

## CISC 181 Project 3

### Maze Traversal

(from exercises 5.25, 5.26, and 5.27 in  
Deitel and Deitel, *C++ How to Program*).

Due: At 11:00am (class time) on Tuesday, April 26th, 2005.

This project basically consists of problems 5.25, 5.26, and 5.27 in the text. These problems explain how to represent a maze, solve it (by finding a path through it), and generate random mazes of random sizes. You should do the problems one at a time to build up to the larger project.

## 1 Traversing a Maze – 5.25

Let's take the problems in order here. First, 5.25 asks you to represent a maze and write a program that is able to traverse the maze.

What is a maze? It is a 12X12 character array containing '.' (at places you can go) or '#' at places where you can't. There is going to be a specified entrance to the maze (this is going to be along the edge of the maze and will be given as the subscripts in the array for the position that is the start – you probably want to make this starting position a global variable). You may assume that there is exactly one entrance and exactly 1 exit to the maze.

So, for example, the following is the maze given in the book. You probably want to use this (or another) specific maze initialized in your main program in order to get the mazeTraversal function running correctly.

```
# # # # # # # # # # # #
# . . . # . . . . . #
. . # . # . # # # . #
# # # . # . . . . # . #
# . . . . # # # . # . .
# # # # . # . # . # . #
# . . # . # . # . # . #
# # . # . # . # . # . #
# . . . . . . . # . #
# # # # # # . # # # . #
# . . . . . # . . . #
# # # # # # # # # # #
```

Solving the maze means finding a path through the maze which will take you to a position at an edge of the array (where that position is NOT the Start position). I.e., you know you have solved the maze if you get to an edge again (after you have made the first move).

The book indicates that there is a simple algorithm for finding a path through a maze to the exit (if there is one). If there is not an exit, the algorithm will take you back to the start location. The way the algorithm is described is:

Place your right hand on the wall to your right and begin walking forward. Never remove your hand from the wall. If the maze turns to the right, you follow the wall to the right. As long as you do not remove your hand from the wall, eventually you

will arrive at the exit of the maze. There may be a shorter path than the one you have taken, but you are guaranteed to get out of the maze if you follow the algorithm.

OK – so what does this algorithm mean in terms of an implementation? What it means is that if you are in a particular place in the maze, you are going to try to move to a new position in a specific order. Let’s assume that you enter the maze from the left-hand-side, then you want to try to put your hand on the right-hand-wall. This will mean that you will first try going down, then right, then up, and then left. I.e., if there is a '#' in the square below you (so you can’t move down), try moving right. If that fails, try moving to the square above instead. If that fails, try moving left (thus going back the way you came in).

Notice that you always try to find a way to go in the same order – but where you start in the order depends on where you enter the square from. E.g.,

- if you are moving DOWN into a square, you will first look for the wall on the LEFT (then DOWN, then RIGHT, then UP);
- if you are moving RIGHT into a square, you will first look for the wall going DOWN (then RIGHT, then UP, then LEFT);
- if you are moving UP into a square, you will first look for the wall on the RIGHT (then UP, then LEFT, then DOWN);
- if you are moving LEFT into a square, you will first look for the wall going UP (then LEFT, then DOWN, then RIGHT).

Do you sense a pattern? You probably want to make the direction an enum that you cycle through trying to find a direction in which to go. You “start” into the cycle in a different place depending on the direction from which you came.

Let’s look at some functions you will want to write:

`mazeTraverse(maze, size, x_coord, y_coord, direction)` – takes a maze, its size (assuming it is square – for the last part you would need to make this two sizes since it need not be square then...), the coordinates of the current position (e.g., the array subscripts for the current position) and the direction that you want to try first for your next move Essentially, this function will (1) put an X in the place where you are, (2) print the current maze, (3) decide if the maze has been solved (because the current position is an edge that is not the start state), (4) check to see if the current position is the start position (arriving back at the start means the maze can’t be solved), (5) cycle through the direction possibilities. For each that would give a valid move, make a recursive call which will involve taking a step in that direction. Recall that a valid move would involve moving to an adjacent block that does not have a '#' in it.

`validMove(maze, x_coord, y_coord)` – takes a maze and two subscripts and determines whether or not it is OK to move to that position. (The return type of this function should be bool.)

`isSolved(array_size x_coord, y_coord)` – assuming the maze is square, it takes its size and a particular x and y coordinate, and determines if the coordinates are an exit from the maze. Recall that the exit will be on an edge of the array. (The return type of this function should be bool.)

`printMaze(maze, size)` – prints out the maze.

## 2 Generating a Maze – 5.26

Part 2 (5.26) involves generating a maze. Here is a header for this function:

```
mazeGenerator(char maze[ ][size], size, int *xPtr, int *yPtr)
```

It takes the maze itself, its size (again I am making the assumption it is square here), and two reference parameters which will be set by the function to indicate the starting location in the maze.

This function involves:

1. initializing variables (e.g., you might initialize the maze to contain all '#')
2. randomly choosing a starting location (and putting a '.' there)
3. randomly choosing an exit location (and putting a '.' there)
4. randomly putting '.'s in the interior places of the array (where you have some max number of '.'s you would put)

Notice that you don't need to make sure the maze is "solvable" – the mazeTraversal function may only be able to find its way out the entrance and that would be OK.

## 3 Mazes of any Size – 5.27

For part 3 (5.27) – you will need to generalize your functions to take non-square arrays. It is quite fine to actually set the array-size using a couple of global variables and thus compile the size into the program. (C++ insists that it knows the actual size of arrays for function calls, for example.)

E.g., in file scope, assuming that the array you want to work with is a 10X14 you might have:

```
const int rows = 10, cols = 14; (It is OK if you use #define for this too.)
```

then in main:

```
char maze[rows][cols];
```

You will need to update all functions accordingly.

But, some of you may figure out a way to allow the user to dynamically set the size of the maze (within certain limits). The "certain limits" would be a large array that you build your maze within (i.e., the actual maze would be in the upper left-hand corner of the large array). This way you could take the size as input and not require compiling in order to change the maze size.

## 4 Putting it all Together

Once you solve the problems – put them together in a program that:

1. prompts the user for a maze size (within a particular bound of width and height – say 25 x 25).
2. randomly generates a maze to the specification and prints it.
3. traverses the maze printing out the progress made with each step.
4. prompts the user to either quit or start again with another maze.

Don't forget, for projects make sure that the work you turn in is your own.

**Late projects will be penalized 5% a day** for each 24 hours beyond the due date. No assignments will be accepted more than 7 days late.