Bryan Konshak:

Assignment 1

10/8/2016

Planning and Design:

I'll need many algorithms to accomplish Langton's ant.

First I need a 2D array to pointers. This should be easy with a dynamically allocated array like the first lab.

I will need to develop a class that keeps track of the ant's movement and location on the board.

Ant's movement:

My board will be blank but regarding the ant's movement I'll need to:

1)

Keep track of which way the ant is facing (I will create an enum for this):

2)

Determine if the current element of the array is either "blank" or "#"

3)

If blank, ant turns right (based on direction facing).

If #, ant will turn left.

4)

Ant must be accounted for on the board so every new iteration * will replace the blank or # element.

5)

When the ant moves, the * must move to the new element

6)

If the element was blank it becomes a * and vice versa.

I will have an Ant class with two function, one for when the ant will move left and one for when the ant will move right:

Pseudocode:

Enum states for North South East and West.

This will create 8 scenarios: The ant will face one of four directions, and will move either left or right.

Let's say the ant has an enum state of North:

Whether the current square array[row][column] = " " or "#", that square will be replaced with the ant *.

If the space is an #, then the ant will be moving left, so I will have variables for rows and columns.

In this scenario I would subtract one from the column (column--).

Then I would print the array to show the current position of the ant.

After this I need to replace the element that has the ant with a blank square: array[row][column+1]= ' '.

Finally I will change the Enum direction to North.

So place the ant, change the row or column variable based on ant's enum, print the board, and then replace the ant's position with either a blank or #.

Board Design:

For the board, I'll need to suggest a size but allow the user to input variables for how many rows and columns there will be and check for input verification. I'll probably suggest a larger board so they can see patterns develop and use input validation to make sure the board is neither to big nor too small.

I'll use these inputs to create the 2d array, dynamically allocated using a for loop. These would be different than the variables that keep track of the ant:

Board variables: rowCount, and colCount.

I will then populate an empty board with character "blankspace".

Additionally I will create a function that prints the board after every turn

I also need to ask the user how many turns. This will create a variable I can use for the iterations of movements for the ant (for i=0; i<turns;i++) and execute the Ant functions for movements for each iteration.

Ant's starting position:

For starting position, I'll need a menu to let either the user pick or to make the starting position random.

The user can pick the starting position and create the variable that track the movement row and column.

The number chosen can be 0 through rowCount-1 or 0 through colCount -1 to keep it inside the board.

The user can also pick whether the starting position is North, South, East or West.

Or it can all be random: I can use srand to create random starting positions: colCount = rand() % colCount.

Menu Function

I will create a reusable menu function that offers choices 1 through up to four but easily adaptable to any program. I will create a do while loop for input validation. I will use break statements for menus.

Input Validation

I will create input validation programs using do while loops to account for input outside of the recommended range.

Recap of design:

Design the board and print the board. Create menus that allow user to select random or user inputs for board size and ant placement. Create ant class to update board and print. Create input validation functions.

Reflections:

My first major change was using Enums to keep track of direction. It wasn't necessary and I just created an antDirection function that built off my menu function to return a char and set a variable equal to either N, S, W, or E.

One very big mistake was doing too much in my main file before creating my functions. I would test and make sure the program worked, then had to rethink how it would work with a function with parameters or void. This led to a lot of variables being passed by reference (I chose this over pointers due to my comfort level) which I think could lead to potential debugging issues.

inputVal.cpp

Included functions: int inputVal() int inputCol() int antRow(int rowCount) int antCol(int colCount) and antTurns()

All of my cpp files have hpp files. My first function utilized my created input validation functions to ask the user how big they wanted the board to be. In main I make the suggestion of a 45X70 and don't allow the user to go bigger, nor do I let the board be smaller than 3X3.

At first I only accounted for numbers outside of this range, but by including !(cin ≫ input) in a while loop, I accounted for letters entered. I could not account for doubles being entered accidentally though. The text says that the user must enter in the correct data type but we know that's not going to happen every time.

I also used this input validation function for the user to pick the row and column of the ant, functions used in the menu.cpp file function antDirection.

Testing Plan:

Test 1: Test below range and verify while loop Passed

Test 2: Test above range and verify while loop Passed

Test 3: Enter text or garbage Passed

Test 4: Enter a double—only failed test.

Menu.cpp

Included function: char antDirection(int& rowCount, int& colCount)

All of my cpp files have hpp files. My first function utilized both menus and input validation. I had intended to use breaks but ended up using strings for my menu choices. This eliminated any kind of input validation issue. I commented out a generic version of my menu that can easily be applied to future projects.

There's a lot going on in this function and had wanted to decompose it more, but ran into too many bugs and kept the working working code.

1. It takes as parameters the rowCount and colCount produced by the inputVal functions.
2. It uses a menu to ask if user wants to pick the starting position or let it be random. Made this a string to help with input validation.
3. It uses inputVal functions antRow and antCol to get starting positions from user.
4. It uses another menu based on the template to ask which direction the ant will face.
5. Or it uses srand() % colCount/rowCount and srand() % 4 and return north, south, east, and west to randomize starting position.

Testing plan:

Test 1: Menus work with options given only are inputted and loop otherwise.

Test 2: All tests that did not match menu options looped back the original question (pass)

Test 3: Initial = rand() % 3: Did not produce all random directional results. Failed then fixed

Test 4: Verify the function returns char nsew that indicates to Ant class what direction the ant starts. Passed

Test 5: Only options available fit within the board. Passed

Board.cpp

Function: board(antBoard, rowCount, colCount)

This function uses the created board from main and fills it with empty spaces using rowCount and colCount, variables created by inputVal() and inputCol() using a nested for loop.

I tested different values here individually and filled the array with '.' To see it using my next function, printBoard.

printBoard.cpp

Function: void printBoard(char** board, int rowCount, int colCount)

So simple I decided not to have a Board class. I tested this to see how big I could make my 2d array when running the program.

Ant.cpp

Ant class with private 2d of pointers antBoard, int row, int column, int rowCount, int colCount, char antDir.

I did the setter functions for the private members for testing.

With six parameters, my two antLeft and antRight functions had a lot of information stored in them. My logic for the movement of the board was perfect, but the segmentation faults when the ant went outside of the array crashed my program.

I had to figure out what I was going to do with the ant in this scenario, and I decided to have the ant wraparound. In antLeft:

```
if (antDir == 'N')        {

antBoard[row][column] = '*';

if (column == 0)

    {

        column = (colCount-1);

         printBoard(antBoard, rowCount, colCount);

        antBoard[row][(0)] = ' ';

        antDir = 'W';

      }
```

Everything worked perfectly except initially in testing I didn't subtract 1 from both the initial rowCount and colCount and that produced segmentation faults. Everything else was perfect. I made the board 3*3 to test and trace the turns and verify the ant was going in the correct direction.

Test 1: segmentation fault off the screen

Test 2: Ant board works as expected. Wrap arounds North South East and West Passed

newMain.cpp

This is what happens when you build too much in your main.cpp before creating function…you end up renaming it newMain.cpp‼

My main file:

1. Calls inputVal() and inputCol to create variables to initialize the 2d array.
2. Initializes the 2d array
3. Calls the board function to create an empty 2d array or pointers.
4. Asks how many turns in main.

5. Creates variable row and column and assigns them rowCount-1 and colCount -1 respectively.
6. Runs antDir function to determine starting point of ant (and if its random)
7. Creates the Ant object tracker.
8. Uses usleep(105000) and \33[2J\033[1;1H to clear the screen in a for loop based on input of turns
9. Runs tracker.antRight if the element of the array is blank and runs track.antLeft if it is #.
10. Watch Langton's ant‼

Test 1: Verify turn number is accurate. Failed initially—had to add one to the turns variable.

I tested every function individually and have a working program. I wanted to flesh out the Ant class better and I think looking back having my menu.cpp function with the antDirection function could have been rolled into the Ant class.

Again the main issue I had were although testing as I went, I built too much into my main function and then started breaking it out into functions.

I think my code could have been cleaner. Passing parameters that needed to be changed in return functions was not ideal, and it would have been better to return multiple variables by a class or structure.