Clinton Hawkes
CS162-400
Reflection Document

Design:

Program will have a main function, an Ant class, a Board class, a menu function(s), and a validation function(s).

1) Main function – gets data input from the user for the initial setup of the And and Board classes. Mainly used to call functions and
   A) Create a bool variable “State” to hold the current state of the simulation
   B) Call the menu function to display the initial menu
      I. Ask user if they want to begin simulation or quit
         a) If begin simulation, return true to the main function
         b) If quit, return false to the main function
   C) If “State” is false, return 0
   D) Create Ant object
   E) Create Board object
   F) Create a while loop that loops until the state variable is false
      I. Ask user how many rows they want the game board to have
         a) Set Board variable for rows equal to input
      II. Ask user how many columns they want the game board to have
         a) Set Board variable for columns equal to input
      III. Ask user how many iterations they want the simulation to perform
         a) Set Ant variable for number of steps equal to input
      IV. Ask user if they want the Ant to start in a random location
         a) If yes, seed the rand() and assign one random number to the Ant’s x location variable, and then assign another random number to the Ant’s y location variable
         b) If no
            i. Ask user what row they would like the Ant to start on
               1) Set Ant’s x location equal to user input
            ii. Ask user what column they would like the Ant to start on
               1) Set Ant’s y location equal to user input
      V. Call Board function that creates the game board
      VI. Create while loop that loops until number of iterations is reached
         a) Call Board function that prints out the game board
         b) Call Ant function that changes its orientation

c) Call Board function that changes the color of the cell

d) Call Ant function that advances the Ant

VII. Call secondary menu function to ask user if they want to continue or quit

a) If quit, return false back to the state variable in main

b) If continue, return true back to the state variable in main

VIII. If state is true, call Board destructor to clear the game board before beginning the simulation again.

Ant Class: Used to keep track of variables pertaining to each Ant object, change the orientation of the Ant, and change the current position of the Ant.

Variables:	int antXCoord antYCoord orientation numOfSteps	Functions:	getXCoord setXCoord getYCoord setYCoord getOrientation setOrientation getNumSteps setNumSteps advance changeOrientation

I. Ant::advance() - changes Ant's coordinates

a) Use orientation variable in a switch statement (0=west, 1=north, 2=east, 3=south)

i. Case 0: antYCoord--

ii. Case 1: antXCoord--

iii. Case 2: antYCoord++

iv. Case 3: antXCoord++

b) Check bounds of board against Ant coordinates. If a coordinate is off the board, move it to the other side of the board.

II. Ant::changeOrientation() - changes the direction the Ant if facing

a) If Ant is on a ' ', orientation = (orientation+5)%4 (*eliminating negative values)

b) If Ant is on a #, orientation = (orientation+3)%4 (*eliminating negative values)

Board Class: Used to keep track of variables pertaining to every Board object.

Variables:	int numRow numCol char** board	Functions:	getNumRow setNumRow getNumCol setNumCol createBoard printBoard changeColor

1) Board::createBoard() - dynamically creates a 2d array of chars for the game
   A) Point char** board to an array of pointers the size of numRow
      I. Create an array of chars for each pointer in board. Each array of chars should be the size of numCol
   B) Fill the 2d array of chars with spaces

2) Board::printBoard() - prints out the game board
   A) Loop for the number of rows
      I. Loop for the number of columns
         a) If current position is same as the Ant
            I. Print *
         b) Print board[row][col]

3) Board::changeColor() - changes the color of the board where the ant is located
   A) If current position is ‘ ‘
      I. Change to #
   B) Else
      I. Change to ‘ ‘

Menu Function: The menu functions print out greetings, instructions and options to the user. The functions will get input from the user and return the appropriate bool to the main().

Validation Function: The validation function will receive input from the user, check that the input is the correct data type, check if the input is within the specified range, and if it fails these test, the user will be prompted to enter a valid input. The validation functions will return a value of the indicated type back to the main() after all the tests are passed.

Test Plan for yesNoValidation()

Test Case	Input Value	Expected Result	Observed Result
Char (not y or n)	A, c, +, .,	“Please enter yes or no.”	“Please enter yes or no.”
String (not beginning with y or n)	Four, .five, go, .no	“Please enter yes or no.”	“Please enter yes or no.”
Float	6.31, 07.0, 8.8	“Please enter yes or no.”	“Please enter yes or no.”
Integer	1, 121, 0	“Please enter yes or no.”	“Please enter yes or no.”
String (beginning with y or n)	Yes, yes, You, No, no, not	Function accepts input if first letter in string is upper/lower y or n. Returns appropriate result to main()	Function accepts input if first letter in string is upper/lower y or n. Returns appropriate result to main()
Char (y or n)	Y, y, N, n	Function accepts input if char is upper/lower y or n. Returns appropriate result to main()	Function accepts input if char is upper/lower y or n. Returns appropriate result to main()

Test Plan for integerValidation()

Test Case	Input Value	Expected Result	Observed Result
Char	A, c, +, .	“Please enter a valid integer.”	“Please enter a valid integer.”
String	Four, .five, yes	“Please enter a valid integer.”	“Please enter a valid integer.”
Float	6.31, 07.0, 8.8	“Please enter a valid integer.”	“Please enter a valid integer.”
< Min (set at 5)	0, 4, -5	“Please enter a valid integer.”	“Please enter a valid integer.”
> Max (set at 200)	201, 1005	“Please enter a valid integer.”	“Please enter a valid integer.”
Lower limit (min=5)	5	Function accept and return value to main()	Function accepted 5 and returned 5 to main()
Upper limit (max=200)	200	Function accept and return value to main()	Function accepted 200 and returned 200 to main()

Test Plan for Langton’s Ant

Test Case	Description	Expected Result
Board Creation	1. Call createBoard() function after receiving size specs from the user. 2. Rows=10 and Cols=30 3. Call printBoard() function to display the newly created board.	Board should be dynamically created and contain 10 rows/30 columns. Placement of the Ant should match the Ant’s X and Y coordinates.
Ant Movement	1. Set number of moves to be 100 2. Set number of rows to 10 3. Set number of cols to 10 4. Set ant start row to 5 5. Set ant start col to 5	Ant should turn right 90 degrees if it is on a white square and advance. Ant should turn left 90 degrees if it is on a black square and advance. Ant should move forward in the correct direction base on its current orientation. Ant should move to the opposite side of the board when it encounters a wall.
Board Color	1. Set number of moves to be 100 2. Set number of rows to 10 3. Set number of cols to 10 4. Set ant start row to 5 5. Set ant start col to 5	Board square should change color after the Ant leaves each square.
Menu 1	1. Choose option to quit 2. Choose option to play	Option to quit exited the program. Option to play started the simulation.
Menu 2	1. Choose option to quit 2. Choose option to play	Option to quit exited the program. Option to play started the simulation.
Memory Leak	1. Start and run program using valgrind. Play simulation 3 times without exiting program between. Use different board sizes each time the simulation is run. Exit 2. Start program and immediately quit.	There should be no memory leaks for either of these cases.
Board Size	1. Run program rows=5 and col=5 2. Run program rows=100 and col=80 3. Run program rows=150 and col=150	Board should be created according to spec in each of these cases. Program should not crash at anytime during use.
Iterations	1. Run program with 5 iterations	Program should run without crashing for each

	2. Run program with 25000 iterations 3. Run program with 12000 iterations	of these cases.

Reflection:

After spending the past couple weeks working on Langton's Ant problem, I can definitively say this project has been my favorite and most rewarding in the program thus far. It hasn't been easy for me, mind you. There where times when I had no idea how to proceed for hours at a time. I encountered roadblocks and frustration here and there, but the times when I felt on top of the world after solving the problems I was facing made up for the hassle. All in all, I learned a lot and I enjoyed doing it. Now, time for the details.

For this reflection, I will try to group my narrative into the following topics: Issues encountered and changes made.

Issues Encountered:

The first issue I encountered during this project was finding the desire to create a design and test plan before coding. Every time I start a new coding project, I have to force myself to sit down and plan out. I always feel the need to hop on vim and begin coding before I even know what I am going to code. I know coding before planning is not efficient and typically turns out rougher code as projects get larger, but I can't seem to break the habit. I'm glad this course emphasizes the need to create a design/test/reflection document before coding, because I hope it will give me the push needed to change my ways.

Another issue that I encountered was memory leaks. This one had me stumped for a bit. I would run my program one time through and end the program with no leaks. I then ran the program a couple times without quitting between runs and experienced leaks upon termination. I soon found out that I was not deleting the allocated memory before running the program again, so I scrolled to the end of my while loop and typed a line of code that calls the destructor after each loop. I thought this would do the trick, but it actually caused some problems of its own. I was now deleting more memory than I had allocated dynamically. I discovered that I needed to type an if statement that explicitly calls the Ant destructor only if the user chooses to play again. If the user chooses to quit, the destructor would be called on its own when the Ant object went out of scope. If I did not type the if statement, the destructor would be called twice when the user quit. Once explicitly when the while loop ended and again when the Ant object went out of scope. This is what caused the issue of too much memory to being deleted. This issue taught me a lot.

Last but not least, input validation. I spent much more time on input validation than I thought I was going to need. I tried several approaches, but none of them were successful at weeding out the invalid input. After searching online for a couple hours, I finally found a post on StackExchange about using stringstream to validate data. It sounded promising, so I gave it a shot. It worked! All the invalid inputs were rejected, and I have a validation function that I can use in future projects. It is such a relief to have this function in my arsenal now.

Changes Made:

I had originally planned to use an Ant class and a Board class to implement Langton's Ant. The further I got into the project, the more I started to doubt my approach of using two different classes. By combining the classes into one class, I would able to avoid using two different types of class

objects for very closely related operations. I concluded that the Board and Ant classes did not differ enough to warrant a class for each of them, so I combined them into one class named Ant.

Another change I made along the way was to breakup the validation function into multiple functions. The function used depends on the type of data being validated, so if I need to validate an integer, I would use the integerValidation function. If I needed to validate a yes or no response, I use the yesNoValidation function. Breaking the validation function up into separate functions allowed the code to be clean and concise, because there are few conditions that need to be checked if the function is only working with one data type.

During the planning process, I thought I only needed one menu function for the project. I wasn't thinking about the additional code that would be required to work in the way I needed. By making a menu function for the main menu and another menu function for the secondary menu, I was able to keep the code in the functions simple and clean. Best of all, they are both reusable.

This concludes my reflection of project 1. I don't remember everything that occurred while completing the project, but I believe the I included the main points. Moving forward, I think I will keep a log while working on the projects so I don't forget any details. It should make the project design document easier to complete as well. I look forward to the next challenge in this course.