Project 1: Langton's Ant Simulator

Nicole Reynoldson

Ant Class - will store all the information regarding the ant's location. Responsible for moving the ant, changing direction and coordinating ants movement with the state of the board.

Member Variables	Member Functions
● X and Y position ● Previous x and y position ● Color of square ant is presently on ● Direction ● Board	**Constructor** ● Should take x, y, rows and columns ● Call the board constructor to make the matrix ● Set x and y start position of the ant ● Call board function to set the square color and place the ant on the board ● Set previous x and y to current to initialize ● Set the color to white to initialize the value it holds **Move forward** ● If direction is north, check if y position is 0 ○ If at edge, reset the y position to the rows - 1, else decrement the y position ● If direction is east, check if x position is equal to the columns -1 ○ If at edge, reset the x position to 0, else increment the x position ● If direction is south, check if y position is equal to the rows -1 ○ If at edge, reset the y position to 0, else increment the y position ● If direction is west, check if x position is 0 ○ If at edge, reset the x position to columns - 1, else decrement the x position **Change direction** ● If the color is black, turn ant to the left ● If the color is white, turn the ant to the right **Makemove function** ● Move ant forward (wrap or not) ● Set square of board at previous x and previous y to opposite color ● Change color to the color at new location ● Change matrix at x and y to the ant ● Change direction ● Display board

Board Class - responsible for dynamically allocating the board, initializing the board to white and storing characters as the board is updated through ant object.

Member Variables	Member Functions
● Rows ● Columns	● Board constructor ○ Dynamically allocate the board based on rows and columns ○ Will initialize the board to empty ● Set matrix ○ Takes an x, y coordinate and a char to change a location on the board ● Get matrix ○ Return the char at a given location ● Display board ○ Will print the board to the screen ● Destructor ○ Free memory used to dynamically allocate the board

Input Validation function

getInput()

- Takes two ints as parameters for maximum and minimum range
- Gets input in the form of a string
- strtol() function tries to convert the string to an integer
    - Pointer to char passed in as parameter will point to first char that is not a digit
    - If the pointer is anything but the null character, then it will reprompt
- If the string is converted, then check that the value is within the range specified by parameters
    - Continue reprompting as long as value is out of range
- Return the value converted to an int

Menus

Two functions for start and end menus

- Should not take any parameters and should return no values
- Print options to the screen

Random Location function

- Takes two ints as parameters, the max and min value that can be generated
    - 0 to the max rows or columns
- Returns the randomly generated number
- Uses rand() function and time() for seed from <ctime> and <cstdlib>

Main

Display the start menu
Get the input from input validation function
If the choice is quit, set simulation state flag to false and exit program

While simulation state flag is true

- Prompt user for each setting
- Get choice for each setting from validation function and store in local variables
    - Set maxSteps from user input, initialize current steps to 0

- Ask if user wants random location or not
    - If random location, call the random location function and store in local variable for X and Y position of ant
    - Else have the user enter values for the local variables X and Y
- Set up an ant object passing in local variables that were set with user input
- While the steps are less than or equal to maxSteps
    - Display direction
    - Call the Makemove function
    - Display the board

- Display the end menu
- If not playing again set flag to false

Thank user for playing
Return from main

	Test	Expected outcome
Board Generation	● Attempt calling board with varying integer values/non-integer inputs for rows and columns ● Test square and rectangle configurations to ensure rows and columns appear correctly and are not swapped	● Non-integer values are rejected. Values less than 1 or greater than 200 are rejected ● Input gives correct rows and columns ● Board is initialized and set to all white
	Possible test values for both rows and columns: 200, 1, 150, 0, gh7, 0938b, 8 ht, 10 9, 2, 1, -4, 6.0, 5.2, 8\ hello, HI, 1000	**Actual outcome**: as described above
Ant Generation	● Choose two options for location starting type: user input or random. Test non-integer values and integer values outside of options ● Enter different value for row than entered for column starting position ex. Row: 20 & column: 30	● Input for choice should be rejected if outside of the range of choices or a non-integer value For random: ● Running program several times should give different starting position each time ● Ant should not be able to start off the board (position generated is less than or equal to number of rows or columns and greater than 0) For user input: ● Non-integer values are rejected. Values that are greater than the rows or columns on the board are rejected. ● The x and y position of the ant should be correctly updated and reflected accurately on the board by '*' ● Previous x and y position of the ant should be initialized to the start position
	Possible Test Values: **Menu choice**: 1, 2, 150, tyn, 0938b, 8 ht, 2, 7.8, -4, hello, HI, 10000, 10 9, 8\ **User Input X and Y:** 200, 1, 0, 2, 10, 150, tyn, 0938b, 8 ht, 2, 7.8, -4, hello, HI, 10000 **test 10th row 5th column for example to see that x and y values are on correct axis	**Actual outcome**: as above
Ant Movement	● Test each possible scenario for ant positioning by manipulating start position ○ 4 corners of the board ○ Edges of the board (N, E, S, W)	● Input for simulation steps should be rejected if less than 0 or outside upper bound of 30000 ● When ant is away from the edge, it should move forward one space in the direction it's facing and be updated on the board

	○ Middle of the board ● Run simulation with a low number of steps and large number of steps (10, 52, 12000 etc.) ○ low number of steps to hand trace and ensure that ant is following the correct behaviour	● The color of the square the ant was previously on should be updated to opposite color ● Color of the square the ant has moved to should be stored; expect the ant to turn left if the square is black and turn right if white ● Steps should update for each move ● If the ant is on an edge or in a corner, expect that the ant will wrap to the other side of the board ○ If moving east or west, only column number should change ○ If moving north or south, only row number should change Expect different shapes to appear when ant is placed in the middle of the board for different number of steps ● Heart appears at step 52 ● Highway pattern should begin appear after around 10,000 steps
	Test values: 10x10 board, ant at (1,1) 10 x10 board, ant at (10,10) 10x10 board, ant at (10, 1) 10 x10 board, ant at (1, 10) 10 x 10 start at N edge, S edge, W edge, E edge 10 steps for each Test small board to observe behaviour on edges: 4x4 board, ant at (2,2) - 30 steps and trace Test large board 80 x 100 for 20,000 steps	**Actual outcome:** ● Two ants '*' appearing on the board - moveForward() originally written to so that only one of the previous coordinates was updated (ex. Ant moving east, so prevX is updated to X, but y position is not updated) causing sync issues with printing board which then causes abhorrent behaviour - rewritten to update both X and Y positions at the beginning of each step
Entering Simulation	User has a choice of start or quit: ● Test start ● Test quit	● Start should launch into the settings menu ● Quit should safely exit the game and display the goodbye message ● Should reject any non integer values or values besides 1 or 2
	Possible test values: 1, 2 150, 0, gh7, 0938b, 8 ht, 10 9, -4, 6.0, 5.2, 8\ hello, HI, 1000	**Actual outcome**: as described above
Exiting Simulation	User has a choice of play again or quit ● Choose quit ● Choose play again and do a full play through	● Should reject any non- integer values or values besides 1 or 2 ● Quit should safely exit the game and display the goodbye message ● Play again should launch the game again from the settings menu ● Ensure all values should be reinitialized ○ Follow all test procedures above for second playthrough
	Possible test values: 1, 2 150, 0, gh7, 0938b, 8 ht, 10 9, -4, 6.0, 5.2, 8\ hello, HI, 1000	**Actual outcome**: as described above

Reflection

When I first began designing my program, I had the idea that I would create the ant class, and place an ant object within a board class. I wanted the ant class to be able to keep track of all of the movements of the ant and its location. The board class was intended to dynamically allocate space for the board and interact with the ant to update positions and display them to the screen. Because the ant has to do so many more operations than the board, I quickly decided that it would be easier to place the board object inside the ant class instead of the other way around. This prevented me from having to write many setter and getter functions for the ants locations and also helped prevent my code from becoming very clunky and hard to read.

To handle edge cases, I originally wanted my ant to rotate until it was no longer in a situation where it could be moved off the board. The logic broke down when I tried testing corner cases. I had my ant continue rotating in the same direction as the current square color dictated until it was not at risk of moving off the board. This led to the ant rotating in one direction and back in the other direction so that it never left the corner. I considered playing around with having the ant rotate in a random direction at the edge, but thought this might alter the behavior of the ant too much. I decided to go back to my design plan and have the ant wrap around the board. This ended up being a very simple fix to just give two options in my moveForward() function for each possible direction and as it turns out, helped make the highway pattern much clearer.

For my input validation, the original idea was to have a function that ensured the input was of the correct type and another that would check that the value was in range. When I started implementing my code, I realized that I was overcomplicating and could get input, validate its type and check that it was within a range in the same function just by taking two integers as a minimum and maximum for the range of values.

I decided to use a constructor to initialize all of the member variables rather than using setter functions. I did not want to introduce any public functions that could change aspects of the ants orientation/position and felt that using a constructor would be safer for obeying the rules for each step. My makeMove() function completes all the actions necessary when the ant is taking a step and I realized as I was coding this section that it made sense to keep the separate pieces of the movement private as well. None of these pieces should ever need to be called in main since makeMove() takes care of it, and doing so would mean that adequate steps are not taken to update the board.