Objective:
Produce a program that simulates the convention of Langton's ant. The rules are as follows:

- If the ant is on a **white space**, turn **right** 90 degrees and change the **space to black**.
- If the ant is on a **black space**, turn **left** 90 degrees and change the **space to white.**

Initial Design Strategy
I want the user to be able to choose the Ant's location or let the program decide for the user (randomly). The user will firstly be prompted to determine the initial size of the board. I am considering allowing the user the option to generate the board size at random (and implementing limitations on the size) but for now I'm keeping it simple.

A general sample menu is as follows:

```
1. Prompt user initially and ask them if they would like to play the program or quit.
    1. Play
    2. Quit
2. If the user selects play, the program asks the user to first determine the size of the game board
    1. How big would you like the board to be?
        1. Request input for X
        2. Request input for Y
3. After determining the size of the board, the user will be asked to determine whether or not they would like to manually input the ant's
position or allow the program to determine it for them.
    1. Input the board constraints
        1. Request input for board size X
        2. Request input for board size Y
    2. Allow the program to determine it for me
        1. Program runs a function which randomly generates the ant's position on the board based on the size of the initial board. It's important
that the board size is generated first because the ant's random position is determined by the ultimate size of the board.
```

Once the initial details are set by the user, the game starts and runs all of the necessary functions needed in order to play the game.

The entire program can be broken up into several sections in order to make it a lot easier. Namely, there will be a total of 3 function files, 3 specification files, and 1 main file (the one that starts the program):

```
1. board.hpp
    1. board.cpp
2. ant.hpp
    1. ant.cpp
3. validation
    1. validation.cpp
4. main.cpp
```

Test Strategy

Test cases for input validation

Test Case	Input Values	Driver Functions	Expected Outcome	Observed Outcome
Input below 0	Input = 1	validX( ) from main.cpp	INVALID input - prompt user again.	Loops back and asks the user to try again with an input.
Input at 0	Input = 0	validX( ) from main.cpp	INVALID input - prompt user again.	Loops back and asks the user to try again with an input.
Input at 0	Input = 0	validW && validH from main.cpp	VALID input - allows the user to proceed	Input is valid and allows the user to proceed
boardX = 100 User attempts to enter 101 for Ant starting X position	Input = 101	validW( ) from main.cpp	INVALID INPUT - prompt user to try again.	Segmentation fault - core dump.
Input is a negative value	Input = -1	Any validator function	INVALID INPUT - prompt user to try again.	User is prompted that the input is incorrect and that they should try again.
Input is NOT an integer (is a character or letter)	Input = 'a'	Any validator function	INVALID INPUT - prompt user to try again.	User is prompted that the input is incorrect and that they should try again.

Test cases for game board generation and ant movement

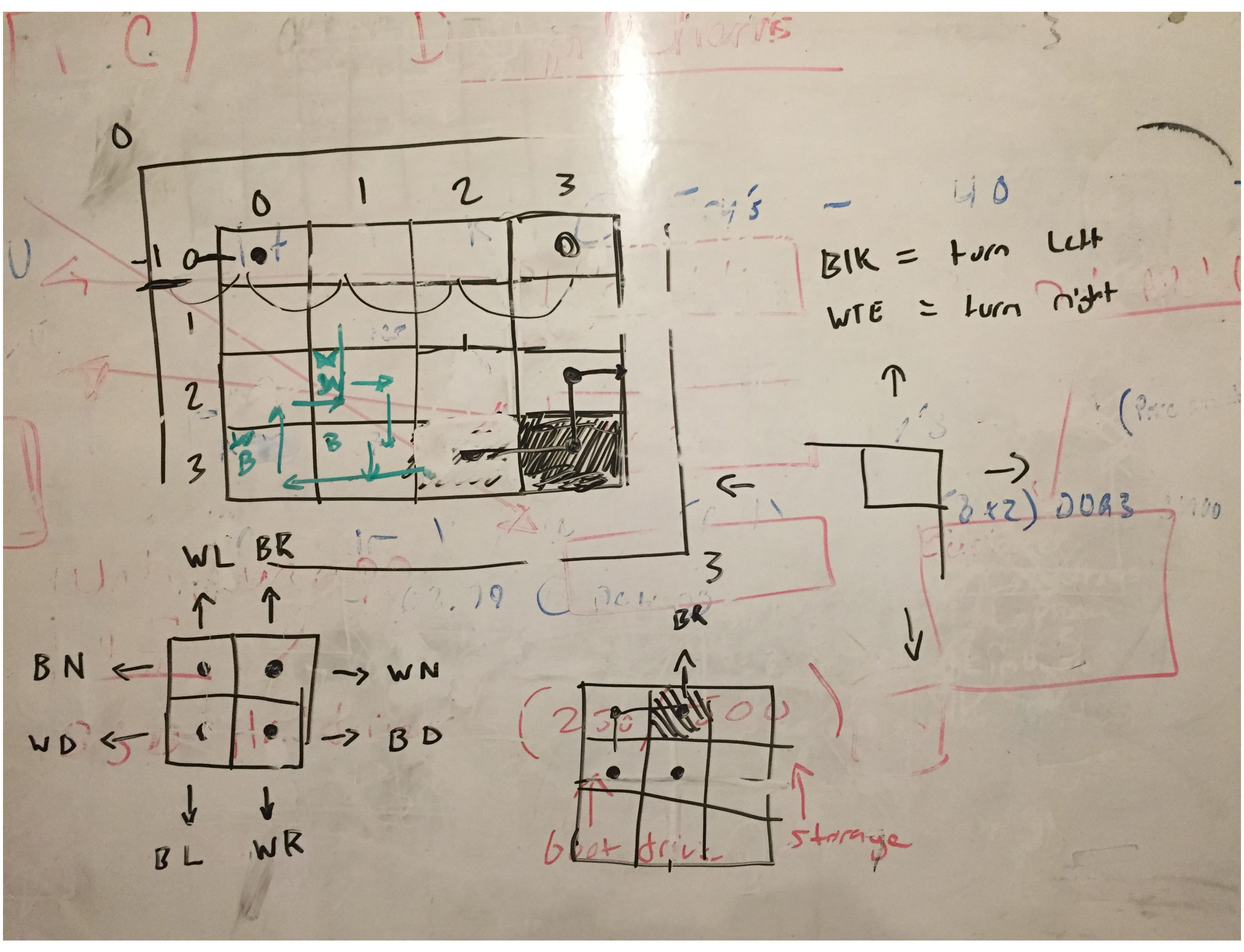
Test Case	Input Values	Driver Functions	Expected Outcome	Observed Outcome
Board class generation	N/A	Board class creation from main( );	Board class is generated and passes in necessary information needed for board dimensions and ant position	Board class is produced and additionally creates an ant class with all of the pertinent information.
Ant hits 2d array boundary	N/A	Main Driver function is the ant.move( ) function from the ant.cpp file.	Ant is carried over to the opposite end of the array relative to the ant's facing	BAD_ACCESS errors (related to trying to send the ant out of bounds)
Cell color behavior from ant interaction.	N/A	Main driver function is the recolor( ) function from the board.cpp file.	Board changes cell color when the ant leaves it	As the Ant leaves the cell, it moves in the direction it would based on it's facing and whether its on a black or white tile, and the tile it USED to be on also changes its color.

Project Reflection

Damn, this project was incredibly difficult due to the way I had to think about organizing all of the possible functions that I would need in order to make the final product. Establishing all of the menu functions were *somewhat* easy. I originally had some difficulty in getting the do-while loop to function (this would effectively loop the game again if the user wanted to at the end of the game). I managed to get it after establishing all of the minor input/output menus and wrapping it into the do-while loop. It was definitely a case of overthinking things.

For the most part, I was pretty close, execution wise, in the files that I ended up building. I used a validation file to offload some repeatable code (and not repeat myself when running validation in the main.cpp), and I also established a board and ant class. ORIGINALLY I was generating both the ant and the board from the main.cpp, however I figured that I didn't need to generate the ant, and could do so from within the Board class. This made it a lot easier because each time a new board is created, an ant class is *also* created. When I deallocate the memory from the board, it also purges everything inside of the Board (which means the ant gets deleted as well). Saves me from having to delete two classes.

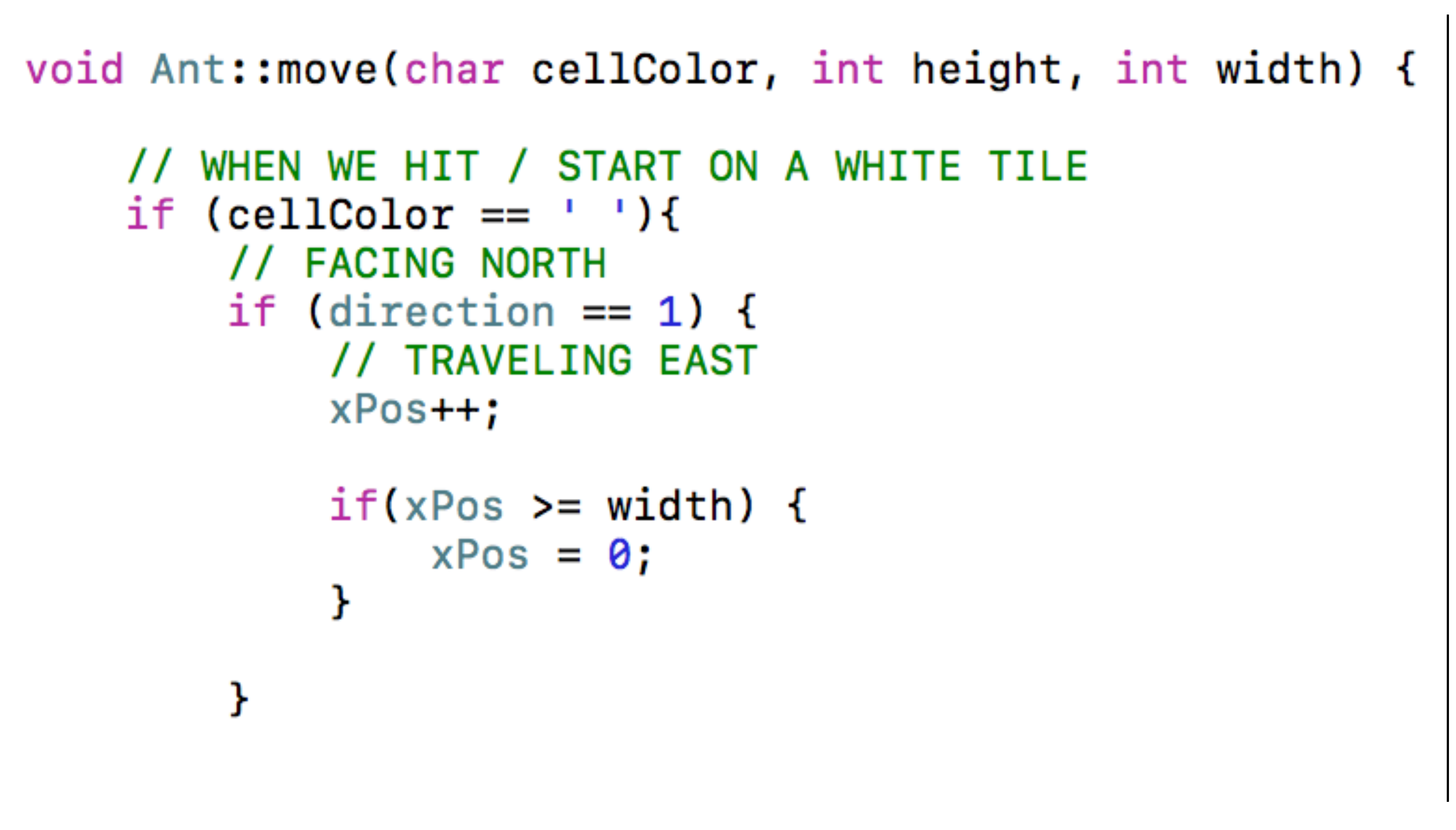
The absolute biggest issue I ran into was the ant positioning and direction logic. Below is a picture illustrating the 2d Array visualization I was using to sort of go through various steps and determine WHEN the ant was hitting or going over a boundary:



I concluded that usually, the only time you would go out of bounds is if you hit a specific tile from a specific direction. For example, if you were facing NORTH and you were on a BLACK tile, you would effectively go out of bounds if the ant's Y coordinate was 0. I broke down a bunch of these conditions in the bottom left of the white board, and attempted to apply that logic to the .move( ); function.

This was by far the most time-consuming part of the project because I essentially had to determine **how** I was going to shift the block to the OPPOSITE side of the array when it went out of bounds. I toyed with several cases within the .run( ); function but the problem here was that the move was occurring BEFORE the if-condition could get triggered, so it would run out of bounds, redraw the map, and then tell me their was a BAD_ACCESS error. Very annoying.

My solution ultimately turned into this:



```
void Ant::move(char cellColor, int height, int width) {

    // WHEN WE HIT / START ON A WHITE TILE
    if (cellColor == ' '){
        // FACING NORTH
        if (direction == 1) {
            // TRAVELING EAST
            xPos++;

            if(xPos >= width) {
                xPos = 0;
            }

        }
```

Essentially, in my move( ) function I had two new integer values (height and width) which grabbed the maximum size of the board. I then used these values inside of the move to calculate what happens based on the direction and the current tile of the ant. In the case above, IF the ant is facing NORTH, we tell the ant to head east because he is on a white tile (hence xPos++). However, if the ant's move goes beyond or equal to the maximum width of the array, it'll reposition the xPos to 0 - effectively starting it at the opposite end of the board and going from there.

There's a lot more that can be said about this project and how challenging it was, but this is the cut off point for me to be honest. I'm running on 3.5 hours of sleep because the logic of the game was frying my brain, and my Dog was throwing up all over the place the same night I decided to resolve this issue…so I'll leave it at that. Overall, the project was very challenging, and the relief that came from finally figuring out how to control the logic of the ant was great. Really tests your mettle as a programmer.

- Chris