Andrew Clos
CS-162 - 400
Project 1 Design, Test Table, and Reflection of Langton's Ant

Design:

Classes:

Menu:

The Menu class will greet the user, and present options in text format. Menu options will be numerical and will expect the user to enter an integer to make their choice. The menu class will also collected data from the user to establish the parameters of the simulation. This data will be stored in an array that the menu takes as a pointer from the caller.

Key functions:

Main Menu: Welcome message with options to begin the simulation or quit. Returns this choice as a bool to the caller.

Parameter Menu: If the user elects to begin the simulation in the Main Menu, the program will collect the necessary parameters from the user for: board dimensions, number of steps, and ant start position. The user will also have the option to choose a random Ant starting location. After the parameters are collected, a legend will be printed, along with the chosen parameters for the current simulation (separate helper function). This function will not return anything, but it will need to be given a pointer to an array in order to populate the values.

Integer Validation: Integer validation will be performed on all user menu options as well as all user entry of simulation parameters. This function will prompt the user to enter data and store that entry as a string. It will analyze each character within that array and make sure it is valid. It will be passed the upper and lower limits by the caller as an optional extra feature. The function will use a while loop to continue to prompt for data entry until a valid entry is received. It will have descriptive cout statements to explain to the user why the entry failed. This function will return the validated integer to the caller.

Play Again: This public function will be called after a simulation has completed to ask the user if they'd like to play again or quit.

Ant:

The Ant class will contain private data members for all of the parameters of the simulation: board dimensions, Ant starting position, number of steps and Ant direction. Several functions of the Ant class will work with a dynamically allocated 2d array to represent the game board.

Key functions:

Take Step: Taking a step will involve three main actions. (1) Move the Ant forward, (2) turn the Ant based on the color of the square it has landed on, (3) toggle the color of the square the Ant landed on to the opposite of what it was before. The turning of the ant and toggling of the color will be performed with private helper functions called from this

one. This function will also have to have some logic to prevent the Ant from moving out of bounds. It is my intention to have the ant "clip" across to the other side of the matrix if it moves out of bounds.

Make Board: This function will simply create a board based on the user's desired dimensions and populate it with space symbols to represent white spaces.

Print Board: Prints the current status of the board to the console including space color and Ant location. Will color the Ant based on the direction it is facing and will print the legend of the color to direction relationship.

Main method:

The array for game parameters and the dynamically allocated 2d array will be created here and passed to functions as pointers. The cycling of steps and calls to the print functions will also occur here. Finally, calls to menu functions like "main menu" and "play again" will take place. The memory for the dynamic array will need to be freed up.

Test Plan:

Test Category	Description of Test	Expected Results / Handling
Input validation: is a digit	1. Enter symbols: '%', '&', ''' '"', space, etc. both by themselves and sandwiched between valid input. 2. Enter characters alone or before, after or sandwiched between valid input. Ie. '2a' ,'a2', '2a2', 'a2a' 3. Negative numbers! '-0', '-1' etc.	-This should be caught early in by not allowing ASCII ranges other than 48-57 at any position in the string. -Must make sure entry isn't being truncated. If user enters '2a,' we cannot assume they meant '2'!! -Will need to allow for '-' character in first digit to signify negative number
Input validation: is an integer	1. Entering leading zeros, any other non 1-9 char in the first position or any non 0-9 in subsequent positions. 2. '2.1' float / double handling. 3. Adhere's to size boundaries of integers. If limit is set too high or not at all.	-Explain special case of leading 0 not being allowed with cout message. -Must make sure entry isn't being truncated. If user enters '2.1,' we cannot assume they meant '2'!! -If not establishing an upper limit, warn user that their entry is above size of signed int (32,767), etc.

Andrew Clos
CS-162 - 400
Project 1 Design, Test Table, and Reflection of Langton's Ant

Input validation: adheres to limits	1. Make sure the optional limit checking feature of input validation is working by entering values higher or lower than limits. 2. Enter 0's, negative numbers. 3. Check numbers that are equal to the bounds of the limits. 4. Check numbers 1 greater or less than the limits.	-if limits are exceeded, program may through a segmentation fault -could indicate a logic or even syntax issue (variable swapping).
Board Sizes / Ant location	1. Does the board print the correct number of rows and columns, are they reversed (enter: 2 x 2, 3 x 2, 2 x 3), etc. 2. Does the Ant start in the right place? Test: antRow = 1, ant Col = 0 3. Negative or 0 board dimensions not allowed	-Print small and default boards with a special symbol for testing only to visually determine that board dimensions are correct. - Should be: # # # * # # Not: # * # # # # -Out of range issues should be handled by input validation
Steps	1. No crashing with high numbers of steps (32,767) 2. Proper number of steps are run always. Test 1, 2, 3, 50, etc.	-test in Visual Studio and FLIP server, very high values of steps. -steps not off by 1 due to looping / counter logic issues.
Langton's Ant Function	1. Does the 'highway' get created. Does this behave similarly to other implementations. 2. Does the ant move, turn and toggle color properly. 3. Edge clipping / boundary issues. Test ant facing each boundary of the matrix.	-check Wikipedia and linked resources to compare for typical behavior of this simulation. -Run many simulations with just a 1-5 steps and track each movement of Ant to make sure it behaved per specification. -When Ant is facing a boundary and takes a step, does the ant appear on the other side, no seg faults.

Andrew Clos
CS-162 - 400
Project 1 Design, Test Table, and Reflection of Langton's Ant

Random Function	1. Run many simulations with consistent board sizes. 2. Ant never placed out of bounds. 3. Ant gets randomly placed in to any row or column	-Expect to see Ant placed all over the board. Not just in one corner or the same place. -Run many simulations to make sure ant cannot go out. -Lots of small board simulations to make sure Ant is sometimes placed in max row/col and min row/col.
Print Function	1. Do the correct symbols, boundaries, legends, colors print. 2. Are the boundaries and symbols aligned. Test 1x10, 10x1, square and other sizes.	-Symbols should always be aligned no matter the board size or dimensions. -Small number of step simulation to make sure Ant color changes properly (matches legend).
Memory Leaks	1. Valgrind with -v options. 2. Test different array sizes. 3. Test case of quitting right away	-no memory leaks of any kind

Reflection:

Input Validation:

Input validation took a lot of time to implement and I was stuck between choosing whether to send the input validation the variable that the user had entered, or have the input validation function prompt the user for the variable itself. In the end, I choose to have the function prompt the user for the input which seemed to make for a cleaner, but more self contained function.

Also for input validation, I had initially had several types of validation (one for ints, one for menu choices and another for ranges of integers). I decided to include the option to pass upper and lower limits for the ranges and have the input validation address those limits if 0's had not been entered. This helped keep all input validation in one place.

Although I always set upper and lower limits (and flagged useLimits 'y') for this program's implementation of input validation, I wanted to include the ability to ignore the limit checking component of this function for potential future utility in other programs. Not all future programs may have limits below the max size of an integer, so I will need to add additional checks in the input validation to make sure the number entered does not get truncated or other unexpected behavior occurs.

Regarding the variable badCount. I chose to use an integer (initialized to 0) to keep track of the number of invalid digits found during the first step of integer validation. I could have used a simpler bool variable type, but I thought that for future projects, having a count of the total number of invalid

Andrew Clos
CS-162 - 400
Project 1 Design, Test Table, and Reflection of Langton's Ant
digits could be useful for debugging or as feedback to the user. I included a cout near the end of the intValidate while loop to output the number of invalid digits to the user.

I learned about using cin.clear and cin.ignore, but initially had them placed inside the for-loop that steps through the strings. This caused problems, where if a string length of more than 2 was entered, the loop was ignoring this input. Moving the clear and ignore commands to the end of the function proved to be helpful and gave the desired behavior.

The Menu class was almost a function instead. This ended up being a simple class, but because I wanted to house the input validation within it and break out other bits of code as their own functions to avoid duplicating text, I turned this in to a class. In the future I may decide to make input validation its own class to handling the validation of integers, doubles, strings, etc.

It seems that this (and many other) programs "live" inside of while loops. From this assignment, I experimented with bool and int triggers to keep the while loops running or exit them. They are powerful, but I will need to keep scope in mind when operating this way. Thankfully, it does seem that mistakes with while loops are generally fairly obvious, because the program either unexpectedly ends, or never lets you out of the loop.

I also learned a lot about debugging features in both Visual Studio and with the makefile / valgrind functions on flip. This program got pretty large at one point and I had lots of changes along the way. This led to me stranding a few variables, which thankfully were found by the debug / valgrind options run with makefile.