# Project 1 (Design + Reflection)

Project 1 (Langton’s Ant) design and reflection

## Design / Flowchart

Project 1 flowchart (PDF) is included within this assignment submission’s .zip file



## Test Cases

Program test tables, including: test plan, expected output, and actual output.

“Quote”

Use styles to easily format your Word documents in no time:

* For example, this text uses the List Bullet style.
* On the Home tab of the ribbon, check out Styles to apply the formatting you want with just a tap.

## Reflection

This assignment reflection will cover the following content: (1) Design changes while developing and implementing the assignment’s program, (2) problems that were experienced and how those problems were overcome, and (3) what was learned from this assignment.

From my initial flowchart (design) draft to my final draft, the primary design changes related to input validation and details, including the “modularization” of an input validation function, for example.

As mentioned above, one major problem that I experienced was modularizing an input validation function, which was specifically for validating that the correct type of character was input by the program’s user. *Type* refers to whether the input is a string, integer, float, etc. When the program’s user must enter in values for board attributes, like the number of rows and columns, and the starting coordinates of the ant, accepting the starting row and column as user inputs, the program is designed to only accept integers, as each row and column corresponds to a pointer’s array element, and thus, there is no float/decimal row or column value. And if a number is required, then a letter or special character should not be acceptable as input either. This problem was solved by taking advantage of *cin’s* error flags for user-input values.

The input validation’s function accepted a parameter by reference, corresponding to an integer variable, which would represent the number of columns or rows on the board, the number of steps taken by the ant, and the ant’s starting position by row and column. The parameter was passed by reference so that the corresponding variable’s value is updated without having to return a value with the function. The function begins by using a *while loop*, which evaluates the user’s input to determine if there was an input error. If there was an error, then the input validation function’s contents will execute, which begins by clearing the input stream, then asks for user input to overwrite the previous value, which caused an error.

When implementing this function, I encountered an issue where the user was unable to enter in inputs a second time, using the input validation function, because some contents would continuously print over and over again without giving the user a chance to enter inputs. This was resolved by implementing a *cin.ignore()* line of code within the input validation function, which was located after *cin.clear()*. By ignoring any remaining characters within the input buffer (with *cin.ignore*), the problem was resolved, allowing the user to reattempt entering inputs. This problem and solution taught me that characters can still be left in the input stream, preventing the user from re-entering inputs for a variable, even though the remaining characters within the input stream’s buffer are unseen. Thus, this “buffer” of remaining characters must be ignored in order to operate as expected.

A second problem that I encountered was when trying to randomly generate the ant’s starting position, which is represented by column and row values. *Rand()* was used to generate a random number, and then the modulus was taken of the random number by the number of rows and/or columns, depending on whether the ant’s starting row or column was being calculated. By taking the modulus of the random number, the resulting value would be a remainder from zero to column/row minus one. Since the zero element of both the board’s rows and columns represent the board’s border, one was then added to the remainder value. By adding one, the possible value would range from one to the total number of columns/rows, which includes the entire “playable” board, as the number of columns/rows plus one represents the other border “wall”.

The problem was that the ant’s starting position would be the same every time the program ran. The reason for this is because *rand()* uses a “seed” to generate the applicable values. If the seed is not changed each time the program runs, then the “random” numbers will be the same each time the program runs until the “seed” value changes. The solution for this problem was to use *srand()* while using “time(0)” as its parameter, appearing as: *srand(time(0))*. *Srand()* generates the “seed” value, which is used by *rand()* to generate “random” numbers. By using the current time, represented as “time(0)”, as *srand()*’s function parameter, the function will never have the same parameter value, as a certain time only happens once. Since the function parameter of *srand()* is always different, the “seed” value will always be different, and thus, *rand()*’s values will always be different whenever the line of code executes, generating the ant’s starting row and column position. I learned about how C++’s *rand()* function generates random values, which is based on the “seed” value, and I also discovered a means of always generating a unique value, which can be done by taking the current time with, *time(0)*. An entirely unique value based on the current time can be extremely valuable with future functions and programs.

Overall, the two major problems that I faced were with respect to creating a modularized input validation function and generating random values within a certain range each time the program runs. These two challenges were overcome by learning more about *cin* and its input stream of characters and error flags, and understanding how C++ generates random integer values with *rand()*, which uses a “seed” value to generate a series of random values. Encountering challenges while developing solutions to overcome those problems teaches developers, like myself, a detailed understanding of how C++ and its expressions work, enabling us to apply those learnings to future functions and programs.