**CIS 2275 C++ Programming Part II NAME:**

# Program 8 The Game of Life 200 pts P8 total

**Inheritance and Polymorphism 50 pts PatternMaker**

**Grade Breakdown: 200 points**

**20 pt You participate in displaying your project, your own Cell and your patterns.**

**Your program must be running in order for this portion of the score.**

**20 pts Your user interface is simple to use. You follow program specs as described here.**

**75 pt (15 pts each) Successful coding and execution of each of your 5 cell types**

**20 pts Inheritance is used correctly (as specified) and in its own .h and .cpp files.**

**25 pts Polymorphism correctly used with virtual functions and the polymorphic array**

**20 pts Your project is correctly executed using a Console App.**

**20 pts Other program specs met**

**Turn in requirements:**

1. Name your project LastnameP8, such as NelsonP8.
2. **5 pts** Submit to Blackboard, remove the intermediate debug files/folders and then upload your zipped project.

**Program Requirements:**

1. **5 pts** Write your name, email address and file name at the top of your source code in a comment.
2. **5 pts** Use cout statements to write your name, program title, and program objective to the screen so that it is the first thing I see when your program runs. This is your course header.
3. **5 pts** Use good C++ programming style and formatting for your program. Use appropriate comments to explain what you are doing.

**Program Requirements:**

The Game of Life, is a cellular automaton It was devised by the British mathematician John Horton Conway in 1970. It is the best-known example of a cellular automaton. The "game" is a two-dimensional zero-player game, meaning that its evolution is determined by its initial state, requiring no further input from humans. Each cell of the automaton has two states, “alive" or "dead". The program reads an initial configuration and displays how it evolves.

Here is a youtube video of Dr. Conway’s discussion of the game:

<http://thinkorthwim.com/2007/05/27/john-conway-talks-about-the-game-of-life/>

Originally, Conway played the Game of Life on a Go Board, but we will code it using Object – Oriented Principles. The universe of the Game of Life is an infinite two-dimensional orthogonal grid of square cells, each of which is in one of two possible states, alive or dead. Every cell interacts with its eight neighbors, which are the cells that are horizontally, vertically, or diagonally adjacent. This is called a Moore neighborhood.

For the **Conway Cell**, the flow of the game is that at each generation, for each cell, the number of live neighbors is counted. The following transitions occur:

1. Any live cell with fewer than two live neighbors dies, as if caused by under-population.
2. Any live cell with two or three live neighbors lives on to the next generation.
3. Any live cell with more than three live neighbors dies, as if by overcrowding.
4. Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

The initial pattern constitutes the *seed* of the system. The first generation is created by applying the above rules simultaneously to every cell in the seed—births and deaths occur simultaneously, and the discrete moment at which this happens is sometimes called a *tick* (in other words, each generation is a pure function of the preceding one). The rules continue to be applied repeatedly to create further generations.

First, we will build a program to create a file containing the initial pattern. It will be called **PatternMaker,** and will create text files with one pattern at a time for the grid of 45 rows and 78 columns. It will begin by filling the 2\_D grid with ‘.’, which means that the cell is dead. Then code an initial pattern of live cells by placing ‘\*’ in those cells. Create a text file, naming it something descriptive, like “bar.txt” or “cross.txt.” The first line of the file will be the number or rows and the number of columns in the grid, with a space in between (45 78). Then write out the grid. This file will provide the initial pattern for your Game of Life.

These are other cell models that we will use:

I. The **Fredkin Cell** was developed by Edward Fredkin, a physicist at Carnegie Mellon University. It is a replicator where a cell survives or is born if there are an odd number of neighbors. Under this ruleset, every pattern self-replicates; furthermore, every pattern will eventually produce an arbitrary number of copies of itself, all arbitrarily far away from each other. Instead of the Moore neighborhood, it uses the Von Neumann neighborhood, which consists of four directions, up, down, right and left. The rules are:

1. A dead cell becomes a live cell if 1 or 3 neighbors are alive.
2. A live cell becomes a dead cell if 0, 2, or 4 neighbors are alive.
3. Otherwise, the cell stays the same.

II. The **Modified Fredkin Cell** also keeps track of the age of the cell. The rules are the same as for the Fredkin cell, except that if a cell survives to the next generation twice, it becomes a live Conway cell. All of the Conway rules apply to that cell.

III. The **Seeds Cell** is in the same family as the Game of Life, initially investigated by Brian Silverman and named by Mirek Wójtowicz. In Game of Life terminology, a pattern in which all cells that were

on turn off at each step is called a phoenix. Nearly every pattern in Seeds explodes into a chaotic mess that grows to cover the entire universe. The rules are:

1. A dead cell becomes a live cell if exactly 2 neighbors are alive.
2. All other cells become dead.

We will model these five cells using the abstract class LifeCell, below. The Conway Cell, the Fredkin Cell and the Seeds Cell all derive from the Life Cell. The Modified Fredkin Cell derives from the Fredkin Cell. Do some research. Either select a different Cell type and Rules or create one yourself. This last one will be selection 5.

class LifeCell

{

protected:

static const int ROWS{45};

static const int COLS{78};

int count{0}, rows{0}, cols{0}; //count is the number of neighbors alive counted

string pattern; //the index of the initial pattern chosen.

bool bOpen{false}; //states whether the file was opened successfully

char cell [ROWS][COLS]; //Grid of cells. If '\*' -> cell is alive, if '.' -> cell is dead

char nextCellState[ROWS][COLS];

void InitializeBoard(); //reads the pattern file and initializes the board

virtual void GetLivingNeighbors(int r, int c) = 0;

virtual void UpdateCells() = 0; //apply the rules here

void SetNextState(); //set new states into the cells

public:

LifeCell(); //default constructor. All cells are dead initially.

void SetPattern(string pat); //Sets the initial pattern and calls InitializeBoard

void UpdateBoard(); //Calculates the next generation, sets the new values into the grid and calls SetNextState()

string PrintBoard(); //returns a string of the board for display

void Clear(); //Reset the board to all dead cells

};

In main, reset the window size to its largest size for your screen.

Create an array of LifeCell pointers, and create objects of each kind of derived class. Then set the addresses of the derived objects into the pointers:

LifeCell \*pLife[5];

ConwayCell con;

FredkinCell fred;

ModifiedFredkinCell modFred;

Seeds seed;

Nelson dog;

pLife[0] = &con;

pLife[1] = &fred;

pLife[2] = &modFred;

pLife[3] = &seed;

pLife[4] = &dog;

This sets up the polymorphic array. Display information that introduces the game of Life (probably using a function). Next, open a do-while or while loop for the user to play repeatedly. Describe the different cells to the user and ask the user to choose a Cell Type. Show the user a menu of patterns and ask the user to choose an initial pattern. Set the pattern into the cell type object chosen by the user.

Open another do-while or while loop that will set the game in motion and roll the generations forward. Call the derived class object through the base class array pointer to access the class method:

do

{

pLife[config]->UpdateBoard();

cout << pLife[config]->PrintBoard();

Sleep(150);

system("cls");

lifeCount++;

}while(lifeCount < 200);

Ask the user if he/she wants to try another game. When the user has finished, display a good-bye statement.

Be sure to use exception handling for the two user inputs.