<u>Reflections Document – Assignment 1 – Pedro Torres Picón</u>

Initial design

Below is my first pass at a pseudocode design for the Langton's Ant program. Before writing the pseudocode I worked on a piece of paper and tried to figure out how many files I would have and how the program would work towards the user.

```
ask user for number of rows/cols for the matrix
// make all game boards squares for simplicity
// suggest more than 10 rows/cols so the ant has room
ask user for number of moves the ant will make
// suggest more than 100 and more than 10000
ask user for starting position for the ant, or just hit
enter if random

create a matrix with user defined dimensions
create new instance of ant passing it the matrix and
starting location

loop as many times as the user entered and do the following
  clear the console
  display the matrix with current ant position
  the ant checks what the color of the current tile is
    if the tile is white, turn right 90 degrees
    if the tile is black, turn left 90 degrees
  change the color of the current tile to opposite color
    check if moving one square forward is legal and in
bounds
    if move is in bounds move one square forward
    if move is not in bounds, end the loop and show a
messag saying the ant fell over the edge
```

Ways things changed or improved in the design/pseudocode stage

- At first, I thought I would store the color of a current tile in a tileColor int variable, where the color could be 0 or 1. I then changed to a bool type because I thought it would be easier to hold the two values. As I developed the program I realized there needed to be three 'colors' or states, not two: white, black and ant. I changed back to int values and made them 0, 1 and 2. I then created a displayBoard() function that took those three values and displayed a different character for each one. i.e. blank space for 0, # for 1 and * for 2
- On a first pass at the Ant class, I declared moveLeft and moveRight functions that would be invoked depending on the color of the tile the ant is on. Then, I bundled them up in a consolidated move() function that I thought would make things easier. But as I developed the program further I ended up splitting that move() function up into turn(), flipTile() and moveForward().
- Initially, I the coordinates of the starting position being passed to the Ant class when created, and the matrix being created internally within the Ant

class. Then I switched to creating the matrix in main and passing a pointer to it to the Ant class when created. That way I could use the matrix outside of the Ant class to display it, etc.

- At first, I thought I would use an array to pass the starting coordinates to the Ant function, but ended up just passing two int parameters for x and y to make things simpler
- When I first created the Ant class I thought I would pass the number of steps would be to the class when it was created, and it would just move. Towards the end, I decided to put the loop in the main function and just have the necessary functions accessible on the Ant class to make the move happen.
- The original loop to make the ant move was just constrained by the max number of moves the user entered, but I added an 'out of bounds' flag that would cause the loop to end if the ant falls out of one side.
- I had created a nextMoveInBounds() function to check if the next move was legal, but that ended up being too much atomization so I incorporated that functionality back into moveForward()

Problems/decisions I ran into and resolved in the coding stage

Should I use a delay to slow down the display? I initially thought it was a good idea to use a delay so that the user could more clearly see each move made by the Ant. I researched a bunch of ways to introduced delays and learned it is not an easy feat in c++. Ultimately, I settled on using usleep, a utility function I found online that made things work relatively well.

How do I clear the screen between displaying each step of the ant to show the path in as seamless way as possible? Clearing the screen was also an interesting challenge, as there is no easy way to do it natively in c++. I ended up using a combination of ANSI escape codes that basically 'hacked' a solution together by introducing whitespace and moving the cursor to the top left. Not ideal but it got the job done.

Letting the user enter height and width for the board versus just one number to create a square board: I had to decide whether I would ask the user for two values when creating the matrix (i.e. 100 x 200) or just one to always create a square matrix. I decided that the benefits of always having a square matrix (like just having one "size" variable rather than height and width variables that have to be passed everywhere) outweighed any benefit of having the user have such granular control over board size.

What happens when the Ant wants to move to a location that is out of bounds? This was probably the hardest design decision in this program because I had to go against the spec in one way or another. Because the user is specifying a board size and a number of moves, it is quite possible that the ant will run out of room before the number of moves is reached. What happens then? There were many options possible, some of them discussed in the q&a thread such as expanding the board, ending the program, throwing an error, etc. In the end, I decided to treat the number entered by the user as a "max" number of moves, and not as the number of moves the ant **had** to make. That way, if the ant reached a boundary then I would just show a message saying the ant had fallen off the board and end the program. In that message, I suggest to the user to use a larger board next time. I also added a line at the top of the screen that shows the number of moves the ant has made at any point during execution and I show the user how many moves the ant made before falling off.

Testing methodology

I ran various combinations of inputs to test the program:

- Small and large matrices for the board (i.e. 1, 10, 100, 1000)
- Small and large number of moves (i.e. 1, 10, 100, 10000)
- Invalid numbers for matrices and moves (i.e. negative numbers, letters)
- Starting the ant in a random location
- Starting the ant near an edge
- Starting the ant in the middle of the board

Problems uncovered during testing

Input validation was not working when user entered letters instead of numbers: in the getInt() function, I was properly validating that the numbers entered were in a specified range, but when the user entered a character the program would go into an infinite loop showing the error message I had greated. To fix this, I added cin.clear() after the error message to clear the cin error flag, and cin.ignore(10000, '\n') to ignore any characters that were messing things up from the previous invalid input.

The rand() and srand() functions worked on my mac but not on flip: this was easily fixed, but it seems like the compiler/linker on my mac was automatically pulling in the cstdlib library, whereas in flip I had to specifically add an #include <cstdlib> at the top. Took me a while to figure out what was going on.

Usleep worked on my mac but not on flip: same issue as above, but much harder to fix here. The usleep function was working fine on my mac to add a delay to the program, but when I tested on flip it did not work. I tried a bunch of similar functions and ways to make it work with not luck. I ended up deciding to remove the delay anyway because it made the display way to slow for larger (5000+) numbers, which is where things get interesting anyway.