#include "board.hpp"

#include "critter.hpp"

#include "yomu.hpp"

#include <random>

#include <iostream>

using std::cin;

using std::cout;

using std::endl;

void Board::initBoard(){

std::cout << "Would you like to choose the size of the board?" << std::endl;

std::cout << "1. Yes" << std::endl;

std::cout << "2. No" << std::endl;

int userChoice = intYomu(1, 2, "strict");

if (userChoice == 1) {

std::cout << "How big is the board? (Format: X by Y tiles)" << std::endl;

std::cout << "X: " << std::endl;

numberOfColumns = intYomu(20); //My input validator reads this as: "input must be 20 or greater to be valid"

std::cout << "Y: " << std::endl;

numberOfRows = intYomu(20);

customBoard = true;

}

else {

std::cout << "Okay, the board will be 20x20." << std::endl;

numberOfColumns = 20;

numberOfRows = 20;

}

// Each element in array is going to point to its own array of Critter\*

// + 2 is to accomodate a boarder around the game board

// USEABLE GAMESPACE IS FROM ROW 1 THROUGH NUMBER OF ROWS

// USEABLE GAMESPACE IS FROM COL 1 THROUGH NUMBER OF COLS

gameBoardPtr = new Critter\*\*[numberOfRows + 2]; // rows

for (int row = 0; row < numberOfRows + 2; row++)

{

gameBoardPtr[row] = new Critter\*[numberOfColumns + 2]; // columns

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* Setup border critters and start with empty grid

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

for (int row = 0; row < numberOfRows + 2; row++)

{

for (int col = 0; col < numberOfColumns + 2; col++)

{

if (row == 0 || row == numberOfRows + 1)

{

gameBoardPtr[row][col] = new Critter('-', row, col); // These critters are only used to display border

}

else if ((col == 0 && (row > 0 && row < numberOfRows + 1)) ||

(col == numberOfColumns + 1 && (row > 0 && row < numberOfRows + 1)))

{

gameBoardPtr[row][col] = new Critter('|', row, col);

}

else

{

gameBoardPtr[row][col] = new Critter(' ', row, col);

}

}

}

//customBoardDistX sets the range of random X output

std::uniform\_int\_distribution<int> customBoardDistX(1, numberOfRows);

//customBoardDistY sets the range of random Y output

std::uniform\_int\_distribution<int> customBoardDistY(1, numberOfColumns);

//Is this how we're going to set the position of each critter? "boardlocation[x][y] = critter"?

int testRow = customBoardDistX(theTwister);

int testCol = customBoardDistY(theTwister);

delete gameBoardPtr[testRow][testCol];

gameBoardPtr[testRow][testCol] = new Critter('X', testRow, testCol);

//randomSpawnPoint(testCritter);

}

**void** Board::randomStartingSpawns(**int** ant, **int** doodle)

{

**int** ants = ant;

**int** doodles = doodle;

**int** antCount = 0;

**int** doodleCount = 0;

**do**{

**static** randomSpawnPointX();

**int** randomRow = randomSpawnPointX();

**static** randomSpawnPointY();

**int** randomCol = randomSpawnPointY();

gameBoardPtr[randomRow][randomCol] = **new** Ant(**'O'**, randomRow, randomCol);

antCount++;

}**while**(antCount <= ants);

**do**{

**static** randomSpawnPointX();

**int** randomRow = randomSpawnPointX();

**static** randomSpawnPointY();

**int** randomCol = randomSpawnPointY();

**if**(gameBoardPtr[randomRow][randomCol]->getBugLetter() == **'O'**)

{

randomSpawnPointX();

randomSpawnPointY();

}

**else**

{

gameBoardPtr[randomRow][randomCol] = **new** Doodle(**'X'**, randomRow, randomCol);

doodleCount++;

}

}**while**(doodleCount <= doodles);

}

int Board::randomSpawnPointX()

{

if (customBoard)

{

std::uniform\_int\_distribution<int> customBoardDistX(1, numberOfRows);

return customBoardDistX(theTwister);

}

else

{

std::uniform\_int\_distribution<int> standardBoardDist(1, 20);

return standardBoardDist(theTwister);

}

}

int Board::randomSpawnPointY()

{

if (customBoard)

{

std::uniform\_int\_distribution<int> customBoardDistY(1, numberOfColumns);

return customBoardDistY(theTwister);

}

else

{

std::uniform\_int\_distribution<int> standardBoardDist(1, 20);

return standardBoardDist(theTwister);

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* Description: The printBoard function is used to print the board upon

completion of each step in the simulation. The 2D array is printed in

"square" format.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void Board::printBoard()

{

if (gameBoardPtr != NULL)

{

for (int row = 0; row < numberOfRows + 2; row++)

{

for (int col = 0; col < numberOfColumns + 2; col++)

{

cout << gameBoardPtr[row][col]->getBugLetter() << " ";

}

cout << endl;

}

cout << endl;

}

}

std::ostream& operator<<(std::ostream& outStream, Critter bugToPrint)

{

outStream << bugToPrint.getBugLetter();

return outStream;

}

bool Board::isCustom()

{

return customBoard;

}

int Board::getXLimit()

{

return numberOfRows;

}

int Board::getYLimit()

{

return numberOfColumns;

}

void Board::breedCritters() {

//loop through all board locations to and determine if critter is old enough to breed

for (int row = 0; row < numberOfRows + 2; row++) {

for (int col = 0; col < numberOfColumns + 2; col++) {

char getType = gameBoardPtr[row][col]->getBugLetter();

switch (getType) {

//if grid cell location contains an Ant

case 'O': {

//check if Ant age is equal to 3

if (gameBoardPtr[row][col]->getAge() == 3) {

bool spaceFound;

int returnRow, returnCol;

//call Critter breed function

gameBoardPtr[row][col]->breed(gameBoardPtr, returnRow, returnCol, spaceFound);

//if open space found, delete critter object in that position and replace with new Ant

if (spaceFound) {

delete gameBoardPtr[row][col];

//Critter object will need to be replaced by Ant object once class is created

gameBoardPtr[row][col] = new Critter('O', row, col);

}

//reset age to 0. This can probably be contained in breed() method as well

gameBoardPtr[row][col]->setAge(0);

}

break;

}

//if grid cell location contains a Doodlebug

case 'X': {

//check if Doodlebug age is equal to 8

if (gameBoardPtr[row][col]->getAge() == 0) {

bool spaceFound;

int returnRow, returnCol;

//call Critter breed function

gameBoardPtr[row][col]->breed(gameBoardPtr, returnRow, returnCol, spaceFound);

//if open space found, delete critter object in that position and replace with new Doodlebug

if (spaceFound) {

delete gameBoardPtr[row][col];

//Critter object will need to be replaced by Doodlebug object once class is created

gameBoardPtr[row][col] = new Critter('X', row, col);

}

//reset age to 0. This can probably be contained in breed() method as well

gameBoardPtr[row][col]->setAge(0);

}

}

//if neither cell contains Ant or Doodlebug, do nothing

default: {

break;

}

}

}

}

}

void Board::makeMove()

{

// First the doodlebugs will move

for (int row = 1; row <= numberOfRows; row++)

{

for (int col = 1; col <= numberOfColumns; col++)

{

if (gameBoardPtr[row][col]->getBugLetter() == 'X')

{

gameBoardPtr[row][col]->move(row, col, gameBoardPtr, numberOfRows, numberOfColumns);

}

}

}

// Then the ants will move

for (int row = 1; row <= numberOfRows; row++)

{

for (int col = 1; col <= numberOfColumns; col++)

{

if (gameBoardPtr[row][col]->getBugLetter() == 'O')

{

gameBoardPtr[row][col]->move(row, col, gameBoardPtr, numberOfRows, numberOfColumns);

}

}

}

// Reset all Critter's alreadyMoved to false for the next round

for (int row = 1; row <= numberOfRows; row++)

{

for (int col = 1; col <= numberOfColumns;col++)

{

gameBoardPtr[row][col]->setAlreadyMoved(false);

}

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* Delete Board when finished

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Board::~Board() {

for (int row = 0; row < numberOfRows + 2; row++)

{

for (int col = 0; col < numberOfColumns + 2; col++)

{

delete gameBoardPtr[row][col]; // Delete all Critter Type Objects

}

}

delete[] gameBoardPtr; // Free array of Critter pointer pointers

}