Program name: Project1 Langton's Ant
Author: Xiaoying Li
Date: 7/7/2019
Description: Design & Test Plan & Reflection of Project1 Langton's Ant

- **Design**

1. Main

The main function controls the entire project.
Ask user whether he/she wants to start the game or not.
If the user decided not to start the game, quit the project.
Otherwise, start the game,
Prompt the user for following information:
the number of rows of the board;
the number of columns of the board;
the starting row of the ant;
the starting column of the ant;
the starting direction of the ant;
the number of steps the ant will move during the simulation.
Introduce the symbols' meanings and the solution of edge cases to the user before start the simulation.
Start the simulation using information inputted by the user.
After the simulation ends, ask the user whether he/she wants to play again.
If the user decided to play again, start the game again.
Otherwise, quit the project.
Loop the game until the user decided to quit.

2. Menu

The menu function prints menu options to the screen for the user, and after the user makes a choice, verify the user's input, and return the value back to the program.

(1) startMenu:
When the project starts, ask user whether he/she wants to start the game or not.
If the user decided to start the game, return true.
Otherwise, return false.

(2) endMenu:
When the game ends, ask the user whether he/she wants to play the game again.
If the user decided to play the game again, return true.
Otherwise, return false.

3. Ant

The Ant class contains the information of: the board; the ant's location during every step; the ant's direction during every step; and the ant's every move.

List of data members:

(1) Pointer to a 2D array holds the board;
(2) Integer holds the number of rows of the board;
(3) Integer holds the number of columns of the board;
(4) Integer holds the ant’s location of row;
(5) Integer holds the ant’s location of column;
(6) Integer holds the ant’s direction;
(7) Character holds the color of a space.

List of functions:

(1) A constructor initializes data members;
(2) A method creates the 2D board;
Dynamically allocates memory space for the 2Dboard.
Using user specified row number and column number to create the board, which is filled with white space and had 4 edges around it.
(3) A method determines every step of the ant;
If the ant is on a white space, change the current space to black.
If the ant's direction is up, move one step right,
and change the ant's direction to right.
If the ant hits the right edge, moves to the first space of this row.
If the ant's direction is right, move one step down,
and change the ant's direction to down.
If the ant hits the bottom edge, moves to the first space of this column.
If the ant's direction is down, move one step left,
and change the ant's direction to left.
If the ant hits the left edge, moves to the last space of this row.
If the ant's direction is left, move one step up,
and change the ant's direction to up.
If the ant hits the up edge, moves to the last space of this column.
If the ant is on a black space, change the current space to white.
If the ant's direction is up, move one step left,
and change the ant's direction to left.
If the ant hits the left edge, moves to the last space of this row.
If the ant's direction is right, move one step up,
and change the ant's direction to up.
If the ant hits the up edge, moves to the last space of this column.
If the ant's direction is down, move one step right,
and change the ant's direction to right.
If the ant hits the right edge, moves to the first space of this row.
If the ant's direction is left, move one step down,
and change the ant's direction to down.
If the ant hits the bottom edge, moves to the first space of this column.

Print every step and use a variable to store the color of the space the ant locates now to determine next step of the ant

(4) A method prints the board;

(5) A method runs every step of the ant until it reaches the designated step.

After all steps run, free the dynamically allocated memory for the board.

4. Validation

The validation function takes in a string and pass it by reference. The function makes sure the user's input is an integer. Afterwards, it returns the integer the user inputted.

Loop until the user inputs an integer.

When a string inputted, detect whether an extraction has failed and fix it.

If an extraction has failed, ask the user to enter again.

If extraction succeeds, then make sure no extraneous input.

Then detect if every character of the string is digit.

If every character of the string is digit, then the string is an integer.

Otherwise, ask the user to enter again.

If the input string is an integer, transfer it to an integer, and return the integer.

The validation of the range of every integer returned should be validated after the integer returns to the program.

- **Test Plan**

1. Test Plan for Input Validation

Test Object	Input	Expected Result
menu() function		
startOption/ endOption	0	quit the program
	1	start the program
	23	output: Your enter is invalid. Please enter again. Loop until the user enter 0/1.
	-45	output: Your enter is invalid. Please enter again. Loop until the user enter 0/1.
	1.67	output: Your enter is invalid. Please enter again. Loop until the user enter 0/1.
	a8902	output: Your enter is invalid. Please enter again. Loop until the user enter 0/1.
	345b~	output: Your enter is invalid. Please enter again. Loop until the user enter 0/1.
main() function		
rowString/ column	0	output: Your enter is invalid. Please enter again. Loop until the user enter an positive integer.
	1	Return 1 to int row/ int column
	200	Return 200 to int row/ int column
	-3	output: Your enter is invalid. Please enter again.

		Loop until the user enter an positive integer.
	56.7	output: Your enter is invalid. Please enter again. Loop until the user enter an positive integer.
	c890!	output: Your enter is invalid. Please enter again. Loop until the user enter an positive integer.
	12d@	output: Your enter is invalid. Please enter again. Loop until the user enter an positive integer.
startRowString/ startColumnString (for validation purpose, assume row=50, column=75)	0	output: Your enter is invalid. Please enter again. Loop until the user enter an positive integer no more than row/ column.
	1	Return 1 to int startRow/ int startColumn.
	50	Return 50 to int startRow/ int startColumn.
	51	For startRowString: output: Your enter is invalid. Please enter again. Loop until the user enter an positive integer no more than row. For startColumnString: Return 51 to int startColumn.
	75	For startRowString: output: Your enter is invalid. Please enter again. Loop until the user enter an positive integer no more than row. For startColumnString: Return 75 to int startColumn.
	76	output: Your enter is invalid. Please enter again. Loop until the user enter an positive integer no more than row/ column.
	100	output: Your enter is invalid. Please enter again. Loop until the user enter an positive integer no more than row/ column.
	-2	output: Your enter is invalid. Please enter again. Loop until the user enter an positive integer no more than row/ column.
	3.45	output: Your enter is invalid. Please enter again. Loop until the user enter an positive integer no more than row/ column.
	#e678	output: Your enter is invalid. Please enter again. Loop until the user enter an positive integer no more than row/ column.
	9f$	output: Your enter is invalid. Please enter again. Loop until the user enter an positive integer no more than row/ column.
startDirectionString	1	Return 1 to int startDirection
	2	Return 2 to int startDirection

	3	Return 3 to int startDirection
	4	Return 4 to int startDirection
	0	output: Your enter is invalid. Please enter again. Loop until the user enter 1/2/3/4.
	-5	output: Your enter is invalid. Please enter again. Loop until the user enter 1/2/3/4.
	6.78	output: Your enter is invalid. Please enter again. Loop until the user enter 1/2/3/4.
	%g90	output: Your enter is invalid. Please enter again. Loop until the user enter 1/2/3/4.
	1h^	output: Your enter is invalid. Please enter again. Loop until the user enter 1/2/3/4.
stepString	0	Return 0 to int step.
	1	Return 1 to int step.
	20000	Return 20000 to int step.
	-3	output: Your enter is invalid. Please enter again. Loop until the user enter a nonnegative integer.
	4.56	output: Your enter is invalid. Please enter again. Loop until the user enter a nonnegative integer.
	&i78	output: Your enter is invalid. Please enter again. Loop until the user enter a nonnegative integer.
	90*j	output: Your enter is invalid. Please enter again. Loop until the user enter a nonnegative integer.

2. Test Plan for Program Output

	Input		Expected Result
Case 1	row	1	Output 11 boards from step 0 to step 10, every board is a 1*1 board with 4 edges around it and a * in the only one space.
	Column	1	
	startRow	1	
	startColumn	1	
	startDirection	1	
	step	20	
To test if the ant's every step's direction and color change is right as well as the edge case.			
Case 2	row	10	Output 6 boards from step 0 to step 6, every board is a 10*30 board with 4 edges around it. The * should appear at 4th row and 5th column on the 0th board. Then the ant first moves in the clockwise direction. And at step 5, the ant moves in the anti-clockwise direction.
	Column	30	
	startRow	4	
	startColumn	5	
	startDirection	2	
	step	5	
Case 3	row	100	Output 11 boards from step 0 to step 11, every board is a 100*200 board with 4 edges around
	Column	200	

	startRow	45	it. The * should appear at 45th row and 67th column on the 0th board. Then the ant first moves in the clockwise direction. And at step 5, the ant moves in the anti-clockwise direction.
	startColumn	67	
	startDirection	3	
	step	10	
Case 4	row	20	Output 16 boards from step 0 to step 16, every board is a 20*30 board with 4 edges around it. The * should appear at 5th row and 6th column on the 0th board. The ant should appear on the other side when hit edges.
	Column	30	
	startRow	20	
	startColumn	30	
	startDirection	4	
	step	15	
To test after about 10000 steps if the ant traces a pseudo-random path and finally starts building a recurrent "highway" pattern of 104 steps that repeats indefinitely.			
Case 5	row	150	Output 15001 boards from step 0 to step 15000, every board is a 150*160 board with 4 edges around it. The ant should traces a pseudo-random path and finally starts building a recurrent "highway" pattern toward southwest.
	Column	160	
	startRow	30	
	startColumn	110	
	startDirection	1	
	step	15000	
Case 6	row	150	Output 15001 boards from step 0 to step 15000, every board is a 150*160 board with 4 edges around it. The ant should traces a pseudo-random path and finally starts building a recurrent "highway" pattern toward northwest.
	Column	160	
	startRow	40	
	startColumn	120	
	startDirection	2	
	step	15000	
Case 7	row	150	Output 15001 boards from step 0 to step 15000, every board is a 150*160 board with 4 edges around it. The ant should traces a pseudo-random path and finally starts building a recurrent "highway" pattern toward northeast.
	Column	160	
	startRow	20	
	startColumn	100	
	startDirection	3	
	step	15000	
Case 8	row	150	Output 15001 boards from step 0 to step 15000, every board is a 150*160 board with 4 edges around it. The ant should traces a pseudo-random path and finally starts building a recurrent "highway" pattern toward southeast.
	Column	160	
	startRow	40	
	startColumn	130	
	startDirection	4	
	step	15000	

- **Reflection**

After reading through the project description and simulating the ant's move on the paper, I developed my algorithms and converted them to pseudocode. But the

pseudocode I wrote above in the “Design” section was very different from the first one I had wrote. I spent almost one week to finish this project, obviously I encountered many problems and made many changes to the original design.

1. Design changes I have made:

The first change I made was the menu() function. The one I designed at the beginning only contained one function asking user whether he/she wants to start the game or not. If the user decided to start the game, return true. Otherwise, return false. But in the actual test, I found this design had a flaw: when users end the first-time game, program should inquire if they need to play again instead of asking if they need to start game. Thus, I separated menu() function into startMenu() function which was called in the beginning and endMenu() function which was called in the end. The only difference between these two functions were the output inquiring users if they need to start game. This subtle modification significantly improved user experience.

The second change is about input variables’ data type and input validation. I’d like to discuss these two changes together because they are closely related to each other. At first, all the variables input by user shared the same data type: int. It was because the program needs integer to create the board, as well as setting up the ant’s starting location, direction, and number of steps the ant would move during simulation. After I learned knowledge about input validation in Utility Functions from Resources shared by instructor, I designed an input validation to ensure input date type was integer which prevented most invalid input. However, this validation still had a defect that when input was an integer plus non-integer characters like decimals, letters, or symbols, validation() function would return to the integer in input. Non-integer part would be ignored and cleaned. Program would not give any error message. I did a lot of research to fix this defect but hardly made any progress, until I saw Andrew Clos’ reply “I ended up really liking taking the entry as a string and then analyzing each character within that string before converting it to ‘int’ or whatever variable type you need” in Question@40 on Piazza. I did more research about how to detect if every character of a string is digit and how to transfer a string to an integer. I ended up setting user’s input data type to string and running validation on string input. The program would convert integer in string data type to integer data type and return to int variable. Tests proved this method effectively prevented all input error and only non-negative integers could pass validation.

2. How I solved problems encountered:

When I was working on this project, the most difficult part was input validation. Like I discussed above, in the beginning I had no idea how to design input validation at all. Thanks to knowledge about input validation in Utility Functions from Resources provided by instructor, I got a general understanding of how to make an input validation function. When I struggled fixing the input error mentioned above, I viewed discussions on piazza carefully to look for discussions covered my problem, and I did get some tips. Additionally, I found many websites including learncpp.com, cplusplus.com, and geeksforgeeks.com, etc. very useful. They provided much more knowledge and

information than piazza.

3. What I learned for this project:

This project was the first one required us to design all the functions and classes in this program. In CS161, I rarely write pseudocode. But in this project, I realized the importance of pseudocode which made my actual coding work eaiser.

In addition, I learned the importance of a robust test plan. Without a good test plan, it was impossible to discover the problem in validation() function in my code, not to mention solving it.

Moreover, this project showed the necessity to use various learning approaches and tools. Resources on Canvas, piazza, and many coding websites really helped me a lot during this project.