Homework 2: An A-MAZE-ING Assignment! Wait, my bad, I meant to say an “amazing” assignment.

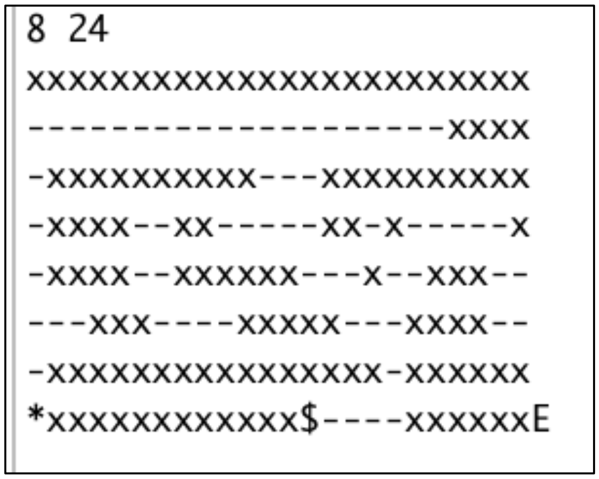
Written by Ashok Basawapatna

2 Part Assignment; 1 part is required, the other is extra credit.

START EARLY. It’s not too bad but there are some new concepts. Pseudocode due Wednesday at Midnight.

<u>**Intro:**</u> Y’know that thing when you come back from data structures class, and you’re just lounging around the apartment/dorm, and you realize that you’ve completed all those Sudokus in that puzzle book. So you bust out that book of Mazes. And then you’re trying to solve one of them, and 30 minutes into it you suddenly say to yourself “Hold up, why am I doing this?! I HATE solving mazes! Wish this crap would solve itself!” Real talk ya’ll. Real talk.

Well now your prayers are answered. In this assignment we will use recursion and dynamic allocation of memory to create a program that can solve any maze of any dimension as long as it’s in the following form.

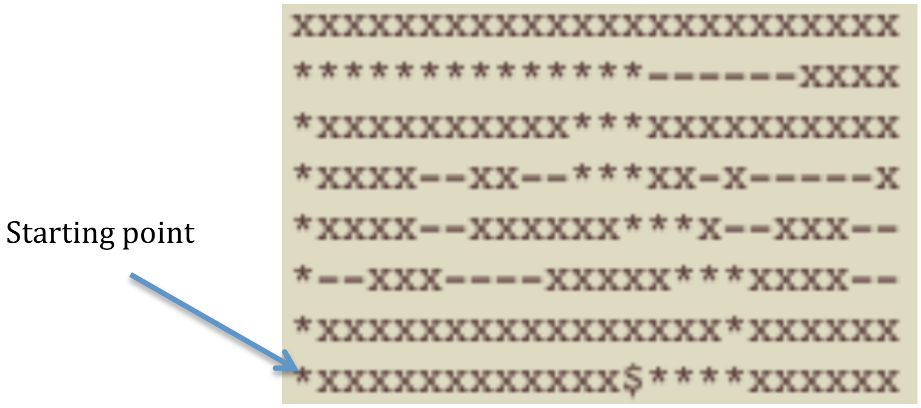


```
8 24
xxxxxxxxxxxxxxxxxxxxxxxx
--------------------xxxx
-xxxxxxxxxx---xxxxxxxxxx
-xxxx--xx-----xx-x-----x
-xxxx--xxxxxx---x--xxx--
---xxx----xxxxx---xxxx--
-xxxxxxxxxxxxxxxx-xxxxxx
*xxxxxxxxxxxx$----xxxxxxE
```

Figure 1: A sample maze we will solve

Figure 1 is the ascii maze that’s included with your assignment.zip. At the top of Figure 1 is two numbers that tell you the dimensions of the maze (yDim=8, xDim=24). The “x” ascii characters represent the walls of the maze. The “-“ ascii characters represent places you can walk. The “*”

character represents the starting point. The “$” character represents the goal you want to reach in our capitalistic society. Finally the “E” character is the end of the maze (not really necessary since we have the dimensions). The goal of this assignment is to create a path of “*” leading from the starting point to the goal. One (not the only) solution looks as follows (spoiler alert):



```
XXXXXXXXXXXXXXXXXXXXXXXX
**************------XXXX
*XXXXXXXXXX***XXXXXXXXXX
*XXXX--XX--***XX-X-----X
*XXXX--XXXXXX***X--XXX--
*--XXX----XXXXX***XXXX--
*XXXXXXXXXXXXXXXX*XXXXXX
*XXXXXXXXXXXX$****XXXXXX
```

Figure 2: Solution To Maze Depicted In Figure 1

Your homework file has 2 things in it, this maze.txt file and a mainProgram.cxx file that instantiates a maze, prints the unsolved maze, calls maze.init() (see below) and then prints the solved maze.

You will make 2 files: A maze.h file and a maze.cxx file.
Your maze.h file will have the following specifications

<u>Private Variables</u>: I had 4 private variables; they are as follows.

1) char* mazeArray: we represent the maze as a 1 dimensional dynamically allocated array that this pointer will point to.
2) startingIndex: an int that represents the starting point of the maze.
3) xDim: The x-dimension of the maze.
4) yDim: The y-dimension of the maze.

<u>Member functions:</u> Feel free to add any member functions you find helpful. I used the following members to solve this.

<u>Constructor:</u> I have a default constructor maze() that does the following

- Creates an ifStream myIns and opens "maze.txt"
- If maze.txt doesn't open (check with myIns.is_open()) the program stops.
- Reads in the x Dimension and y Dimension into xDim, yDim variables respectively
- Dynamically allocates a char array and points mazeArray ptr to it. (ie: mazeArray=new char [xDim*yDim]
- Puts all the maze chars into this new array (i.e. some sort of loop or double loop maybe?)

<u>Other Public Members:</u>

void init():

- Finds the starting position denoted by the "*" character and puts it into the startingIndex private variable
- Calls solve(startingIndex)

bool solve(int index):

- This is where we Recurse, so we
    - First temporarily change the character at the current index of the maze array to "v" to denote that we have visited this spot (so it doesn't get visited again) in our mazeArray.
    - Check up, down, left, and right for the goal: "$". (note up will be your index-xDim, down will be your index+xDim, left will be your index-1, and right will be your index+1). Remember when checking up and down to make sure you haven't gone off the end of your array! Also when checking left and right make sure there is a left and a right (I used the modulo '%' with the xDim to solve this….think about it). If you are at the goal change the character at this index of the maze array to "*" and return true.
    - If you're not at the goal, check the up location for a "-" character (which means we can move there). If that exists, call the solve function passing in the up index, and if that returns true, change the character at this index of the maze

array to “*” and return true (it means that the up gives us a valid path to the goal!)
  - else if do the same with down (i.e. we only do this if up is not a valid path to the goal)
  - else if left, else if right.
  - If none of these return true we change our current character back to “-“ and return false.

<u>NON MEMBER FUNCTIONS</u>

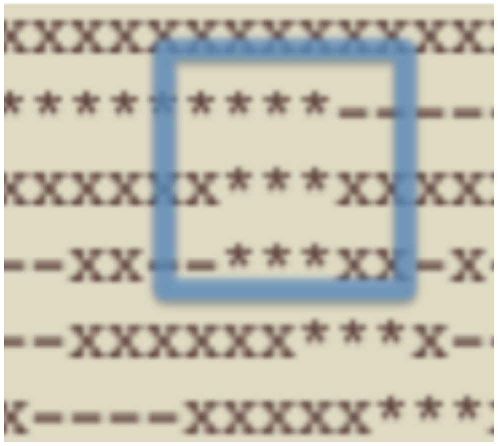
friend std::ostream& operator <<(std::ostream& outs, const maze& source)

- Finally we overload the ostream operator and have it print out the maze. I used a double for loop for this.

You will turn in: maze.cpp, maze.h, and any files you create/used to test+ a readme.txt that outlines bugs, implementation strategy etc.

<u>Extra Credit:</u>

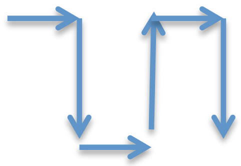
So once you get past the new concepts the above maze is pretty simple to accomplish. The extra credit is as follows. Notice this portion of the maze solution (yours might be different but it will still have this general situation in it)—it’s the top portion of Figure 2.



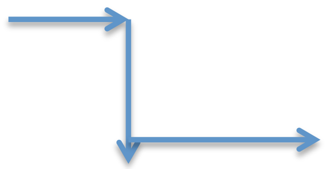
```
XXXXXXXXXXXXX
*********----
XXXXXX***XXXX
--XX--***XX-X
--XXXXXX***X-
X----XXXXX***
```

Figure 3: The Top Portion Of Figure 2

What in the hell is going on here?! The direction our person is taking through the maze at this point, since we check the up direction first in solve, is as follows.



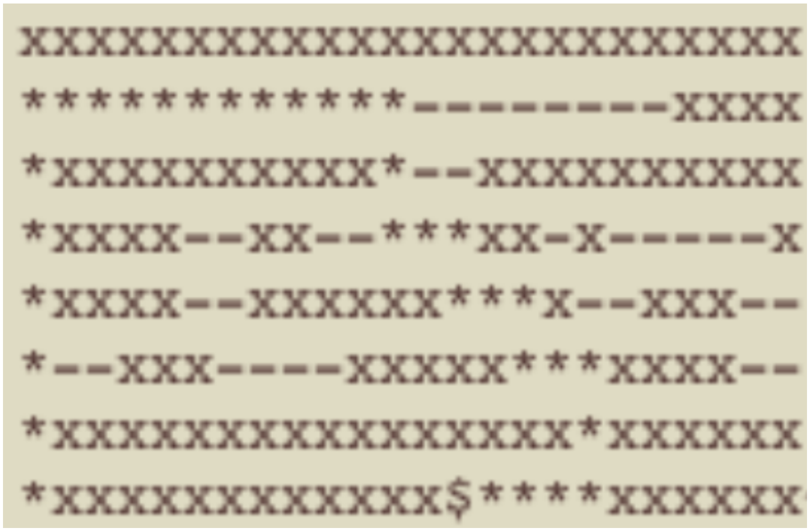
Instead of something more sensible like:



The extra credit portion involves the following (I'm going to describe it generally, try it, and if you have trouble hit me up and I'll try to help): you will create a linked list of your path, and then refactor the path to get rid of these weird u-turns. You will probably have to add a "headPtr" private variable among other private variables to your class to manage your Linked List. Chapter 5 tells you how to make a node class and implement a linked list (we'll also talk about this in class later this week). You don't have to implement everything in this class, just the things you need.

- Create a node class that has 2 members, a current index (your data), and a pointer to the next node (your link).
- Add a node* to your private variables in maze.h (also remember to include your node.h in this). This will be your linked list.
- Every time you set a space on the maze to "*", add it to your linked list (i.e. create a node, set it's data to you're the current spaces index, set the previous end nodes link to this node, and set this nodes pointer to NULL).

- Check every spot for adjacencies in your path that don't happen immediately. I.E. I go through every element of the Linked List and check forward 2 spaces, then 3 spaces, then 4 spaces….If one of those spaces happens to be adjacent to my current space; I change link of the node representing my current space to that space (I should really delete the nodes in-between too but you don't have to).
    - So, contrived example, but if I had the following indexes in my path (and my x Dimension and y Dimension of the maze was 3, let's say): 6,3,0,2,1,4,7.-->I would point the node with the index of 6 to the node with the index of 7, jettisoning indexes 3,0,2,1,4 on my path (draw out a 3 by 3 grid, number the indexes, you'll see what I mean).
- Redraw your maze, "*"-ing only the indexes on your now refactored linked list path (I made a "refactoredMazeArray" private variable that was the original maze so I could alter it separately from the other mazeArray).
- At the end I'll have a Linked List that represents the path without any weird loops in it. It should look as follows:



```
XXXXXXXXXXXXXXXXXXXXXXXX
************--------XXXX
*XXXXXXXXXX*--XXXXXXXXXX
*XXXX--XX--***XX-X-----X
*XXXX--XXXXXX***X--XXX--
*--XXX----XXXXX***XXXX--
*XXXXXXXXXXXXXXXX*XXXXXX
*XXXXXXXXXXXX$****XXXXXX
```

Figure 4: Maze With Refactored Path

Try it out and if you have a better solution go for it (but try to use the linked list somehow….). Good luck!