EECS 268

Lab6 Writeup

Chris Watkins

1. Before starting implementation, design your code base. Design a class that will be in charge of reading the maze file. Make a list of methods and member variables you think it should have

mazeReader class needs to:

- check all necessary aspects of maze
- if file is present/not
- if the dimensions are valid for maze
- check starting position is inside maze bounds
- check maze for exit - if not present we cannot find solution to maze

functions:

- constructor
  - inputs data from file
  - throws exceptions for invalid cases (mentioned above)
  - creates 2D array of chars representing maze
- run
  - simply prints the constructed maze to the terminal
- destructor
  - deleting the created arrays on heap

exception class

- be able to handle the exceptions listed above
- derives from std::exception

2. Design a class that, given a valid maze, will traverse it. Make a list of member variables and method you think it should have.

mazeWalker class

functions

- constructor
  - is passed in a maze that was constructed in the mazeReader class
  - throws exception is starting point is in an X position
  - checks to see whether exit is present
  - creates an array of integers that will be used for the output of solved array
- destructor
  - deleting the created arrays on heap
- printMaze

    - prints the output array of integers with solution path
- solve
    - finds the solution path to the maze if one exists
    - will use recursion and check directions in clockwise order
- run
    - outputs the dimensions, and starting point of the array before calling the solve method

3. Discuss how you plan to detect the need for backtracking should you reach a dead-end in the maze. What will variables/objects/arrays will need to be updated when you backtrack?

My solve function will mark the index that it is currently at with ‘0’ before proceeding to check (up, right, down, left) whether a neighboring index is either a path or the exit index, and if so it will then call the solve function on that index and transfer control to this new method. This repeats until it reaches an index that has no possible moves (dead end) and then will simply return control to the previous function call of solve method beforehand (on the previous index). This process will repeat until the index with a path open can be followed and will eventually find a solution if one exists.

4. Discuss how you plan to detect and handle finding an exit or running out of places to traverse

I will check the maze initially using nested for loops to see whether an ‘E’ character is present. If not, I will throw an exception and not continue the program. Similarly, the structure of my solve function mentioned above easily deals with a case in which there is no path to an Exit in the maze. If no exit is reached, and hence, there is a dead end, then my solution maze will never be called because I will have my printMaze function only called if the specific index I am iterating over is equal to ‘E’. So if the ‘E’ is reached I will turned a Boolean value true, otherwise if it remains false I will print an error message and allow the recursion to simply hit the dead ends before giving control back to main.