University of Wollongong

School of Computer Science and Software Engineering, Faculty of Informatics

CSCI124/MCS9124 Applied Programming Spring 2014

Assignment 2 (6 marks)

Due 11:59pm Sunday September 14 (end of Week 7), 2014.

Background:

To become familiar with pointers and solving a problem.

Task:

This assignment involves writing (or at least modifying) a program to traverse mazes. A maze can be represented by an array of characters thus:

```
# # # # # # # # # # # # # # . # # #
# . . . . . . . . . . . . # . # . #
# # # # # . # # # # # # # # . # . #
# . . . . . # . . . . . . . . # . #
# . # # # # # . # # # # # # # . # .
# . # . . . . . # . . . # . . . # .
# . # . # # # # # . # . # . # . # .
# . # . . . . . . . # . # . # . # .
# . # # # # # # # . # # # . # . # .
# . . . # . . . # . . . . . # . # .
# . # . # . # . # # # # # . # . # .
# . # . . . # . # . # . . . # . # .
# # # # # # # . # . # . # . # # # .
# . . . . . # . # . . . # . . . # .
# . # # # . # . # . # # # . # . # .
# . . . # . . . . . # . . . # . # .
# # # # # # # # # # # # # # # E # #
```

The # represents a wall while the . represents a pathway. The entrance is assumed to be at the top (the gap in the wall) and the exit is at the bottom, marked with an E.

This is a very special sort of maze where there are no freestanding walls, so that a very traditional maze traversal algorithm can be used:

"Put your hand on the right wall and begin moving forward. As long as you keep your hand on the wall you will reach the exit."

This can be described as follows. Suppose you are currently headed in a certain direction and you are at a particular position in the maze. If there is no wall on your right, turn right and go forward one step. Otherwise, go forward if possible, else turn left.

A solution for this problem for mazes of the size above, namely 17 x 17 can be found in `main.cpp`. Although only capable of handling that size, the dimensions of the maze appear throughout the code as constants (once read in as part of the data file). Some functions have extra arguments not used as yet.

Step 1:
Two of the functions in `main.cpp`, namely `TurnLeft()` and `WheresRight(),` are only stubs. By looking at the comments before each function and other similar functions, write the two functions. Test the program using the sample maze in **maze1.dat**. Note that the exit is not listed as one of the steps on the path through the maze.

Place the complete code for this step in `step1.cpp`.

Step 2:
Modify the program so that the array used for Maze is just one-dimensional. This means that element `[i][j]` of the maze will be found at position `mazeWidth*i+j` in the array. So Position will now become just an `int`. Maintain the same report of position by calculating the row and column from the position – you can now use the extra argument.

Note that now the maze can be any dimension, provided `mazeWidth*mazeHeight`, the space needed to store the maze, is less than or equal to the size of the array. Make the size 400 for now. Test with data file `maze2.dat.`

Place the complete code for this step in `step2.cpp`.

Step 3:
We will now modify the program so that the path taken is recorded. Add an array of Positions to the main function to hold the path followed, plus a count of how many steps have been taken so far (the array should be dynamic i.e. not have a fixed size – it should grow and contract). Each time through the loop, where `ReportPosition()` is called, add the current position to the path. Be wary of memory leaks.

At the end of the loop report the entire path. The call to `ReportPosition()` in the main loop can now be removed.

Once you have done this, let's now remove the backtracking from the path recorded. Everytime an addition is to be made to the path, avoid putting a duplicate of the current position (we're turning around). Also if the position to be added is the next-to-last one, take the last one off the path instead (we're backtracking). This means the final path presented is the direct route.

Place the complete code for this step in `step3.cpp`.

Step 4:
We will now modify the program to handle arbitrary sized mazes.

a) Modify the declaration of the Maze type so that it is a pointer to a char.
b) Modify `LoadMaze()` so that the space for the maze is dynamically allocated. This requires the Maze formal argument to become `Maze&` as its value is changing.
c) Delete the memory at the end of the program. Test with `maze2.dat` and then try `maze3.dat`.

Place the complete code for this step in `step4.cpp`.

Step 5:
Modify the program so that, instead of reporting the position of steps through the maze as row and column, the content of the maze data at that position is printed. Try `maze2.dat` with this variation.

Place the complete code for this step in `step5.cpp`.

Step 6:
Modify the program so that each position of the maze is an arbitrary length character string instead of just one character. This will involve changing the Maze type, the dynamic allocation of the maze and the freeing of the memory, along with comparisons of maze contents. Try again with `maze2.dat` and then with `maze4.dat`.

Place the complete code for this step in `step6.cpp`.

Submission Procedure

To submit the assignment, enter the following command. Do not forget to submit ALL your source and header files.

```
submit –u USERNAME –c CSCI124 –a ass2 step1.cpp step2.cpp step3.cpp
step4.cpp step5.cpp step6.cpp
```

Please note assignments that are late, will attract a 1 mark penalty each day late. Assignments will receive a zero mark if they are submitted more then 3 days late.