Kenneth Hall
CS 162 400 W2018
1-21-2018

Design + Reflection

LANGTON'S ANT RULE (per turn):

-move forward
-if land on white space, rotate clockwise, change space to black
-if land on black space, rotate counterclockwise, change space to white

ANT CLASS:

-Board: Because all spaces on the board are either ' ', '#', or '*', I will represent the board as an array of pointers to char arrays (a 2D array), where the array of char array pointers represents the rows of the board, and the arrays that are pointed to by the pointers each represent a row. Using pointers will allow me to dynamically allocate for a grid of any size. User will specify a number of rows and columns, each of which must be greater than or equal to 2, so that the ant can "bounce" off of the edge of the board if it runs into an edge.

-Ant's location: Randomly generated or defined by user to begin with; a pair of ints [x][y] where x is the ant's location (element #) in the array of pointers, and y is ant's location within the array pointed to by x. Enforce that :::(-1) < x,y < (rows - 1),(columns - 1):::

-Ant's orientation: ~~-Store ant as a char representing its orientation (different char for up, down, left, right) for easier debugging of logic errors.~~ Revision: reread program requirements...ant must be printed as '*' char at all times. I still like the idea of representing orientation as a char and am going to record orientation the next move of an Ant object using the following notation: '^' = up; 'V' = down; '>' = right; '<' = left.

LANGTON'S ANT PROGRAM:

"The ant starts at a user specified location on the board": prompt the user for two ints as suggested above to represent ant location. Initial direction of the ant will always be '^'.

"During each step, the program should print board.":

```
for(each_step)
{
        makeMove()
        printBoard()
}
```

The number of steps is given by the user, and is an argument passed in the creation of an Ant object.

Ant member function makeMove(): the direction the ant moves depends on…

1: if the ant is about to walk off the board
    -If the ant is about to walk off the board, rotate it 180 degrees
2: what direction the ant is facing

-if ^, the ant goes from currentRow to currentRow - 1
-if >, the ant goes from currentColumn to currentColumn + 1
-if <, the ant goes from currentColumn to currentColumn - 1
-if V, the ant goes from currentRow to currentRow + 1
Revision: Was seeing very strange behavior with ant cloning itself and then running around in circles endlessly. Discovered that multiple rotations/movements were occurring per turn. Added a boolean flag movedThisTurn, that is set to false at the start of each turn. When the ant moves, movedThisTurn is set to true. Each movement possibility verifies that !movedThisTurn before executing. Thus only one move is possible per turn.

For makeMove, we also need to 'restore' the square currently being obfuscated by the ant, and store the type of square the ant should leave behind after it moves from the square to which it is currently about to move. To this end, the Ant class also needs a member variable squareUnderAnt as intermediate storage so that we can keep track of the square the ant is on and still print the ant on that square.

Thus makeMove has an order of execution described below:
Set movedThisTurn = false
If ant is about to go out of bounds, rotate 180 degrees.
Update the space that the ant is on with the space stored in squareUnderAnt
Based on ant orientation and movedThisTurn, identify target square
Store opposite of target square in squareUnderAnt
Update ant orientation based on target square
Update char value representing ant's current location to '*'
Update int value representing ant current location to match target square

Ant member function printBoard(): fairly straightforward, print white squares as ' ', black squares as '#', and the ant square as '*' per specs, and throw in an endl at the end of each row. Will use for loop nested in a for loop to access each element of the 2D array.
Revision: added "=" to the top of the board and "|" to the sides to better define the edges of the board

```
for(each_row)
{
    Print row
    endl
}
```

MENU:
Menu class will take between 2 and 5 string arguments (need at least 2 items for a choice to be made, and 5 seems like a good amount of possible choices. Can add more constructors for larger menus later if needed.).

Member function int enforceValidInput(): counts the number of items in the menu object, and prompts the user to make a selection based on that number of items. If user enters input that starts with invalid characters, ignore all of input and try again. If user enters input that starts with valid characters and thereafter includes one or more invalid characters, accept input up through last valid character and discard the rest. User will be prompted for next input, and notified that something went wrong on their last input. Plan on using a combination of while loops, cin.clear(), cin.ignore, cin.fail(), the basic boolean logic operators, and basic comparison operators.

Member function displayMenu(): returns each item of the list preceded by the "number" it is on the list,as is given by the member variable, i.e. the item passed to item1 will be preceded by 1: when displayMenu() is called. displayMenu() provides the necessary context for the user to make a selection when enforceValidInput() is called.

For this project, our main function only needs one menu object, which will be passed strings resembling "Play", "Play with a random start location", "Quit"

displayMenu() would then cause the following output when called for that object
1: Play
2: Play with a random start location
3: Quit

enforceValidInput() called for that object would then ask the user to enter a value, and will continue doing so until a value is recognized 1, 2, or 3.

MAIN:
Revision: #include <unistd.h> for more natural turn elapse using usleep()
Build Menu object

Until player decides to quit
{
    Set all arguments back to default values
    Ask the user if they would like to play, play with random start, or quit
    if(user_decides_to_quit)
    {
        Return 0;
    }
    Ask the play for number for number of rows, columns, and steps.
        -Rows and columns enforced to be greater than or equal to 2
        -Steps must be greater than or equal to 1
    Revision: for some reason I flipped the order on the if/else. Shouldn't make any difference.
    If(user_chose_random_start)

```
        {
                -Assign a random starting location (between 0 and numberOfRows/Columns - 1)
        }
        Else
        {
                Get starting location from user.  Same boundaries as random starting location.
        }

        Create the Ant object passing all arguments gathered by main()

        Call Ant.printBoard() to print the initial board before any moves have been made

        for(each_step)
        {
                sleep() (to make the turns feel more organic and flow one at a time)
                Ant.makeMove()
                Ant.printBoard()
        }
}
```

TEST:

@menu prompt

Test Input	Expected Output	Actual Output	Notes
asdf	Reject input, prompt again	Reject input, prompt again	
-2	Reject input, prompt again	Reject input, prompt again	
2.1	Accept input as 2	Accept input as 2	(.1) being passed to next cin...would be a problem if we weren't working exclusively with ints
7	Reject input, prompt again	Reject input, prompt again	
1	Begin simulation	Begin simulation	User def start loc
2	Begin simulation	Begin simulation	Rand start location REV: added <time> to seed rand()

3	Quit (return 0)	Quit (return 0)	

@Ant rows argument prompt

Test Input	Expected Output	Actual Output	Notes
asdf	Reject input, prompt again	Reject input, prompt again	
-2	Reject input, prompt again	Reject input, prompt again	
2.1	Accept input as 2	Accept input as 2	Like above...(.1) being passed to next cin...would be a problem if we weren't working exclusively with ints
1	Reject input, prompt again	Reject input, prompt again1	
5	Accept input as 5	Accept input as 5	
10	Accept input as 10	Accept input as 10	
100	Accept input as 100	Accept input as 100	

@Ant columns argument prompt

Test Input	Expected Output	Actual Output	Notes
asdf	Reject input, prompt again	Reject input, prompt again	
-2	Reject input, prompt again	Reject input, prompt again	
2.1	Accept input as 2	Accept input as 2	Like above...(.1) being passed to next cin...would be a problem if we weren't working exclusively with ints
1	Reject input, prompt	Reject input, prompt	

	again	again	
5	Accept input as 5	Accept input as 5	
10	Accept input as 10	Accept input as 10	
100	Accept input as 100	Accept input as 100	

@Ant steps argument prompt

Test Input	Expected Output	Actual Output	Notes
asdf	Reject input, prompt again	Reject input, prompt again	
-2	Reject input, prompt again	Reject input, prompt again	
2.1	Accept input as 2	Accept input as 2	Like above...(.1) being passed to next cin...would be a problem if we weren't working exclusively with ints
1	Accept input as 1	Accept input as 1	
5	Accept input as 5	Accept input as 5	
10	Accept input as 10	Accept input as 10	
100	Accept input as 100	Accept input as 100	

@Ant start row prompt

Test Input	Expected Output	Actual Output	Notes
asdf	Reject input, prompt again	Accepted input with rows = 5	Not crashing...but very weird behavior
-2	Reject input, prompt again	Reject input, prompt again	
2.1	Accept input as 2	Accept input as 2	Like above...(.1) being passed to next cin...would be a problem if we weren't working exclusively with ints

Equal to rows	Reject input, prompt again	Reject input, prompt again	Just above upper boundary
Equal to rows - 1	Accept input	Accept input	Upper bounds
0	Accept as 0	Accept as 0	Lower bounds

@Ant start column prompt

Test Input	Expected Output	Actual Output	Notes
asdf	Reject input, prompt again	Accepted input with rows = 5	Not crashing...but very weird behavior
-2	Reject input, prompt again	Reject input, prompt again	
2.1	Accept input as 2	Accept input as 2	Like above...(.1) being passed to next cin...would be a problem if we weren't working exclusively with ints
Equal to columns	Reject input, prompt again	Reject input, prompt again	Just above upper boundary
0	Accept as 0	Accept as 0	Lower boundary
Equal to columns - 1	Accept input	Accept input	Upper bounds

REFLECTION:
I still don't know how to deal with "asdf" being accepted at some of the prompts. I'm thinking of revising my input validation technique to something like:

```
Datatype foo
Cin >> placeholder;
while(cin.fail())
{
        cin.clear(100, '\n')
        cin.ignore()
        Cout << whoops try again
        Cin >> placeholder
}
foo = placeholder
```

I think that would prevent “good bits of bad input” from seeping through when they really shouldn’t. Another route I am considering is letting the user know what just got stored inside of the variable they were just prompted for, and then give them the opportunity to go back and change it if they want.

As was mentioned in the revision notes of the LANGTON’S ANT PROGRAM section, I was stumped for a while on a very strange behavior where the ant appeared to “clone” itself and slowly the board became filled with “ants” that eventually started running in circles around each other. I discovered that the sequence of if statements I had created sometimes led to “chains” of movements on a single turn. I.e.

```
if(ant_pointing_up)
{
        If next square is white square
                Turn right
}

if(ant_pointing_right)
{
        If next square is white square
                Turn right
}
```

Both of the above statements could trigger on the same “turn” until I added a flag to mark when the ant had already moved for the turn, and subsequent “opportunities” to move should be ignored.

```
if(ant_pointing_up && !already_moved)
{
        If next square is white square
                Turn right
}

if(ant_pointing_right && !already_moved)
{
        If next square is white square
                Turn right
}
```

I started out with a totally blank board with no borders and almost left it that way, but I thought it looked much better with some borders, so I figured out how to stick some borders on it. Initially, I got the top-bottom border mixed up with the left-right border, so I had some very bizarre looking boards for a few revisions, but after a little tinkering I eventually got it looking right.

Initially, I wanted to have a dynamically updated ant character that corresponded to the ant's orientation, but after rereading the specifications, was disappointed to find that we had to use '*' for the ant no matter what the orientation was. I still managed to implement the idea in part by storing the ant's orientation as a char (at least ^, V, <, > are all intuitive orientation representations to me). It's likely I saved myself 200 keystrokes or so by using chars instead of strings i.e. "up", "down", "left", "right" but here I am writing about it, so I guess I didn't really save myself the keystrokes after all!

I could talk a lot more about what I learned in vim and how I'm excited to being able to reflexively do things like jump to the end of any particular line with <line#> <shift> G $, but this pdf is creeping up on 9 pages long and you probably have a lot more to grade (and I don't think that's exactly in the scope of "what you learned for this project"). In short, I had a lot of fun on this project, and I'm really looking forward to the rest of the class!