            for (int i = 1; i < BoardHeight - 1; i++) { //Skip outside columns/rows.

                  for (int j = 1; j < BoardWidth - 1; j++) {

                        if (GameBoard[i][j] == true) { //If we find that one cell is
                                                         still alive.

                              AllDead = false;
                              Reason = DEAD;
                              //Skip to the end of the loops. No more calculations.
                              We only need one counterexample.

                              j = BoardWidth;
                              i = BoardHeight;

                        }

                  }

            }

      }

      return AllDead; //If we get this far, we know the game's changed. So we return
                        whether all cells are dead or not.

}
```

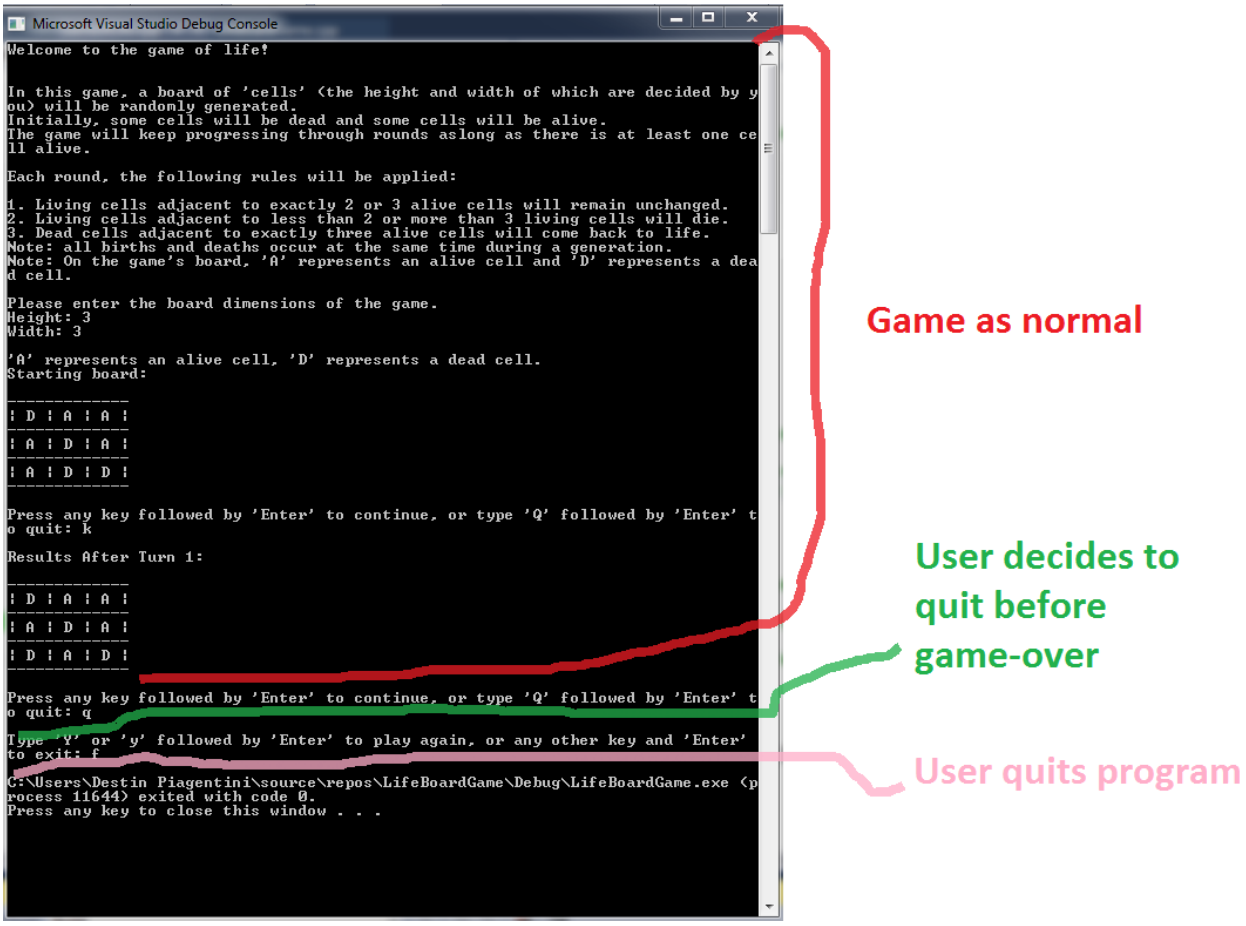
Tests

A Typical Game

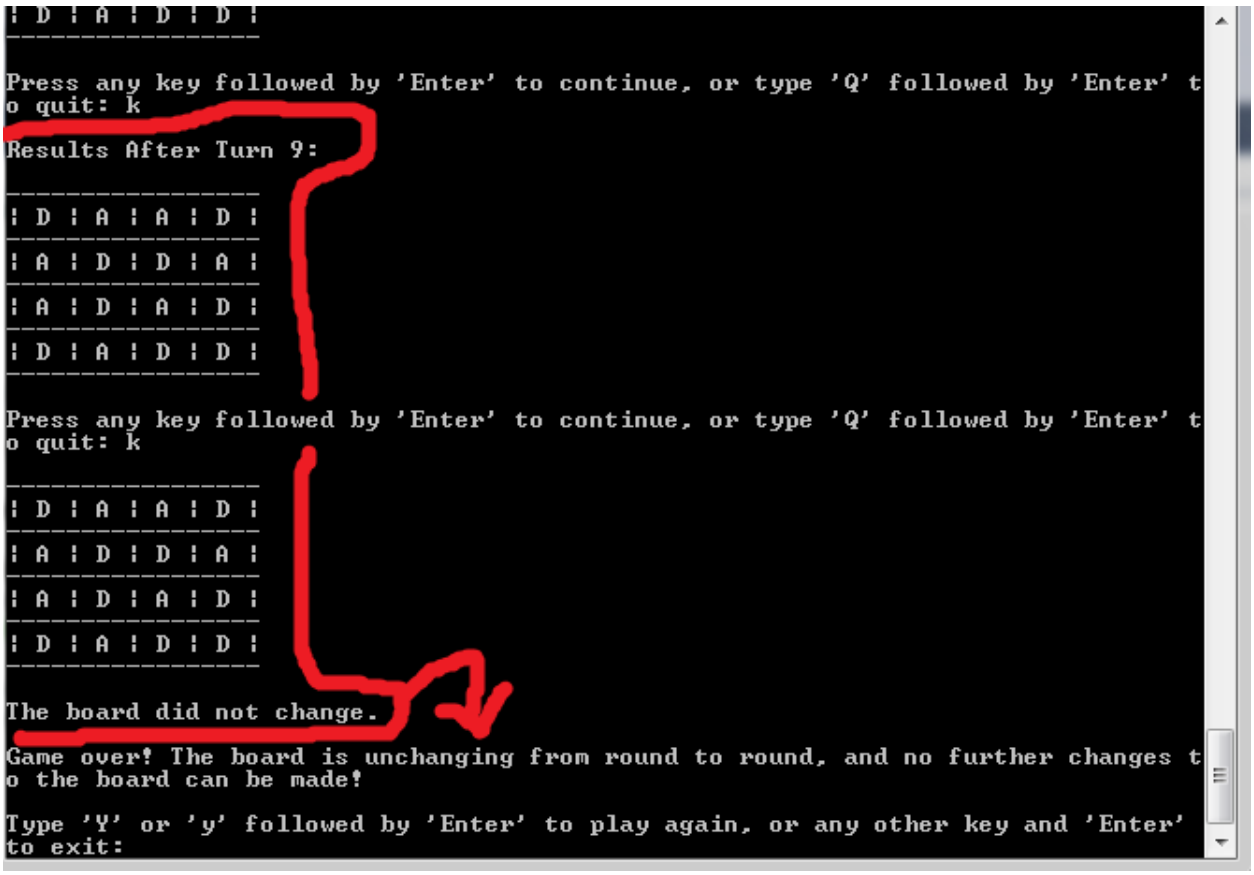
The following screenshots represent the user experience for a typical game in the final build of the program.


Results After Turn 4:
Press any key followed by 'Enter' to continue, or type 'Q' followed by 'Enter' to quit: k
Results After Turn 5:
Press any key followed by 'Enter' to continue, or type 'Q' followed by 'Enter' to quit: k
Results After Turn 6:
Game over! All cells died after 6 turns!
Type 'Y' or 'y' followed by 'Enter' to play again, or any other key and 'Enter' to exit: y
Please enter the board dimensions of the game.
Height: 3
Width: 3
'A' represents an alive cell, 'D' represents a dead cell.
Starting board:
Press any key followed by 'Enter' to continue, or type 'Q' followed by 'Enter' to quit: k
Results After Turn 1:
Game over! All cells died after 1 turns!
Type 'Y' or 'y' followed by 'Enter' to play again, or any other key and 'Enter' to exit: k
C:\Users\Destin Piagentini\source\repos\LifeBoardGame\Debug\LifeBoardGame.exe (process 4560) exited with code 0.
Press any key to close this window . . .
Normal turns
Game Over
User Decides to Play Again
User enters new board dimensions
Game over Again
User does not enter 'y' or 'Y', so the progream ends.

A scenario in which the user exits before game-over.

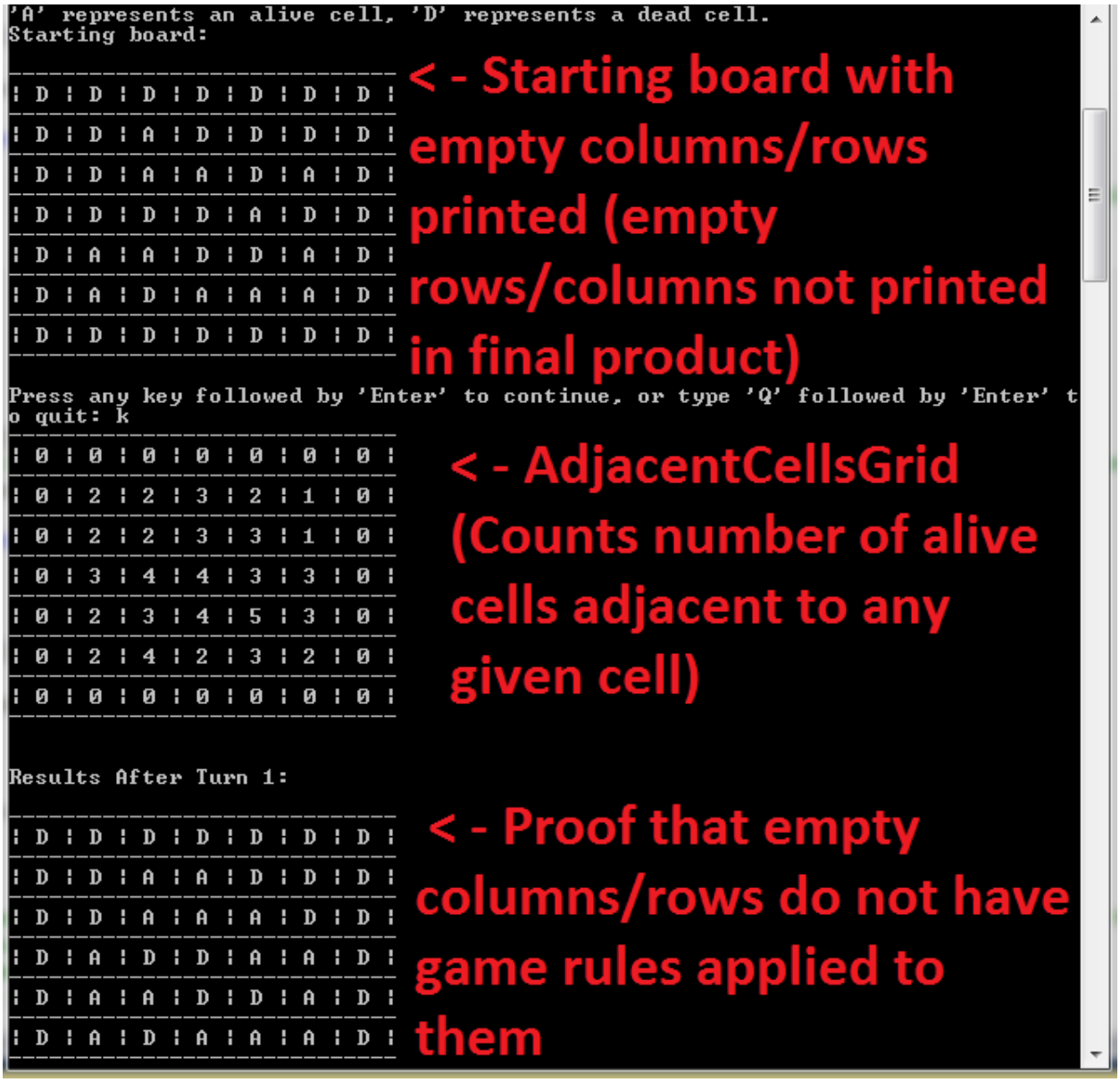


A scenario in which the game is over due to unchanging cells

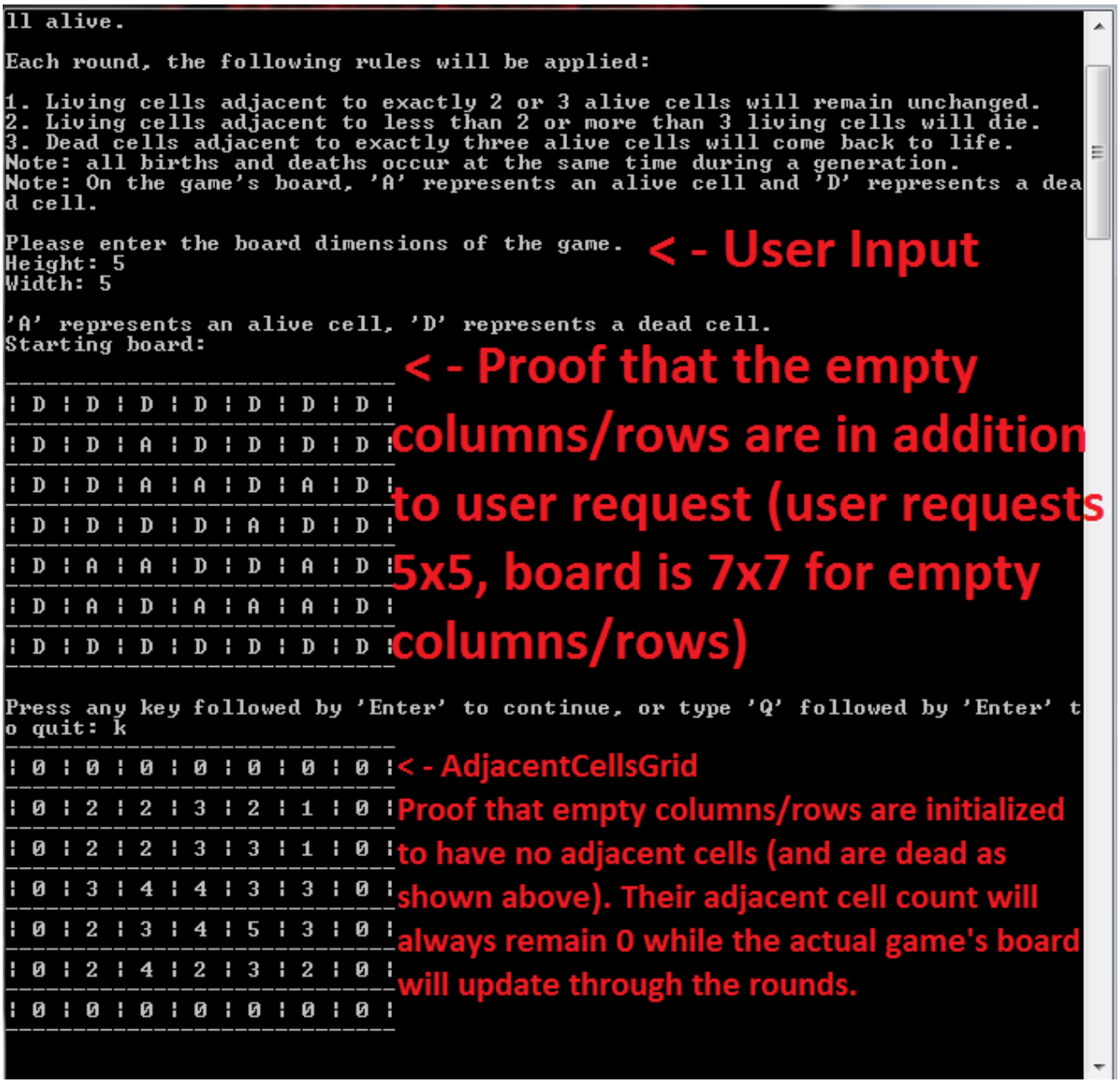


Debugging and Proof of Working Code

The following screenshot includes pictures of the tables used for debugging. This proves the working state of AdjacentCellsGrid and that GameBoard is, indeed , allocated extra Rows/Columns. The differences here vs the final build is that a) the empty columns/rows are not printed, and b) the user does not see the AdjacentCellsGrid. These tables are used for debugging and proving that the code works as intended. Note that the empty rows/columns do not effect gameplay.

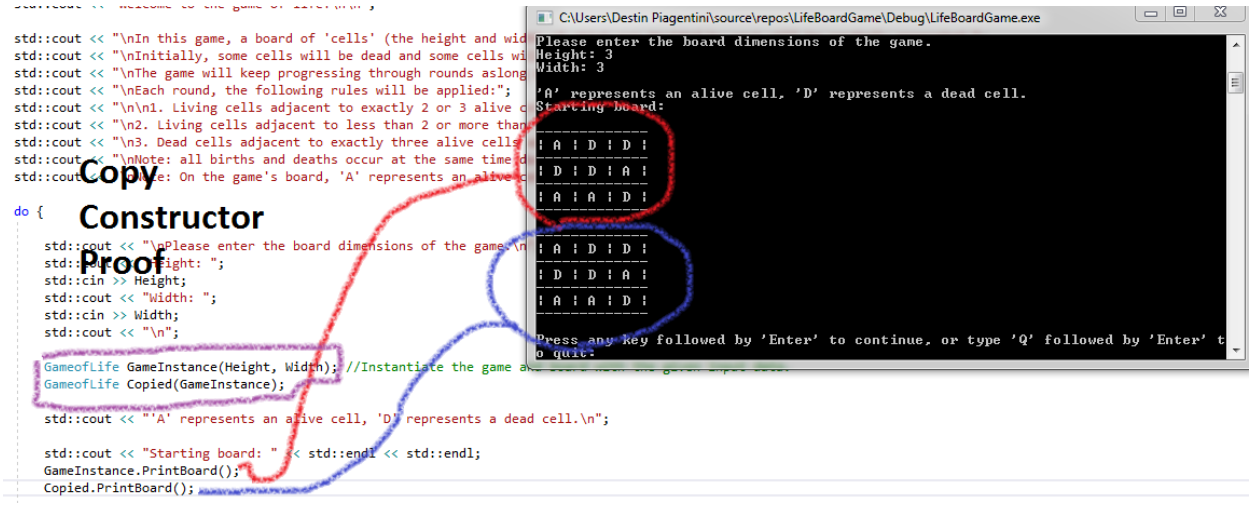


The following screenshot is proof that the empty rows/columns surrounding the table is not apart of the dimensions of the board the user requests. In other words, if the user requests a 5x5 board, we create a 7x7 board and don't use part of the requested 5x5 board for the empty columns/rows.

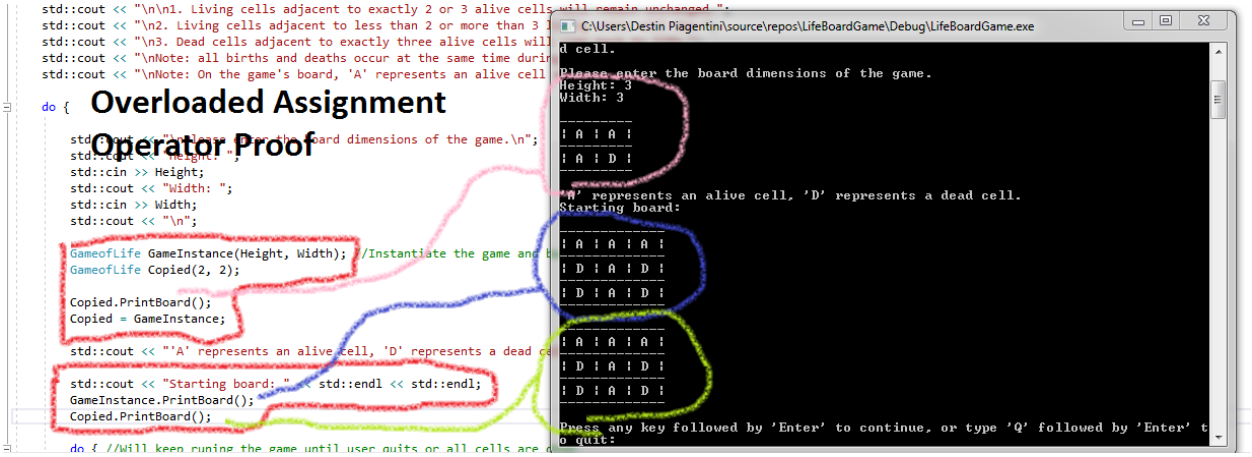


The following screenshot shows a special scenario in which we test that the copy constructor works. This does not exist in the final build, and is for testing purposes only to

ensure that the copy instructor is implemented as intended.



The following screenshot shows a special scenario in which we test that the overloaded assignment operator works. This does not exist in the final build and is for testing purposes only to ensure that the overloaded assignment operator is implemented as intended.



Lessons Learned

This Game of Life project utilizes two dimensional arrays and nested loops. Loops can get very complicated quickly, if they are nested and it is important to make sure the code is readable so that we can find and correct mistakes. Comments are just as important because they can tell you what each part of the program is supposed to do. One of the most helpful things in a big project, is debugging statements, and debugging operations such as printing AdjacentCellsGrid. They helped us to determine how the program operated during tests, and to see if it was running as intended. In tests we always had AdjacentCellsGrid printed alongside the GameBoard. This is very important in determining if the program employed the stated rules for each cell.

An important concept we also learned was the importance of efficient design. We had a working version of the program very early in, but refactoring our code and eliminating as much needless operations as possible took longer than figuring out the initial process. We had to change are code in big ways several times including (but not limited to) moving AdjacentCellsGrid to being part of the class itself and not a singular function, and redesigning our table and all functions dealing with the table to support extra, empty rows/columns to make the CountAdjacentCells function much more efficient and much less comparison heavy. In the end, the effort shows in the speed and efficiency of the code. The effort of refactoring was clearly worth it.

On smaller notes, several concepts from CS1 and CS2 were reinforced during the development of this program including the “Big Three,” random generation, and 2-D arrays. This program help cement our understanding and application of each topic.

References

During our Computer Science 1 class with Professor Kounavelis, we explored ways to randomize numbers using built in C++ functions. We drew from some of the code from that assignment when randomizing the cells at the beginning of the program, setting each cell to alive or dead based on a 50/50 chance.

We also looked back on several programs created during our CS2 class to relearn how to create and implement the “Big Three” (destructor, copy constructor, and overloaded assignment operator).We also consulted a thread on StackOverflow in hopes of better understanding the ways in which overloaded assignment operators work. The most helpful answer was from user “dasblinkenlight”.

The thread can be found here: https://stackoverflow.com/questions/42335200/assignment-operator-overloading-returning-void-versus-returning-reference-param