Finding a Path Through a Maze

A maze is a rectangular grid of cells, each of which is either a hall or a wall.

```
  0 1 2 3 4 5 6 7 8 9
+---------------------
0 | 0 1 0 0 1 0 1 1 1 1
1 | 0 1 1 0 1 0 1 0 0 0
2 | 0 0 0 0 0 0 1 0 1 0
3 | 1 1 1 1 1 0 1 1 0 0
4 | 0 0 0 1 0 0 0 0 0 1
5 | 0 1 0 1 0 1 1 1 0 1
6 | 0 1 0 1 0 0 0 1 0 0
7 | 1 0 0 1 1 1 0 1 1 0
8 | 0 1 0 0 1 0 0 1 1 0
9 | 0 1 0 1 1 0 1 0 0 0
```

The problem is to find a path from a given starting cell to a given end cell, or determine that no such path exists.
Decisions and Memory

Solving the problem requires walking through the maze, until we either reach the end cell or determine that no further progress is possible.

This requires several basic capabilities:

- know where we are right now
- decide where to go next
- remember where we’ve already been
- know when a place we’ve already been is no longer productive, and so know it’s time to backtrack
- know which of the places we’ve already been might still be productive
Representing the Maze

It’s natural to view the maze as a grid of cells.

A cell knows whether it’s a hall or a wall.

It’s also useful to be able to mark a cell as having been visited when we move through the maze.

```cpp
class Cell {
private:
    bool amIAWall;
    bool beenVisited;
    ...
};
```
Position and Direction

It’s natural to view our current location as a grid of physical locations represented by row and column numbers, corresponding to some coordinate system imposed on the maze.

It’s also natural to represent the direction in which we will consider moving from our current location:

From any position, there are up to eight possible directions in which we may move, corresponding to the eight major points of the compass.

We may model these directions with a class that supports a rotation-based increment operation.

We will systematically try all possible directions in some sensible pattern; the exact pattern doesn’t really matter (clockwise, counter-clockwise, etc.) nor does it really matter which direction we try first.
Our current location can be modeled by an aggregation that represents a position in a current direction in which we may attempt to go next.

```cpp
class Location {
private:
    int iRow;
    int iCol;
    Direction iDir;
public:
    ...'
    bool operator==(const Location& RHS) const;
    bool operator!=(const Location& RHS) const;
    Location Go() const;
    void Turn();
    ...'
};
```
Remembering the Path

The primary consideration is that if we reach a dead end, we must be able to back up to our previous location, and possibly further back along our path.

That suggests we represent our path by storing locations in a stack.

It’s also important to note that we must be sure that if we back up to a previous location we remember in which direction we should try to go next.
Finding a Solution

Consider the maze given earlier, and try to find a path from (0, 0) to (9, 9).

We’ll try going north first, and rotate clockwise when choosing our next direction.

Then try NE, then E, then SE… none are feasible.

But S is:

Again we start by looking N, and eventually rotate around to look SE, which is feasible:
Finding a Solution

Continuing…

Now, at this point we’ve reached a dead end at (0, 2): there are no unvisited neighboring cells to travel to.

So we must back up… which is easily done by popping the stack. We back up until we find a cell from which there is an untried direction:
Eventually we will reach (9, 9) and the stack will contain a record of the path, excluding the dead ends since those were popped off during the search:

```
0 1 2 3 4 5 6 7 8 9
+---------------------------------------------
0 | 0* 1 0* 0* 1 0* 1 1 1 1
1 | 0* 1 1 0* 1 0* 1 0* 0* 0*
2 | 0 0* 0* 0 0* 0* 1 0* 1 0*
3 | 1 1 1 1 1 0* 1 1 0* 0*
4 | 0 0 0 1 0 0 0* 0* 0* 1
5 | 0 1 0 1 0 1 1 1 0* 1
6 | 0 1 0 1 0 0 0 1 0 0*
7 | 1 0 0 1 1 1 0 1 1 0*
8 | 0 1 0 0 1 0 0 1 1 0*
9 | 0 1 0 1 1 0 1 0 0 0*  
```

```
9 9 N
• •
3 5 S
2 5 SW
1 5 SW
2 4 E
1 3 S
2 2 E
2 1 SE
1 0 S
0 0 SW
```
The Algorithm

Push starting Location onto stack.

While the stack isn’t empty and we aren’t at the end:

Pop stack to get current location.

While we haven’t tried every possible direction:

Let Step be the Location from Current in the next direction.

If Step is the end cell:
    Push Step onto the stack.
    Quit.

Else If Step hasn’t been visited and it’s not a wall:
    Mark Step as visited.
    Rotate the direction at Current (in case we back up).
    Push Current onto the stack (so we can back up).
    Set Current to Step.

Else
    Rotate the direction at Current.

EndWhile

EndWhile