CS 162 – Assignment 1
Langton's Ant

Daniel Beyer
Dan.beyer@gmail.com
04/06/2016

Requirements

Implement a simulation of Langton's ant, a simulation of a cellular automaton on a matrix of cells. Langton's ant follows 2 simple rules:

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

The program will prompt the user for the number of rows and columns for the grid, the number of moves to perform, and if they wish the ant to be placed randomly on the grid or if they want to input their own starting coordinates. The ant will then begin facing a random direction and continue to move until it has completed the specified number of moves.

Program Input

User enters:

1. Number of rows
2. Number of columns
3. Number of moves for the ant to make
4. Choose random location or not?
    a. If not random location, user enters row and column coordinates for starting position.

Program Output

Program displays a grid of the specified size that updates each time the ant moves until the ant has completed the specified number of moves.

Design

Main method pseudocode

Declare variables rows, columns, numMoves

Prompt user for number of rows
    If rows < 0, prompt for new positive number
Prompt user for number of columns
    If columns < 0, prompt for new positive number
Prompt user for number of moves to make with suggestions such as a few hundred or 10,000
    If numMoves < 0, prompt for new positive number

Set rows2 = rows+2
Set columns2 = columns+2

Create Grid object and fill with " "
Create Ant object

Prompt user with Ant menu:
    Does user want ant placed randomly?
        If Yes, send random ant coordinates to Ant
        If No, prompt user for starting coordinates and send to Ant

Run move function to begin ant moves

Call *delete* to free allocated memory and return it to the heap

Grid class methods

Private Variables:

- Dynamically created 2d array of chars
- Int number of rows
- Int number of columns

grid()
Class constructor that dynamically allocates the 2d array

fill()
Initializes grid to all ' ' (white spaces)

setColorWhite()
Sets color of current array location to ' ' (white)

setColorBlack()
Sets color of current array location to '#' (black)

getColor()
Gets color of current array location (white or black)

updateLocation()
Takes new array coordinates as parameters and changes that array location to an ant '*'.

remove()
Free allocated memory and return it to heap

print()
Print grid with current ant location to the screen

Ant class methods

Private Variables:

- enum direction North, South, East, West
- int moves, numMoves
- int row coordinates, column coordinates
- char space (to save the color of the current occupied cell)

ant()

Ant class constructor

- set direction to random enum from North, South, East, or West
- sets initial coordinates to [0][0]
- sets initial space color as ' '(white)
- sets moves to 0

move()

- print initial blank grid
- while (moves<use specified total number of moves)
    - switch(direction)
        - Case N:
            - If going into white cell, move row-1 and turn E.
                - If previous cell was white(space = ' '), turn it black
                - If previous cell was black (space = '#'), turn it white
                - Set space = ' '
                - Pass new coordinates of Ant to Grid via updateLocations
            - If going into black cell, move row-1 and turn W
                - If previous cell was white(space = ' '), turn it black
                - If previous cell was black (space = '#'), turn it white
                - Set space = '#'
                - Pass new coordinates of Ant to Grid via updateLocation
            - Save value
        - Case S: Repeat above steps
        - Case E: Repeat above steps
        - Case W: Repeat above steps
    - Moves++
    - Print Grid

menu()

Menu function to determine if ant is placed randomly or at user-defined coordinates

- Prompt user if they would like to have ant placed randomly or not (Y or N)
    - Switch(answer)
        - Case Y: Randomly generate coordinates based on number of rows from Grid object and pass them to int row and int coordinates in Ant
        - Case N: Prompt user for row and column coordinates with input validation that the number is > 0 and < number of rows and columns.

setRowCoord()
Sets row coordinates of ant

setColCoord()
Sets column coordinates of ant

setNumMoves()
Sets total number of moves based on user-defined value in main method

Testing

Test: Invalid number of rows or columns

- Input: Number < 0
- Expected Output: "Please enter an integer greater than 0", reprompt for number
- Program outputs as expected

Test: Invalid number of moves for ant

- Input: Number < 0
- Expected Output: "Please enter an integer greater than 0", reprompt for number
- Program outputs as expected

Test: Invalid user-specified starting coordinates for ant

- Input: Number > number of rows or columns
- Expected output: "Please enter a number greater than 0 and less than the number of rows or columns", reprompt for number
- Program outputs as expected

Test: Invalid input for Ant starting location menu

- Input: Non-char input
- Expected output: "Invalid input, please enter Y/y or N/n", reprompt for input
- Program outputs as expected

Test: Insert Ant to random coordinates in Grid

- Input: Select 'Y' to have Ant placed randomly on Grid
- Expected output: Ant starting location coordinates are random

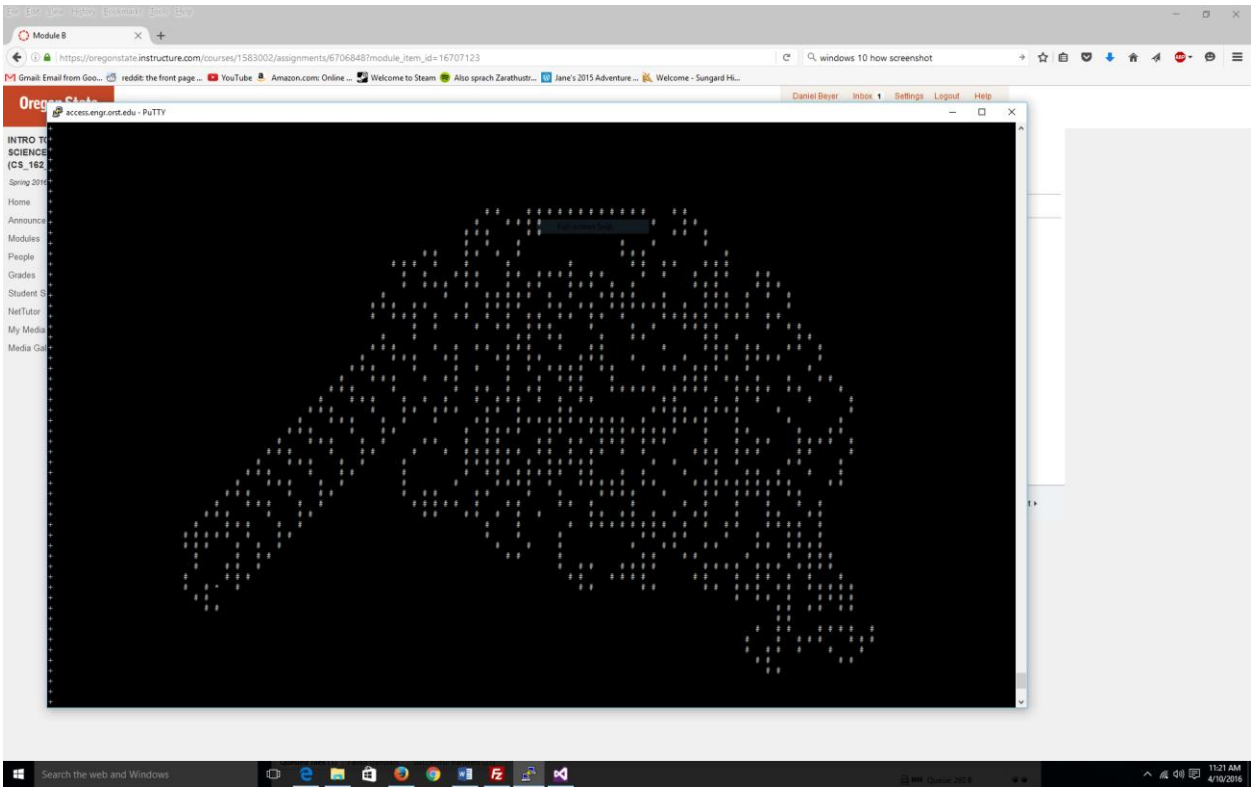
- Program outputs as expected. Multiple executions resulted in starting coordinates of [2,10], [20, 4], [1, 6], and [31, 17].

Test: Ant movement follows rule of Langton's Ant as laid out at the beginning of this document

- Input: Grid of 50x50, starting Ant location of [25,25], 300 moves
- Expected Output: Ant moves in simple, sometimes symmetric pattern and follows the 2 rules for Langton's ant movement.
- Program outputs as expected. When Ant moves to a new cell that is black, it changes its direction 90 degrees to the left and changes the cell to white when it leaves. When it moves to a white cell, it changes its direction to the right by 90 degrees and changes the cell to black when it leaves.

Test: Ant movement shows "highway" pattern after 11,000 steps

- Input: Grid 100x100, starting Ant location of [25,25], 11,000 moves
- Expected Output: Ant shows recurring "highway" pattern near 11,000 moves.
- Program outputs as expected, after 11,000 moves it showed a distinct "highway" pattern appendage extending towards the lower right of the screen.



Test: Ant hits border of grid before finishing all moves

- Input: Grid 10x10, starting Ant location random, 500 moves
- Expected Output: Ant reverses direction when hitting border
- Program outputs as expected. Setting a variable = enum for each loop of the move() while loop then printing out the current enum after every loop shows the Ant reverses direction when hitting the border.

Reflection

My initial design seemed to work well, but there were a few changes that I implemented in the final code.

When designing the move() function in my Ant class, I was unsure how I wanted the ant to react when it hit a border. I knew I wanted to keep the simulation running, but I was not sure if I wanted the ant to wrap around to the other side or have the ant simply stop and retrieve a new random direction or have the ant reverse direction. And I was unsure how any of these options would affect the simulation overall. In the end I opted to have the ant set a new reverse direction opposite of the border it was hitting. I ran a few simulations past 11,000 moves which produced the "highway" pattern so I was satisfied this was a reasonable option to go with.

I initially did not anticipate the need to "animate" the ant using a partially cleared screen and time delay, so I did not document this in my initial design. After receiving feedback on my Module A that this was recommended, I tried various techniques for clearing the screen and implementing a time delay. I knew I did not want to completely clear the screen because in the assignment description it states we need to display each step, but I wanted to provide some space between each grid to make the screen "cleaner". I finally settled on using `cout << string(10, '\n');` for clearing the screen and my sleep function to pause the program and act as a time delay. I chose this specific sleep function because it uses standard libraries that will be available on Windows and on Flip.

I also struggled with input validation initially. I had no problem validating if an integer entered was too small or too big, but I did not know how to proceed if a char or non-integer was entered. Utilizing online sources I tried using !cin, but I got a repeating response after pressing "enter". I finally found using `cin.clear(); cin.ignore();` resolved this problem nicely.

When designing my Ant class I also did not originally anticipate the need to specify whether the cell currently inhabited by the ant was black or white. I originally thought I could determine the color of the space by the direction the ant took when leaving the space, then change the color back appropriately, but this turned out to not be the case. Instead I created the char *space* variable that is set after every move of the ant to save the color of the space the ant is moving into. After each move, the color of the previously inhabited cell is reset to the opposite of whatever color is saved in *space.*

This was an interesting program to design and implement. It was incredibly gratifying to run the simulation past 11,000 moves and to see the distinct “highway” pattern emerge!