Eddie Peters
CS 162 OSU W14
1/19/14

Conway's Life Initial Design Document

Structure

My design of life will be built with five classes, one for cells (called Cell), one for collections of cells (Board), one for games (Game), one for an individual starting configuration (Config) and one for sets of starting configurations (ConfigSet).

The classes and all their functions will be defined in an interface/implementation file pair called life (life.h and life.cpp).

Some configurations will be provided in the life library. The user will be able to define more as he/she wants and load them into the interface using a method of the Game class.

Algorithm
(for making successive generations)

1. An initial configuration will be loaded by filling an array with boolean values.
2. The next generation will be calculated by leaving the original array intact and filling a second array by determining the fate of each element in the first array.
3. The first array will be replaced by the second.
4. Return to step 2 until a generation produces no change or the user issues a keyboard interrupt.

Classes and Functions in life library

Game class

Member	Description	Input/Work/Output
Public		
Game()	Constructor that sets the default size of the game board.	In: rows, cols Work: rows = 22 cols = 88 Out: Nothing. Just initializes rows/cols.
void play()	Starts a game of life with board size 22 x 88, unless user has changed this dimension with the dims() method. Only	In: User choices given by keyboard. Work:

	default configurations will be loaded unless user has added configurations using add_config() method.	Give user choices from list of loaded configurations. Allow them to choose using positive int values. call config_menu() Give user choice over number of generations to run, selected by entering a positive int at the keyboard. call user_gens() Run the game. call run_game() Out: First, list of possible starting configurations. Then the successive generations based on the initial configuration and Conway's rules. Stops when either the next generation is the same as the last or numGens generations have been printed.
void add_config(Config newConfig, int rows, int cols)	Adds a configuration to the configuration list. Clears out configuration list if a new Config is added that is not the same size as the other Configs in the list.	In: configList Work: dims(rows, cols) configList.add(newConfig) Out: Nothing. configList has been changed.
Private		
ConfigSet configList	Stores the loaded configurations.	
int confChoice	Stores the configuration chosen to play out in the game. Corresponds to a configuration in configList, indexed starting at 0.	
int numGens	Stores the number of generations to run the game. Value of 0 means infinite.	
int rows	Stores number of rows in the board used for the game.	
int cols	Stores number of collumns in the board used for the game.	
void config_menu()	Allows the user to choose a configuration from those loaded in configList	In: configList Work: for i in range (0, configList.num()) Print i + 1 Print configList.name(i) confChoice = get_int(1, configList.num()) - 1 //gets an int in range

		(0, configList.num()), and marks it as the one the user wants to use Out: Nothing. Sets value of confChoice
void user_gens()	Allows the user to choose the number of generations to run.	In: numGens Work: numGens = get_int(0) Out: Nothing. Sets value of numGens.

void run_game()	Runs the game based on values of configList, confChoice, numGens, rows, and cols.	In: configList, confChoice, numGens, rows, cols
		
		Work:
		
		try
		Board gameBoard = Board(configList.conf(confChoice), rows, cols)
		catch (NoSuchConfig)
		print that config does not exist.
		exit (-1) //remember to include cstdlib
		
		gameBoard.print()
		
		for (i = 0; i < numGens; i++)
		
		try
		gameBoard.next() //Makes next gen current gen
		gameBoard.print() //Prints current generation
		
		catch (BoardStatic) //Gameboard has stopped changing
		
		print that gameboard has stopped changing
		
		Out:
		
		numGens generations of the gameBoard
		(infinite if numGens is 0)

void dims(int height, int width)	Changes the dimensions of the game board, and if they are not 22 x 88, empties configList.	In: rows, cols, configList Work: if !(height = rows & width = cols) rows = height cols = width configList.empty() print that conflist has been emptied since all Configs must be the same dimension

Cell class

Member	Description	Input/Work/Output
Public		
Cell(bool val)	Constructor that creates a Cell with a status of alive (true) or dead (false). Note there's no default constructor, so Cell must be created with a value	In: bool val argument Work: Sets private state value to value of argument val Out: None. Cell is created with private member variable state set to argument val
bool status()	Return value of private state variable, which stores the boolean value indicated alive (true) or dead (false)	In: state Work: return state Out: Value of state

bool next_state()	Returns next state of the cell based on number of living neighbors (object must be embedded in an array)	In: current bool value of var state
		
		Work:
		
		int neighbors = Cell.count_neigbors() //Count living neighbors
		
		//throw exception here when number of neighbors < 0 or > 8
		
		if state = true //currently alive
		if neighbors < 2 // (underpopulation)
		return false //dead next time
		else if neighbors < 4 // 2 <= neighbors <= 3 (balanced state)
		return true //alive next time
		else if neighbors // 4 <= neighbors <= 8 (overpopulation)
		return false //dead next time
		else //logic mistake or error occurred
		print that something bad happened
		
		else if state = false //currently dead
		if neighbors = 3 //(reproduction)
		return true //alive next time
		
		else //boolean values stopped working like they used to
		print that something bad happened
		
		//catch out of range neighbors exception here
		
		Out: return statements above
		
		Will return false (dead) or true (alive) according to above tree.

Private		
bool state	Current state of Cell (true = alive, false = dead)	

int count_neigbors()	Counts the number of living neighbors (object must be embedded in an array)	In: array in which Cell is embedded, dimensions of this array, and location of the Cell in that array
		
		Work:
		
		location of current Cell:
		r = row #
		c = col #
		
		Count following elements of containing array:
		
		[r + -1][c + -1]
		[r + -1][c + 0]
		[r + -1][c + 1]
		[r + 0][c + -1]
		//Note: do not count actual cell's value [r + 0][c + 0]
		[r + 0][c + 1]
		[r + 1][c + -1]
		[r + 1][c + 0]
		[r + 1][c + 1]
		
		To handle edges, modulate each index by the width or height of the containing array. For instance, if number of rows in array is 22, modulate r +1 above by 22.
		
		Do so using the following loop:
		
		for i in range(-1, 1)
		for j in range (-1, 1)
		if not (r = 0 & c = 0)
		//rows is containing array height
		//cols is containing array width
		//lastGen comes from Board class (a friend)
		numNeighbors +=
		lastGen[(r + i) % rows][(c + j) % cols].status()
		
		Output:
		
		return numNeighbors, the number of living neighbors surrounding the Cell

Board class

Member	**Description**	**Input/Work/Output**
Public		
Board(Config initConfig, boardH, boardW)	Constructs a Board whose configuration is initConfig and sets up lastGen as a	In: initConfig, nextGen, lastGen, boardH, boardW Work:

	meaningless ptr. Also sets boardSize	nextGen = initConfig.get_cells()
		//Gets the number of items
		rows = boardH
		cols = boardW
		
		//lastGen must be something so it can be removed by next()
		lastGen = new Cell
		
		Out:
		
		Nothing. Changes what nextGen and lastGen point to.

friend class Cell	Allows Cell objects to access important members of Board objects, such as their rows & cols, and the main array itself.	None

void next() throw (BoardStatic)	Deletes lastGen, points lastGen to nextGen, and points nextGen to the next generation of Cells.	In: lastGen, nextGen, rows, cols
		
		Work:
		
		delete lastGen
		
		lastGen = nextGen
		
		nextGen = new Cell*[rows]
		for i in range(0, cols)
		  nextGen[i] = new Cell[cols]
		
		for i in range (0, rows)
		  for j in range (0, cols)
		  nextGen[i][j] = lastGen[i][j].next()

void print()	Does not change nextGen, but prints it to the std out.	In: nextGen
		
		Work:
		
		for i in range (0, rows)
		  for j in range (0, cols)
		  print nextGen[i][j]
		
		//Might need to play with whitespace here.
		
		Out:
		
		nextGen printed to screen in row/column format.

Private		
Cell ** nextGen		
Cell ** lastGen		
int rows	Stores number of rows in the Board.	
int cols	Stores number of cols in the Board.	

ConfigSet class

Member	Description	Input/Work/Output
Public		
ConfigSet()	Constructor that creates a ConfigSet by loading the default Configs stored in the library in the unnamed namespace.	In: set, NUM_DEFAULTS Work: set = new Config[NUM_DEFAULTS] for i in range (0, NUM_DEFAULTS) set[i] = defaultConfigs[i]
int num()	Returns setSize, the number of configurations in the ConfigSet.	In: setSize Work: return setSize Out: Returns setSize
string name(int index)	Gets the name of the configuration at the index in set.	In: int index Work: return set[index].name() Out: Returns the name of the configuration at index in set.

Config conf(int index) throw (NoSuchConfig)	Returns the Config at index in the set.	In: index, set, setSize
		
		Work:
		
		if (index <= setSize)
		  return set[index]
		
		else
		  throw NoSuchConfig()
		
		Out:
		
		The Config at set[index] if the index exists, but a NoSuchConfig exception otherwise.

void add(Config newConf)	Adds a configuration to the set. First sets a temp to the existing set, and creates a new set one element larger than the old one, fills it with temp, puts newConfig in the final position, and deletes temp.	In: set, setSize, newConf Work: Config *temp = set set = new Config[++setsize]

		for i in range(0, setSize - 1) set[i] = temp[i] set[setSize - 1] = newConf delete [] temp Out: Nothing. setSize and set have been changed according to the above code.
void empty()	Deletes the original set from the heap and intializes set to a new, empty set.	In: set, setSize Work: setSize = 0 delete [] set set = new Config[0] Out: Nothing. Sets setSize to 0 and points set to a new, empty array.
Private		
int setSize	Keeps track of the number of Configs in the set.	
Config set[]	Holds all the Configs of the set.	

Config class

Member	Description	Input/Work/Output
Public		

Config(bool ** arr, arrH, arrW)	Builds a configuration out of an array of boolean values, and sets the dimensions of the Config.	In: arr, arrH, arrW
		
		Work:
		
		configRows = arrH
		configCols = arrW
		
		configuration = new Config*[arrH]
		for i in range (0, arrH)
		  configuration[i] = new Config[arrW]
		
		for i in range (0, arrH)
		  for j in range (0, arrW)
		  configuration = Cell(arr[i][j])
		
		Out:

		Nothing, but configuration is initialized.
string get_name()	Returns the value of the name private member var.	In: name Work: return name Out: name
Cell ** get_cells()	Returns a pointer to the cells in the configuration.	In: configuration Work: return configuration Out: configuration
Private		
string name	Stores the name of the configuration.	
Cell ** configuration[]	Stores the cells of the configuration.	
int configRows		
int configCols		

Unnamed namespace

Member	Description	Input/Work/Output
Public		

int get_int(int lowBound, int highBound = 0)	Asks user for a number in range lowBound to highBound. If highBound is 0, maximum value is INT_MAX. If value entered is out of range or of the wrong type, asks for a new one.	In: lowBound, highBound, INT_MAX
		
		if highBound = 0,
		highBound = INT_MAX //include climits
		
		cin userInt
		While (cin.fail() ‖
		(userInt < lowBound) ‖
		(userInt > highBound) ‖
		(INT_MAX - userInt < 0) //while type wasn't right or value was out of range
		
		cin.clear(); //clear stream
		cin.ignore(numeric_limits<streamsize>::max(), '\n'); //ignore left over data
		print that input is invalid and must be int in range (lowBound, highBound)
		
		return userInt

class BoardStatic	Trivial class used to signal that the Board has stopped changing.	Nothing.
class NoSuchConfig	Trivial class used to signal that a nonexistent config was attempted to be accessed from the ConfigSet.	
Config defaultConfigs[]	A collection of 22 x 88 default configurations.	I don't know which these will be yet.
int NUM_DEFAULTS	Number of default configurations included with the library.	