Imperial College London

Department of Computing

# MSc C++ Programming – Assessed Exercise No. 1

| | |
|---|---|
| **Issued**: | Friday 14 October 2011 |
| **Due**: | **Friday 28 October 2011** |
| **Lab Sessions**: | Friday 21 October (am) |
| | Wednesday 26 October (am) |
| | Friday 28 October (am) |

## Problem Description

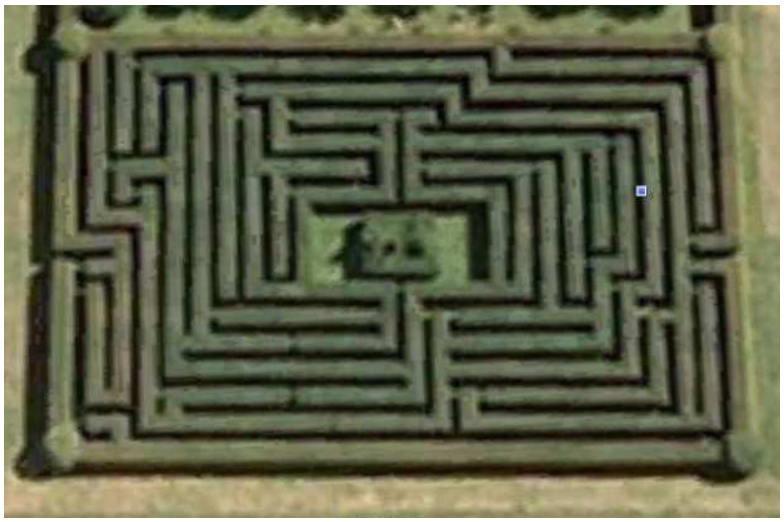
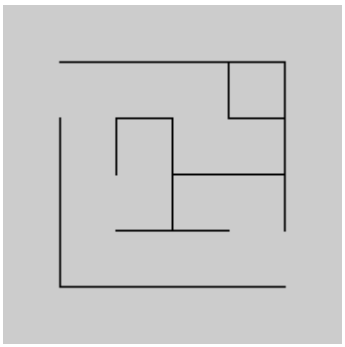


Figure 1: An aerial view of the hedge maze at Hatfield House (with thanks to Google Earth).

.

A hedge maze[1] is an outdoor maze or labyrinth in which the "walls" between passages are made of vertical hedges. Hedge mazes were a popular form of amusement for British and European royalty and aristocrats in the 17th and 18th centuries. Although less numerous (and less elitist) today, there are currently more than 100 mazes open to the public in Britain at venues such as Hampton Court Palace, Leeds Castle and Hatfield House (see Figure 1).
As shown in Figure 2, we will represent a maze as a rectangular two-dimensional array of characters, using the "barrier" characters '`-`', '`+`' and '`|`' for hedges and (unique) "marker" characters to denote points of interest; here '`>`' represents the entrance, '`X`' the exit, and '`U`' an unreachable square.
A solution path through the maze is a sequence of directions from one marker character to another using the letters N, S, E and W for north, south, east and west respectively. The path must not pass through any hedges or pass outside the boundaries of the maze. Hence a valid solution path for the example maze from the entrance to the exit (shown on the right in Figure 2) is "ESSSSSSEEEEEEE". Although it leads from the entrance to the exit, the path "EEEEEEEESSSSSS" is not a valid solution path since it passes through hedges.

[1] A desirable accessory if you are ever lucky enough to own a luxurious castle or a stately country manor.



```
  012345678          012345678
0 +-+-+-+-+        0 +-+-+-+-+
1 >     |U|        1 ##    |U|
2 + +-+ +-+        2 +#+-+ +-+
3 | | |   |        3 |#| |   |
4 + + +-+-+        4 +#+ +-+-+
5 |   |   |        5 |#  |   |
6 + +-+-+ +        6 +#+-+-+ +
7 |       X        7 |########
8 +-+-+-+-+        8 +-+-+-+-+
```

Figure 2: A simple hedge maze (left), its textual representation with marker characters (middle), and a valid solution path from the entrance to the exit (right).

## Pre-supplied functions and files

To get you started, you are initially supplied with some functions (with prototypes in **maze.h** and implementations in the file **maze.cpp**):

1. `char **allocate_2D_array(int rows, int columns)` is a helper function that dynamically allocates a two-dimensional (`rows` × `columns`) array of characters, returning the 2D array.

   For example, the code:

   ```
   char **m = allocate_2D_array(3,4);
   m[1][2] = '+';
   ```

   results in `m` being a 2D array of characters with 3 rows and 4 columns, and sets `m[1][2]` to '+'.

2. `deallocate_2D_array(char **m, int rows)` is a helper function that frees up the dynamically allocated 2D array `m`.

3. `char **load_maze(const char *filename, int &height, int &width)` is a function which reads in a maze from the file with name `filename`, sets the output parameters `height` and `width` according to the dimensions of the maze, and returns a 2D (`height` × `width`) array of characters representing the maze.

4. `void print_maze(char **m, int height, int width)` is a function which prints out the maze stored in the 2D (`height` × `width`) array of characters `m`. Row and column numbers are also shown.

You are also supplied with two text files containing mazes (**simple.txt** and **hatfield.txt**). You can view the contents of these files using the UNIX commands **cat simple.txt** and **cat hatfield.txt**.

## Specific Tasks

1. Write a function `find_marker(ch, maze, height, width, row, column)` which finds the coordinates of marker character `ch` in the 2D array of characters `maze` of dimension `height` × `width`. `row` and `column` are output parameters. When the maze contains the marker character, `row` and `column` should be set to the row and column coordinates of the marker character respectively, and the function should return `true`. If the maze does not contain the marker character `row` and `column` should both be set to -1, and the function should return `false`.

   For example, using the simple maze of Figure 2, the code:

   ```
   int height, width, row, column;
   char **maze = load_maze("simple.txt", height, width);
   bool success = find_marker('X', maze, height, width,
     row, column);
   ```

should result in `row` being 7, `column` being 8, and `success` being `true`.

2. Write a function `valid_solution(path, maze, height, width)` which determines if a given path through a `height` × `width` maze leads from the entrance marker '`>`' to the exit marker '`X`' without moving outside the boundaries of the maze or passing through a hedge. The parameter `path` is a string of uppercase characters, each of which is in the set `{'N','S','E','W'}` (corresponding to North, South, East and West respectively).

   For example, using the simple maze of Figure 2, the code:

   ```
   cout << "The move sequence 'ESSSSSSEEEEEEE' is ";
   if (!valid_solution("ESSSSSSEEEEEEE", maze, height, width))
     cout << "NOT ";
   cout << "a solution to the maze" << endl << endl;
   ```

   should result in the output:

   ```
   The move sequence 'ESSSSSSEEEEEEE' is a solution to the maze.
   ```

3. Write a function `find_path(maze, height, width, start, end)` which finds a valid solution path through a `height` × `width` maze beginning at marker character `start` and terminating at marker character `end`. The path should be marked on the `maze` using '`#`' characters, and the function should return a string of direction movements. If there is no path, the function should return the string `"no solution"`.

   For example, using the simple maze of Figure 2, the code:

   ```
   cout << "A path through the maze from '>' to 'X' is: ";
   cout << find_path(maze, height, width, '>', 'X') << endl;
   print_maze(maze, height, width);
   ```

   should result in the output:

   ```
   A path through the maze from '>' to 'X' is:
   ESSSSSSEEEEEEE

     012345678
   0 +-+-+-+-+
   1 ##    |U|
   2 +#+-+ +-+
   3 |#| |   |
   4 +#+ +-+-+
   5 |#  |   |
   6 +#+-+-+ +
   7 |########
   8 +-+-+-+-+
   ```

   **For full credit for this part, your function – or helper function if you choose to use one – should be recursive**.

Place your function implementations in the file **maze.cpp** and corresponding function declarations in the file **maze.h**. Use the file **main.cpp** to test your functions. Create a **makefile** which compiles your submission into an executable file called **maze**. Details of how to submit your files electronically will be emailed to you.

## How You Will Be Marked

You will be assigned a mark (for all your programming assignments) according to:

- whether your program works or not,
- whether your program is clearly set out with adequate blank space and indentation,
- whether your program is adequately commented,
- whether you have used meaningful names for variables and functions, and
- whether you have used a clear, appropriate and logical design.

## Optional Bonus Challenge (no credit)

Apply your `find_path` function to the Hatfield House maze (see **hatfield.txt**), finding a route from the entrance '`>`' to the middle '`M`', and then a route from the middle '`M`' to the exit '`X`'.
You can check your solution after the exam by watching the YouTube video (aptly entitled "Amazed") at:
`http://www.youtube.com/watch?v=nKswNWNwBSo`

## Hints

1. You will save a lot of time if you begin by carefully studying the supplied maze files **simple.txt** and **hatfield.txt**, the main program in **main.cpp** as well as the supplied functions in **maze.cpp**.
2. Feel free to define any auxiliary functions which would help to make your code more elegant.
3. One (but by no means the only) way to solve Question 3 is to use a helper function which implements a "recursive flood" algorithm based on your current position within the maze. First, for the base case consider (a) how will you know when you have solved the maze, and (b) how will you know when your current position is definitely not en route to a valid solution? Next, use recursive calls to see if going North, South, East or West from your current position leads to a valid solution. Concurrently, you will need to maintain the string of taken directions, and to mark the position of explored squares on the maze (remembering to unmark them when dead ends are encountered).
4. Try to attempt all questions. If you cannot get one of the questions to work, try the next one (all three may be tackled independently).