Sylvan Canales

CS 162 Assignment 1 Reflections

Design

Problem Statement

Design, implement, and test a program that implements a simulation of Langton's Ant. On an array or matrix of cells, each turn the value of each cell may change based upon the following two rules:

1. In a white square, turn right 90˚ and change the square to black.
2. In a black square, turn left 90˚ and change the square to white.

Analysis

The flow of the program will be broken up into two blocks: the setup menu and then the simulation loop.

During the setup menu, options will be presented to the user to customize the simulation by changing the size of the grid, the starting location for the ant, and the number of steps.

After the user has chosen their options and opted to start the simulation, the setup menu loop ends and a simulation is initialized and then loops until completion.

The Ant Class

In order for the ant to follow the rules of the game, the program must keep track of the ant's position as he moves across the board, and the ant must be able to determine the color of the cell it is on so that it can execute its turn and move forward. An Ant class is a good candidate to encapsulate the information and functionality needed to accomplish these tasks. To determine a possible list of data members and methods, consider the following questions about the ant:

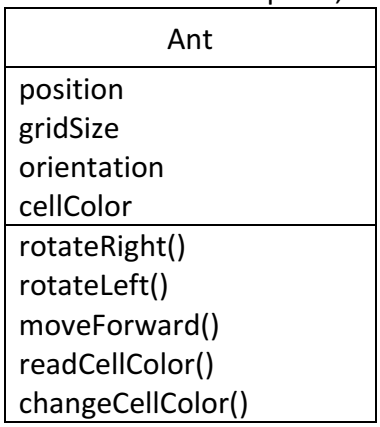
What does an ant need to know?

- Where it is located, or a grid position
- The size of the grid
- Which direction it is pointing
- What color is the cell that it resides in

What does an ant need to do?

- Turn right or left 90˚
- Move forward
- Read the color of the cell
- Change the color of the cell

A basic UML diagram of the class, as described to this point, looks like this:



Constructors

The Ant class has a default constructor that initializes its member variables, but it really isn't meant to be used because there are no methods provided to set the variables after instantiation. Instead, an overloaded constructor is provided to initialize the position, size of grid, and direction of the ant to the values requested by the user.

Variables

row

row stores the row component of the ant's position on the grid

col

col stores the column component of the ant's position on the grid

rows

rows stores the total number of rows on the grid, so the ant can know if it's on the boundary

cols

cols stores the total number of columns on the grid, so the ant can know if it's on the boundary

orientation

I created a new enum type called directionStates to hold the directions { north, east, south, west }. Orientation stores the value of the direction the ant is currently pointing.

currentSpace

This variable holds the color of the cell the ant is occupying, since the ant will overwrite the actual character stored in the grid.

Methods

rotateRight()

This function rotates the ant in a clockwise direction, i.e. if he is facing north, he turns to face east.

rotateLeft()

This function turns the opposite direction of rotateRight.

moveForward()

Checks if the ant is not going to walk off the edge of the grid, then advances one cell in the direction it's facing. Returns a bool to indicate if the move succeeded or not. If not successful, then the program can decide what will be done.

updateGridPosition()

Combined the functionality of reading the cell color and then turning it into a '*'.

changeCellColor()

Changes the character of the previous cell to the opposite color when moving to a new position. This function also needs a reference to the grid variable so that it can change the character.

After some refinement the class diagram looks like this:

Ant
- row : int - col : int - rows : int - cols : int - orientation: directionStates - currentSpace : char
+ Ant() + Ant(startRow : int, startCol : int, numRows : int, numCols : int, startOrientation : int) + rotateRight() : void + rotateLeft() : void + moveForward() : bool + updateGridPosition() : void + changeCellColor() : void

The Menu Class

The initial setup of the experiment requires some choices to be made by the user, and a simple menu system will be implemented to present a list and prompt for a selection. An attempt will be made to generalize the class so that it may be reused by other programs that also require some form of menu selection.

At its most basic level, a menu must present one or more options to the user and handle the user input to make the selection. Asking again the following questions about the class:

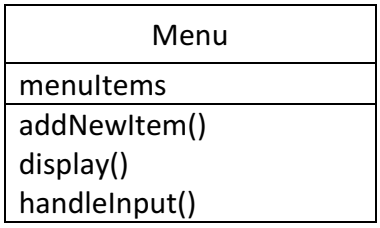
What does a menu need to know?

- Its menu items

What does a menu need to do?

- Add strings to the list
- Display itself to the user
- Handle input

A basic UML diagram of the Menu class, as described to this point, might look like this:



Constructors

I decided that, in the interest of making the menu class as general as possible, I would include multiple constructors for initializing a new menu instance, giving more flexibility to be implemented into different programs. The menu object may therefore be passed in a list of strings in either vector or array format, or created without a pre-built list to allow for adding new menu items at runtime if desired. I also decided that some kind of title would be nice, so it is now included as a data member.

Variables

menuItems

I settled on a vector of strings for the menuItems because it was easy to handle adding new menu items at runtime without knowing ahead of time how many items would be in the list.

Methods

presentToUser()

I opted to make an option to clear the console each time the menu is presented to the user. This is done by way of passing in true or false to the bool paremeter clear.

getSelectionFromUser()

This method contains a loop to validate the input, and prompt if necessary to re-enter a selection if the input was not valid (either not an integer, or out of range of the menu items). The return value is the valid menu item that was selected.

addNewItem()

This takes a single string parameter and pushes it to the back of the menuItems vector. Initially I had this function automatically adding a "Quit" item at the end of the vector, but I ended up abandoning that idea, and instead the Menu class will make no assumptions about what the menu items should be. That will be up to the calling program to determine.

After a more detailed analysis the class diagram looks like this:

Menu
- menuItems: vector <string> - title: string
+ Menu() + Menu(newTitle : string) + Menu(newTitle : string, newItems : vector <string>) +Menu(newTitle : string, newItems : string[], numItems : int) + setTitle(theString : string) : void + addNewItem(item : string) : void + presentToUser(clear : bool) : void + getSelectionFromUser() : int

Input Validation

In order to process user inputs for this program I created some helper functions to ensure that the inputs are of the proper type and that they fall within acceptable limits. For this program the only data type that needs to be input is the integer type, so I looked at ways to validate input for integer data types.

My goal was to create a generic black box validation function that could accept as inputs the upper and lower bounds, and the user input, and then return true or false if it was a valid integer and return the value. Unfortunately in C++, a single function can't return more than one value unless using custom data types. I thought about making an input validation class but in the end I decided to split the validation into three functions.

getValidatedIntegerInput()

This is the primary function that will get called by parts of the program that require proper integer inputs. The parameters are the lower and upper bounds for acceptable input. Inside this function, the user will be prompted for input, and a loop will capture the user until a valid integer is input. Once an integer within the acceptable range is input, it is returned to the calling function.

isStringAnInteger()

The first step in checking whether an input is valid is whether or not it is actually an integer. If using cin to read input directly into an int variable, undesirable behavior results if bad characters are read into the input stream. This includes non-integer characters and whitespace. To get around this problem, a string is input using getline() to capture all characters, including whitespace, up to the end of the line. Then this string is passed to isStringAnInteger() where it analyzes the string and determines if it contains only an integer. If it is an integer, then the string is passed to convertStringToInteger() to get the actual numeric value of the string representation.

convertStringToInteger()

This function takes a string parameter and computes the numeric value of the integer that is represented by the characters in the string. So a string containing the characters “256” will be converted to an int with value 256. It handles negative integers as well.

Program Flow

As previously mentioned, the program will be divided primarily into two loops, one for the menu inputs and one loop to run the simulation. Some initial default values will be provided so the user can start a simulation quickly if so desired.

If, during the simulation, the ant attempts to move forward into a cell that is outside the matrix, then I have elected to have the loop terminate and the program end with a message explaining what happened. I felt that was more in the spirit of the simulation rather than trying to move the ant to a different location, which would make incorrect patterns on the grid.

Note that I have chosen to set a maximum grid size to 121 based on what I felt would be a reasonable size to display on someone's terminal. So if the user tries to set a grid size > 121, the program will prompt for another input.

In pseudocode, the program flow will go like this:

```
Get a new random seed based on the current time
Initialize number of turns to 1000
Initialize grid size to 41x41
Initialize start position to 21,21
Initialize integer constant MAX to 121
Initialize starting orientation to north
Initialize grid color to ' ' (white)
Instantiate a new menu with a title "Langton's Ant – Main Menu"
Add menu items to the menu
Do
        Display menu to the console
        Input menu selection
        If selection is not one of the menu choices
                Re-input, loop until valid
        End if
        Switch selection
                Case grid size
                        Input grid size
                        If grid size < position
                                Change position to fit new grid size
                        End if
                        If grid size < 1 or > MAX
                                Re-input, loop until valid
                        End if
                Case starting position
                        Input starting position
                        If starting position > grid size or < 0
```

Re-input, loop until valid
End if
Case randomize position
Generate random start position and orientation
Case number of turns
Input number of turns
If turns < 1
Re-input, loop until valid
End if
Case run simulation
Do nothing (end the loop)
Case quit
return from main
End switch
While menu selection != run simulation
Dynamically allocate a new 2D array of characters and initialize to ‘ ‘ (white)
Initialize turn to 0
Instantiate an Ant object and pass it the starting position, grid size, and starting orientation
Do
Ant records color of the cell it’s occupying
Ant changes the character of the cell to ‘’*
Display the grid to the console
If color is white
Ant turns right
Ant tries to advance
If move was valid
Ant updates its position
Ant swaps the color of previous cell
turn += 1
Else if move was not valid
Do nothing, program will end gracefully
End if
else if color is black
Ant turns left
Ant tries to advance
If move was valid
Ant updates its position
Ant swaps the color of previous cell
turn += 1
Else if move was not valid
Do nothing, program will end gracefully
End if
End if
While move is valid and turn < number of turns

If last move was valid

print “Simulation ended successfully. Completed [turn] turns.”

Else

print “Simulation aborted while trying to go past the edge of the grid. Try a larger grid next time. Completed [turn] turns.”

End if

Testing

The main emphasis for this test plan is to ensure that the integer input validation works properly, and the program exits gracefully when the ant tries to exit the grid.

Validating Integer Input

Test Case	Input Values	Driver Functions	Expected Outcomes	Observed Outcomes
Input not an int	Input = df ag	main() do..while menu != run simulation	Prompt to re-enter an integer between 1 and last menu item	Prompt to re-enter an integer between 1 and last menu item
Input not an int	Input = 2h	main() do..while menu != run simulation	Prompt to re-enter an integer between 1 and last menu item	Prompt to re-enter an integer between 1 and last menu item
Input not an int	Input = 4 1	main() do..while menu != run simulation	Prompt to re-enter an integer between 1 and last menu item	Prompt to re-enter an integer between 1 and last menu item

Validating Menu Selection

Input too low	Input < 1	main() do..while menu != run simulation	Prompt to re-enter an integer between 1 and last menu item	Prompt to re-enter an integer between 1 and last menu item
Input too high	Input > menu items	main() do..while menu != run simulation	Prompt to re-enter an integer between 1 and last menu item	Prompt to re-enter an integer between 1 and last menu item
Input in correct range	0 < input <= menu items	main() do..while menu != run simulation	Selected case is performed	Selected case is performed
Random start position	Input = 7	main() do..while menu != run simulation	Start position has been randomly selected within 0 <= position < grid size	Random start position within the bounds of the grid

Validating Simulation Parameters

Test Case	Input Values	Driver Functions	Expected Outcomes	Observed Outcomes
Input too low	Input < 0	main() specifying start position	Prompt to re-enter an integer between 0 and grid size - 1	Prompt to re-enter an integer between 0 and grid size - 1
Input too high	Input > grid size -1	main() specifying start position	Prompt to re-enter an integer between 0 and grid size - 1	Prompt to re-enter an integer between 0 and grid size - 1
Input extreme low	Input = 0	main() specifying start position	Start position is set at row or column = 0	Start position set to 0
Input extreme high	Input = grid size-1	main() specifying start position	Start position is set at row or column = grid size -1	Start position set to grid size-1
Input in correct range	0 <= input < grid size	main() specifying start position	Start position is set to input	Start position is set to input
Input too low	Input < 1	main() specifying grid size	Prompt to re-enter an integer between 1 and 121	Prompt to re-enter an integer between 1 and 121
Input too high	Input > 121	main() specifying grid size	Prompt to re-enter an integer between 1 and 121	Prompt to re-enter an integer between 1 and 121
Input extreme low	Input = 1	main() specifying grid size	Grid size is set to 1	Grid size is set to 1
Input extreme high	Input = 121	main() specifying grid size	Grid size is set to 121	Grid size is set to 121
Input in correct range	0 < input <= 121	main() specifying grid size	Grid size is set to input	Grid size is set to input
Input too low	Input < 1	main() specifying number of turns	Prompt to re-enter an integer between 1 and 2147483647	Prompt to re-enter an integer between 1 and 2147483647
Input too high	Input > 2147483647	main() specifying number of turns	Prompt to re-enter an integer between 1 and 2147483647	Prompt to re-enter an integer between 1 and 2147483647
Input extreme low	Input = 1	main() specifying number of turns	Turns set to 1	Turns set to 1

Input extreme high	Input = 2147483647	main() specifying number of turns	Turns set to 2147483647	Turns set to 2147483647
Input in correct range	0 < input <= 2147483647	main() specifying number of turns	Turns set to input	Turns set to input

Validating Ant Movement

Test Case	Input Values	Driver Functions	Expected Outcomes	Observed Outcomes
Ant moving too far to the right	Ant on white, pointing north, at col = grid size-1, turn 1	main() do..while validMove	Simulation aborted, completed zero turns	Simulation aborted. Completed 0 turns.
Ant moving too far to the left	Ant on black, pointing north, at col = 0, turn 1	main() do..while validMove	Simulation aborted, completed zero turns	Simulation aborted. Completed 0 turns.
Number of turns correct	White grid, grid size = [1,2], ant at [0,0], pointing north	main() do..while validMove	Simulation aborted, completed one turn	Simulation aborted, completed 1 turn.
Number of turns correct	Large grid, ant at center, 10 turns	main() do..while validMove and turn < numberOfTurns	Simulation ended successfully, completed [numberOfTurns] turns	Simulation ended successfully. Completed 10 turns.

Reflections

During the course of building the program I deviated a bit from my original design, mostly in an attempt to make my code cleaner and more readable. The logic of my program worked fine; all my test cases performed as expected. I just changed some functions a bit and re-wrote some of my class methods so that it fit my preferred style a bit better.

New Data Types

Rather than storing grid position and size as pairs of ints, I created two new structures. In my opinion this makes passing them to function parameters easier, and makes my code more readable.

```
struct Matrix2DPosition {
   int row;
   int col;
};

struct Matrix2DSize {
   int rows;
   int cols;
};
```

The Ant Class (Revised)

Originally the Ant class had int data members for the row, column, and the grid size rows and columns. First, I decided that the ant was better off not storing the value of the grid rows and columns, in case at runtime the size of the grid happened to change for whatever reason. So instead, the grid size is passed to the ant during each turn that it needs to process a move.

The Ant data members and methods have also been revised to take advantage of the new data types Matrix2DPosition, and Matrix2DSize.

nextMove()

Instead of having the program call ant.rotateRight(), ant.moveForward(), ant.updateGridPosition(), ant.changeCellColor(), I consolidated some of the functionality and exposed a new method nextMove() to the calling function. I then privatized the rotate() and moveForward() functions to keep them for internal use only. Now, nextMove() takes in the grid and gridSize as parameters on each turn, and it checks the cell color and makes its rotation and attempts to move forward. If the move fails, then nextMove() returns false to the calling program. I think this is a cleaner approach and I like that it exposes fewer methods.

The final class diagram:

Ant
- orientation: directionStates - currentSpace : char - position : Matrix2DPosition
+ Ant() + Ant(startingPosition : Matrix2DPosition, startingOrientation : directionStates) - rotateRight() : void - rotateLeft() : void - moveForward(gridSize : Matrix2DSize) : bool + updateGridPosition(grid : char *[]) : void + nextMove (grid : char *[], gridSize : Matrix2DSize) : bool

The Menu Class

The menu class that I created worked exactly as I intended, and didn't need any re-design. Perhaps in future programs I might decide that I need to add more functionality, but for this program it worked well.

Input Validation

My input validation functions worked very well, and I was pleased that they appear to capture any undesired input. In the future I might consider combining my functions into a class.

Program Flow

After turning my program into code, I was impressed with how well my design was able to function as a program. I didn't see any need to change the flow of my program, and since it was working so well I decided to add in a few extra menu items to make things more interesting. I added options to change the color of the board to all white, or all black, or randomize the cell colors. As mentioned in the revised Ant class section, I did change some of the function calls for the ant in the do..while main simulation loop. The result is a bit cleaner in my opinion but the logic didn't really change.