CS 162, Section 400, Fall 2018: Group Project [Predator Prey Simulation]

Group 18 Members: Christina Brasco, Christopher Gundlach, Russell James, Robert Saraceno, Amy Stockinger
Date Submitted: November 4, 2018

Table of Contents:

Class outline/design

- World (+ a bit of sample/pseudo code)
    - variables:
        - Critter*** critterSim
            - nullptr used for missing critters, check for nullptr in printing
        - int numAnts, numDoodles
        - int nRow, nCol (constant NUM_ROW, NUM_COL until we do extra credit, but if we want to leave that option open, we should program the whole thing with variables anyway, and set using constants initially)
    - functions:
        - World(nrow, ncol, numAnts, numDoodle) (initially we'll just call this with 20,20,100,5)
            - allocates Critter*** critterSim:
                - critterSim = new Critter* [nrow]
                - for (int i=0; i<nrow; i++)
                - critterSim[i] = new Critter*[ncol]
                - for (int j=0; j<ncol; j++)
                - critterSim[i][j]=nullptr;
            - place critters on the board
                - for (int i=0; i<numAnts; i++)
                - generate randX, randY
                - while (critterSim[randX][randY]!=nullptr)
                - regenerate randX, randY
                - critterSim[randX][randY] = new Ant(randX, randY);
                - for (int i=0; i<numDoodle; i++)
                - generate randX, randY
                - while (critterSim[randX][randY]!=nullptr)
                - regenerate randX, randY

        - critterSim[randX][randY] = new Doodlebug(randX, randY);
- void runIter() - runs one iteration of the simulation
    - call resetCritters()
    - moveCritters(DOODLEBUG)
    - breedCritters(DOODLEBUG)
    - starveCritters(DOODLEBUG)
    - moveCritters(ANT)
    - breedCritters(ANT)
- Void resetCritters
    - loop through all rows/cols of critterSim, and if the element is not a nullptr (indicating empty), dereference and call resetBool() which sets hasMovedThisTurn to false for every critter
- Void moveCritters(critterType)
    - loop through all rows/cols of critterSim, and if the element is not a nullptr (indicating empty), dereference to see if it matches inputted critterType, and call move()
    - if critter moved, set critterSim pointer at new location (stored in/obtained from Critter obj) to that critter's pointer and set oldXpos, oldYpos to nullptr
        - ie. critterSim[newXPos][newYPos] = critterSim[oldXPos][oldYPos]
        - old positions would be i, j (some loop iterators) and new positions would be returned by getX/getY
- Void breedCritters(critterType)
    - loop through all rows/cols of critterSim, and if the element is not a nullptr (indicating empty), dereference to see if it matches inputted critterType, and call breed()
    - if critter was bred, allocate correct critter memory at location of baby critter
    - Baby critters shouldn't move on day of birth because breed happens after move (?)
- Void starveCritters()
    - Loop through all rows/cols of critterSim, and if the element is not a nullptr (indicating empty), see if it is above its starvation limit (doodle's starve func rtn true?), and delete memory/set to nullptr if it is

- void printGrid()
    - print surrounded by borders (like langton's ant)
    - for each row/col: print appropriate char based on whether the pointer is null (print a space) or not
- void runSim(int nSteps)

        - call runIter steps times
- Critter
    - variables:
        - int x (position)
        - int y (position)
        - bool hasMovedThisTurn (represents whether this critter has been moved this turn)
        - int stepsSinceBreed (steps since last breeding)
        - int stepsSinceEat (steps since last feeding)
        - critterType getCritterType (return enum to indicate which type of critter the object is)
    - functions
        - int getX, getY, getStepsSinceEat, getStepsSinceBreed
        - void setX, setY
        - void resetBool() - sets hasMovedThisTurn to false
        - virtual ~Critter() {}
        - pure virtual functions that need to be overridden by Ant/Doodlebug:
            - virtual critterType getCritterType = 0;
            - virtual bool move() = 0; >> execute move, set hasMovedThisTurn to true, increments stepsSinceBreed/Eat
                - needs to receive values of adjacent cells (array of spaceValues enum values), update x and y according to where the critter should move
                - return true if critter moved, false if not
            - virtual direction (enum) breed() = 0 (should reset stepsSinceBreed)
                - needs to receive values of adjacent cells (array of spaceValues enum values), output space which will get a baby critter (output none if there is no breeding)
            - virtual bool starve() = 0;
- Ant (inherits from Critter)
    - variables:
        - no additional variables
    - functions
        - constructor (initializes variables and sets position of ant), copy constructor/assignment operator, destructor (all simple, no allocated memory here)
        - override bool move()
            - Execute move
            - Set hasMovedThisTurn = true
            - Increment step
            - Return true if moved
        - Override breed() as appropriate for ant move/breed logic

            - Check if steps = 3
            - Check adjacent cells
            - If adjacent cell open, spawn into (random choice if > 1)
            - Reset steps to 0
        - override starve with nothing
        - override getCritterType to return appropriate enum
        - ~Ant() {}
- Doodlebug (inherits from Critter)
    - variables:
        - no additional variables
    - functions
        - constructor (initializes variables and sets position of doodlebug), copy constructor/assignment operator, destructor (all simple, no allocated memory here)
        - override move()
        - breed()
            - Check if age = 8 ()
            - Check adjacent
            - If space available, spawn into (random if > 1 spaces)
            - Reset steps to 0
        - starve() as appropriate for doodlebug move/breed/starve logic
            - If lastMeal = 3
            - Die an agonizing death
            - Bool function - return true if starved
        - override getCritterType to return appropriate enum
        - ~Doodlebug() {}

General requirements as people code:

- everyone should keep track of:
    - constants they program with (update a constants.hpp file in our repo) or enums they create
        - enums to start with:
            - critterType (ANT, DOODLEBUG)
            - spaceValues (ANT, DOODLEBUG, EMPTY, OUTOFBOUNDS)
            - direction (NONE=-1, UP, RIGHT, DOWN, LEFT) [ordering allows for rotation by modular division, if it's ever appropriate)
    - tests they run
    - Any additional appropriate helper functions created (and update outline if necessary)

Project Timeline:

By 25-Oct: project code written

By 29-Oct: testing, menu/input validation incorporated, extra credit

By 31-Oct: plan/test table/reflection
By 3-Nov: style guide checked
4-Nov: submit

Test Case/Requirement	Target Function(s)	Input	Expected Result	Observed Result
Ants placed in random cell	World constructor	rows: 20, cols:20, numAnts: 100, numDoodles:5	ants and doodles placed in random rows. Different setup each time.	Ants and doodles are placed in random rows and the configuration is different for each setup
Ants and doodles do not occupy the same cell	World::move, setAdjacents, Ant/Doodle::move()	rows: 3, cols:3, numAnts: 4, numDoodles:1	Ants and doodles will move according to specs.	Ants and doodle move according to specs. Doodle will eat the ant and ant removed, but otherwise no conflict with occupying same space.
Ant does not move off grid	move(), setAdjacent	rows: 3, cols:3, numAnts: 4, numDoodles:0	Ants will fill the board, but will not cause seg faults	no seg faults observed, successful valgrind with no mem leaks
Ant will move to unoccupied space	world::move, ant::move	rows: 3, cols:3, numAnts: 1, numDoodles:0, steps: 10	Ant will move for 3 steps, then breed	Ant moves right, then left, then up and breeds down (or no move and breeds up). Moves into empty spaces.
Ant will not move to occupied space	world::move, ant::move	rows: 3, cols:3, numAnts: 9, numDoodles:0, steps: 10	board will be full and ants will remain in cells without moving or causing seg faults	Ants stay in cells, no moves made. no seg faults.
Ant movement is random	ant::move, world:: move()	rows: 5, cols:5, numAnts: 1, numDoodles:0, steps: 8	ant will move in different directions over several iterations	step 1: up, step 2: left, step 3: left, then breeds down, step 4/5; no move, step 6: ant 1 move up, ant 2 move right, step 7: ant 1 move up, ant 2 moves down; step 8: ant 1 move up, ant 2 move right, ant 3/4 no move.

Test Case/Requirement	Target Function(s)	Input	Expected Result	Observed Result
not all Ants move in the same direction on the same step (indicates random seeding done correctly/not too often)	see test above (ant movement is random)			
New ant produced after 3 steps	Ant::breed(), World:: breedCritters()	initialize board with 1 ant and 0 doodles, 5 rows, 5 columns, run sim 4 steps.	should produce another ant into open space after 3 steps	Second ant appears on printed board for step 3. This is correct, the board is printed after all actions completed for the "day".
ant will breed into random available space.	Ant::breed(), World:: breedCritters()	initialize board with 1 ant and 0 doodles, 5 rows, 5 columns, run sim 4 steps. Repeat 10 times. Test through gdb to see if breed direction matches newly created ant.	Should produce a new ant in different directions.	1st run, new ant created either left or right of original ant. 2nd run, new ant created to left of parent. 3rd run, new ant created below parent. 4th run, new ant created left of parent. 5th run, new ant created left of parent. 6th run, new ant created to right of parent. 7th run, new ant created above parent. 8th run, new ant created below parent. 9th run, new ant created below parent. 10th run, new ant created left of parent.
If only 1 space available for breed, ant will breed into that space.	Ant::breed(), World:: breedCritters()	initialize board with 2 rows, 2 cols (4 spaces), create 3 ants and 0 doodles, iterate sim 4 steps.	should produce new ant in the only available empty space.	The available space is filled with an ant at the end of step 3. All 4 spaces are filled.

Test Case/Requirement	Target Function(s)	Input	Expected Result	Observed Result
If ant unable to breed, will breed when space available.	Ant::breed(), World:: breedCritters()	board: 3 cols, 3 rows. 7 ants, 1 doodle, iterate for 5 steps. Run sim twice.	Hope to see an ant produce on 4th or 5th step (not divisible by three). Board is sufficiently full to prevent breeding of some ants due to all spaces being filled, but will spawn when space made available.	5 ants on board at end of step 2. One ant dies in step 3 (down to 4 ants), three new ants are spawned (back up to 7 ants, one did not breed) by end of step 3. One ant eaten in step 4 (down to six ants) and one is born (back up to seven ants). A breed occurred in step 4. Ant eaten in step 5, no breeds occur, down to 6 ants.
New ant produced after 3 additional steps (will breed for each 3 steps taken after previous breed)	Ant::breed(), World:: breedCritters(), Critter:: resetBreed()	board: 9 cells. 1 ant initialized, 0 doodles, 6 steps.	ants should breed after 3 steps, and again after 6 steps. Should end up with 4 total ants.	Ants double from 1 to 2 after step 3. Ant population doubles again after step 6, from 2 to 4 ants.
ant will breed into inner/outer most colums, upper/lower rows	Ant::breed(), World:: breedCritters()	board; 4 cells, 1 ant. 4 steps. 5 tests.	should breed into an available cell. All cells in a 2x2 grid are part of a lower, upper, left, or right limit.	1st run: Ant bred into upper or lower limit cell. 2nd run: ant bred into lower-right extreme cell. 3rd run: ant bred into left extreme, lower or upper extreme cell. 4th run: ant bred into lower/left extreme cell. 5th run: ant bred into upper or lower extreme, and left extreme.
Doodles placed in random cell	World constructor	rows: 20, cols:20, numAnts: 100, numDoodles:5	ants and doodles placed in random rows. Different setup each time.	Ants and doodles are placed in random rows and the configuration is different for each setup

Test Case/Requirement	Target Function(s)	Input	Expected Result	Observed Result
Doodle move favors cells with ants.	doodle::move()	rows: 3, cols: 3, ant: 1, doodle 1, steps 4	if doodle is next to ant, will eat ant on next step	step 0, doodle below ant; step 1, doodle eats ant. Dies on step 4.
Doodle moves to unoccupied space	doodle::move()	rows: 3, cols: 3, ant: 0, doodle 1, steps 4	doodle will move to open spaces	step 1, no move (in top row), step 2: move right to open space; dies step 3
Doodle does not move into occupied space	doodle::move()	rows: 3, cols: 3, ant: 0, doodle 9, steps 4	doodles fill the board, will die on step 4	step 0 thru 2, doodles fill the board. Die after step 3. No seg faults, no moves made.
Doodle does not move off grid	doodle::move()	rows: 3, cols: 3, ant: 0, doodle 9, steps 4	doodles fill the board, will die on step 4	step 0 thru 2, doodles fill the board. Die after step 3. No seg faults, no moves made.
Doodle will move onto edges of grid.	doodle::move()	rows: 3, cols: 3, ant: 0, doodle 1, steps 4	doodle will move into upper, lower, left and right most rows	step 0: bottom left cell; step 1, no move; step 2: moves right to remain in bottom cell, then dies. 2nd test - starts in right most cell, no moves made. 3rd test - starts in right most cell, then makes two moves to left most cell.
Doodle will not eat another doodle	doodle::move()	rows: 3, cols: 3, ant: 0, doodle 9, steps 4	doodles fill the board, will die on step 4	step 0 thru 2, doodles fill the board. Die after step 3. No seg faults, no moves made.
Doodle will produce new doodle after 8 steps.	Doodlebug::breed(), World::breed()	board: 4 cells, init 3 ants, 1 doodle; 8 steps	If doodle survives for 8 steps, should spawn another doodle at end of step 8.	1 doodle on the board after 8 steps. In step 8, one doodle spawned, then died, so only 1 was visible when board printed.

Test Case/Requirement	Target Function(s)	Input	Expected Result	Observed Result
Follow up: Doodle will produce new doodle after 8 steps	Doodlebug::breed(), World::breed()	board: 4 cells, init 3 ants, 1 doodle; 9 steps	If doodle survives for 8 steps, should spawn another doodle at end of step 8.	1 doodle on board after 9 steps. ~~Should be 2~~ One doodle spawned and starved during step 8, so only 1 was visible when grid was printed.
Doodle will breed again if survives for 8 steps after previous breed	Doodlebug::breed(), World::breed()	board: 25 cells; init 20 ants and 2 doodles; iterate 17 steps	If doodles don't starve. Should see doodles born after step 8 and again after step 16.	Doodles increase from 2 to 4 in step 8. Doodles increase again from 3 to 6 in step 16.
Doodle will breed into random empty cell	Doodlebug::breed(), World::breed() - breedDir variable, adjacents variable	board: 25 cells; init 20 ants and 2 doodles; iterate 17 steps. Test through gdb, break set at Doodlebug:: breed(), print return vals from breed function.	should return 1 of the available open cells to breed into.	step 8, doodle 1: Open spaces right and down, breed returns direction RIGHT; step 8 doodle 2: open space right and down, breed returns direction RIGHT. step 16, doodle 1: Open space down only, breed returns direction DOWN. step 16, doodle 2: all spaces available, breed returns direction DOWN. Step 16, doodle 3: available spaces up and left, breed returns LEFT.
Doodle will breed into empty cell if only 1 adjacent cell is available.	Doodlebug::breed(), World::breed() - breedDir variable, adjacents variable	board: 25 cells; init 20 ants and 2 doodles; iterate 17 steps. Test through gdb, break set at Doodlebug:: breed(), print return vals from breed function.	should breed into the only available cell	See above. Step 16, doodle 1: down is only open space, breed returns DOWN. Run again, step 8, doodle 2: left is only space available, breed returns LEFT

Test Case/Requirement	Target Function(s)	Input	Expected Result	Observed Result
If doodle doesn't breed, it will breed when space is available, unless it starves first.	Doodlebug::breed(), World::breed() - breedDir variable, adjacents variable, Driver function World:: breedTest()	board: 400 cells; init 0 ants and 300 doodles, 10 steps	driver function increments sinceBreed to 8, so doodles should breed on first step andfill the board. Doodles with no space to breed will not breed.	After step 1, 400 doodlebugs fill the board.
not all Doodles move in the same direction on the same step (indicates random seeding done correctly/not too often)	move functions	board 25 cells with 4 doodles, 4 steps	Should observe doodles moving in different directions during the same turn.	step 1: doodle 1 moves right, doodle 2 moves down, doodle 3 moves left, doodle 4 moves right. step 2: doodle 1 moves up, doodle 2 moves up, doodle 3 moves right, doodle 4 moves left. step 3, doodles move then die.
World Destructor correctly de-allocates memory	World::~World()	main() calling ~World() at termination	valgrind is happy after the end of the program	All heap blocks freed, no leaks possible
World constructor correctly allocates memory, places critters	World::World(), printGrid()	main() calling World(), printGrid()	see tests above	grid created correctly and destroyed as well. no leaks possible.
No critters are allowed to overlap with each other	World::World(), printGrid()	main() calling World(), printGrid()	in the case where nAnts + nDoodles = number of spaces, all spaces on board are filled, with no overlap	see above tests. Working correctly

Test Case/Requirement	Target Function(s)	Input	Expected Result	Observed Result
printGrid() prints grid in correct orientation	printGrid()	printGrid() on a 5x10 and a 10x5 board	provided number of rows/columns should match output, and not be mixed up	Board is constructed correctly

Group 18 Reflection

Christina Brasco, Christopher Gundlach, Russell James, Robert Saraceno, Amy Stockinger

11/4/2018

Design Changes

An initial design meeting was held between the group members and ideas for the project were discussed in a teleconference format. The general approach was confirmed and one group member started an outline in a shared document. The rest of the team reviewed the outline, edited and commented as necessary, and a formal program plan was established. The work was divided among the group members and each started their portion of coding.

The group had the idea that the move and breed functions would accept, as parameters, the status of the adjacent cells. It wasn't initially clear exactly how this information would be passed to the function so members programming for these functions took their own ideas and programmed accordingly. As a result, some of the functions had to be tweaked for compatibility.

Work Distribution

Different parts of the program were distributed among the individual members. The programming was roughly divided into five equal parts:

- Class outline creation; Ant and doodlebug move functions
- Creation and display of 2D array, constructor, destructor, random placement of critters
- Breed functions in critter and world, bounds checking function
- Move functions in World class, iteration function (daily sim function), bounds checking
- Menu functions

All group members tested their respective functions and actively participated in the design process.

Problems Encountered

The setAdjacent function had a problem with the bounds checking. It was allowing a pointer to dereference memory beyond the boundary of the board array. The conditional statements were incorrectly setup for the maximum number of rows and columns, resulting in segmentation faults when boundary checking was performed on the highest row of the 2D array.

The setAdjacent bounds checking function had also transposed the row and column array elements when checking for the elements left of the critter being acted upon.

The seg faults were immediately apparent, but the left bounds checking error was only discovered while searching for the culprit of the seg faults.

The Critter::resetBreed function had a typo that was incorrectly resetting the stepsSinceEat function to zero instead of the stepsSinceBreed function.

There was a bug in the menu function that would allow the user to choose a maximum number of ants that did not complement the minimum number of doodlebugs allowed. After choosing the ants, there was no compatible choice of doodlebugs that would permit the user to progress to the next prompt, resulting in an unending loop of input validation.

Lessons Learned

The project came together fairly quickly once it was started thanks to the thorough design plan. The group was careful to avoid any circular dependencies in the design, as well as keeping coding style relatively constant throughout. Speaking as just one member of the group, I was impressed by some of the approaches that were taken when handling certain functions. Communication was handled pretty well considering the time crunch, the fact that this project is over midterms week, and the fact that students may not live in similar time zones. The team took practical approaches the solving problems, and operated fairly using the principle of majority rules. Overall, working through this project as a group was an invaluable learning experience.