Solving a 2D Maze

Let's use a 2D array to represent a maze. Let's start with a 10x10 array of char. The array of char can hold either 'X' for a wall, ' ' for a blank, and 'E' for the exit. Initially we can hard-code what our maze looks like, as such:

```
#include <iostream>
using namespace std;

const int WIDTH = 10;
const int HEIGHT = 10;

int main()
{
  char maze[HEIGHT][WIDTH] = {
        {'X','X','X','X','X','X','X','X','X','X'},
        {'X',' ',' ',' ',' ',' ','X',' ',' ','X'},
        {'X',' ','X',' ',' ',' ','X',' ',' ','X'},
        {'X',' ','X','X','X',' ','X',' ',' ','X'},
        {'X',' ',' ',' ','X',' ','X',' ',' ','X'},
        {'X',' ',' ',' ','X',' ',' ',' ','X','X'},
        {'X',' ','X','X','X',' ',' ',' ',' ','X'},
        {'X',' ','X',' ',' ',' ','X',' ',' ','X'},
        {'X',' ',' ',' ',' ',' ','X',' ','E','X'},
        {'X','X','X','X','X','X','X','X','X','X'}
  };
  int x=1,y=1;
}
```

Here we are using the format that initializes a 2D array when the array is defined. In this case let's make it so maze[0][0] is the upper left corner and maze[9][9] is the lower right corner. Because of the way 2D array contents are specified in C++, the first index refers to the row (y coordinate) and the second index corresponds to the column (x coordinate). This is a little different than normal! Usually we would use the first index for x and the second index for y. If we weren't hard-coding the array as shown above, we could use the more conventional format.

In addition, let's use integers x and y to store the location of the player. For example, if we start in the upper left corner then we would begin at x=1, y=1. We could have stored the player location as part of the maze, but keeping this separate will make things a little easier later on as we move around.

For debugging purposes a function we will want off the bat is a way to print the maze out:

```
void printMaze(char maze[][WIDTH], int curx, int cury)
{
  for (int y=0; y < HEIGHT;y++)
  {
    for (int x=0; x < WIDTH; x++)
    {
        if ((x==curx) && (y==cury))
                cout << "@";
        else
                cout << maze[y][x];
    }
    cout << endl;
  }
}
```

Calling this function will print out our maze. It might look a little funny if you have a proportional size font so the maze will not look square but rectangular.

Next, let's make it so we can use WASD to move up, down, left, or right. If we hit a wall then make it invalid to move there. It is machine-specific to get a keypress without pressing the enter key, so for now we'll just require the user press a key and then enter to move. If we hit the exit then stop.

Here is complete code:

```
#include <iostream>
using namespace std;

const int WIDTH = 10;
const int HEIGHT = 10;

// Function prototypes
void printMaze(char maze[][WIDTH], int curx, int cury);
bool validMove(char maze[][WIDTH], int newX, int newY);
bool move(char maze[][WIDTH], int &curX, int &curY,
          int newX, int newY);


// Return true or false if moving to the specified coordinate is valid
bool validMove(char maze[][WIDTH], int newX, int newY)
{
 // Check for going off the maze edges
 if (newX < 0 || newX >= WIDTH)
       return false;
 if (newY < 0 || newY >= HEIGHT)
       return false;
 // Check if target is a wall
 if (maze[newY][newX]=='X')
       return false;
 return true;
}
```

```
// Make the move on the maze to move to a new coordinate
// I passed curX and curY by reference so they are changed to
// the new coordinates.  I assume the move coordinates are valid.
// This returns true if we move onto the exit, false otherwise.
bool move(char maze[][WIDTH], int &curX, int &curY,
          int newX, int newY)
{
  bool foundExit = false;
  if (maze[newY][newX]=='E') // Check for exit
       foundExit = true;
  curX = newX;               // Update location
  curY = newY;

  return foundExit;
}

// Display the maze in ASCII
void printMaze(char maze[][WIDTH], int curx, int cury)
{
  for (int y=0; y < HEIGHT;y++)
  {
    for (int x=0; x < WIDTH; x++)
    {
       if ((x==curx) && (y==cury))
              cout << "@";
       else
              cout << maze[y][x];
    }
    cout << endl;
  }
}

int main()
{
 char maze[HEIGHT][WIDTH] = {
        {'X','X','X','X','X','X','X','X','X','X'},
        {'X',' ',' ',' ',' ',' ','X',' ',' ','X'},
        {'X',' ','X',' ',' ',' ','X',' ',' ','X'},
        {'X',' ','X','X','X',' ','X',' ',' ','X'},
        {'X',' ',' ',' ','X',' ','X',' ',' ','X'},
        {'X',' ',' ',' ','X',' ',' ',' ','X','X'},
        {'X',' ','X','X','X',' ',' ',' ',' ','X'},
        {'X',' ','X',' ',' ',' ','X',' ',' ','X'},
        {'X',' ',' ',' ',' ',' ','X',' ','E','X'},
        {'X','X','X','X','X','X','X','X','X','X'}
 };

 int x = 1, y = 1;
 bool foundExit = false;


 while (!foundExit)
```

```
 {
        printMaze(maze,x,y);
        cout << "Enter WASD to move." << endl;
        char c;
        cin >> c;
        c = tolower(c);
        switch (c)
        {
                case 'w':
                        if (validMove(maze,x,y-1))
                                foundExit = move(maze,x,y,x,y-1);
                        break;
                case 'a':
                        if (validMove(maze,x-1,y))
                                foundExit = move(maze,x,y,x-1,y);
                        break;
                case 's':
                        if (validMove(maze,x,y+1))
                                foundExit = move(maze,x,y,x,y+1);
                        break;
                case 'd':
                        if (validMove(maze,x+1,y))
                                foundExit = move(maze,x,y,x+1,y);
        }
 }
}
```

To have the computer solve the maze perhaps the easiest approach is to randomly pick a direction and try it. In theory, eventually we would find out way to the exit. Here are some modification we would make:

```
#include <cstdlib>
#include <ctime>

        …  in main …
        srand(time(NULL));
           …
        printMaze(maze,x,y);
        char c;
        int rnd = rand() % 4;
        if (rnd == 0)
                c = 'w';
        else if (rnd == 1)
                c = 'a';
        else if (rnd == 2)
                c = 's';
        else
                c = 'd';
        switch (c)
        …
```

If we try this then on our simple maze eventually we reach the end, sometimes it takes a while though!

A better solution is to actually search for the exit. We can do this in a fairly small amount of code using **recursion**. The stack used by the computer will help us remember which moves we have made. To remember the cells we have already been to we can make a new 2D array of booleans. Initially they are set to false to indicate we haven't visited them yet. When we visit that location we set it to true. In checking for a valid move, we make sure we haven't already visited that cell yet. If we have, we can consider it invalid.

To declare and initialize this array in main:

```
 bool visited[HEIGHT][WIDTH];

 // Initialize visited locations
 for (int x = 0; x < WIDTH; x++)
  for (int y = 0; y < HEIGHT; y++)
        visited[y][x] = false;
 visited[y][x] = true;
```

Here are updated versions of our functions:

```
bool validMove(char maze[][WIDTH], bool visited[][WIDTH],
               int newX, int newY)
{
 if (newX < 0 || newX >= WIDTH)
        return false;
 if (newY < 0 || newY >= HEIGHT)
        return false;
 if (maze[newY][newX]=='X')
        return false;
 if (visited[newY][newX])
        return false;
 return true;
}
```

```
bool move(char maze[][WIDTH], bool visited[][WIDTH],
          int &curX, int &curY, int newX, int newY)
{
  bool foundExit = false;
  if (maze[newY][newX]=='E')    // Check for exit
        foundExit = true;
  curX = newX;                  // Update location
  curY = newY;
  visited[curY][curX] = true;

  return foundExit;
}
```

We have to change our function calls in main for `validMove` and `move` to also pass in the `visited` array. We could run this with our random move algorithm but it may get stuck! Do you know why?

Now that we have a way to remember where we've been, here is pseudocode for the recursive algorithm to search for the exit. We start with x,y set to our start coordinates:

```
search(maze, visited, x, y)
        If current cell is the exit then return true
        Mark the current cell as being visited
        If up is valid and not visited
                search(maze,visited,curX, curY-1)
        if left is valid and not visited and we haven't found an exit yet
                search(maze,visited,curX-1, cury)
        if right is valid and not visited and we haven't found an exit yet
                search(maze, visited, curX+1,cury)
        if down is valid and not visited and we haven't found an exit yet
                search(maze, visited, curX, cury+1)
        if exit found, return true
        return false;  // couldn't find a solution from this cell in any direction
```

We can trace the algorithm logic from our start position. Let's say we start at 2,2 using the small maze shown below. This is what things look like when we are in main.

Maze

X	X	X	X
X			X
X			X
X	X		X
X	X	E	X

Visited

F	F	F	F
F	F	F	F
F	F	F	F
F	F	F	F
F	F	F	F

Stack

Main: x=2,y=2 → search(maze,visited,2,2)

When we call search for the first time we check that we are not at the exit and set visited to true:

Maze

X	X	X	X
X			X
X		@	X
X	X		X
X	X	E	X

Visited

F	F	F	F
F	F	F	F
F	F	T	F
F	F	F	F
F	F	F	F

Stack

search: x=2,y=2

Main: x=2,y=2 → search(maze,visited,2,2)

Inside search, we see that it is valid to move up to (2,1). It is not visited and is not a wall. So we make a recursive call to search with y-1.

Maze

X	X	X	X
X		@	X
X			X
X	X		X
X	X	E	X

Visited

F	F	F	F
F	F	T	F
F	F	T	F
F	F	F	F
F	F	F	F

Stack

search: x=2,y=1

search: x=2,y=2 → search(maze,visited,2,1) // up

Main: x=2,y=2 → search(maze,visited,2,2)

In this recursive call, moving up is not valid because there is a wall at (2,0). Next we would try moving left, which is allowed because there is no wall at (1,1) and it is not visited.

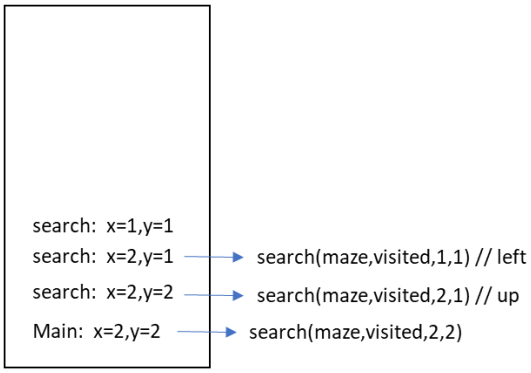
Maze

X	X	X	X
X	@		X
X			X
X	X		X
X	X	E	X

Visited

F	F	F	F
F	T	T	F
F	F	T	F
F	F	F	F
F	F	F	F

Stack



In this recursive call, we check up and that is not valid because of the wall at (1,0). We check left, and that is not valid because of the wall at (0,1). Next we check right and that is not allowed because we already visited (2,1). So finally we check down and that is allowed because (1,2) is a blank and not visited.

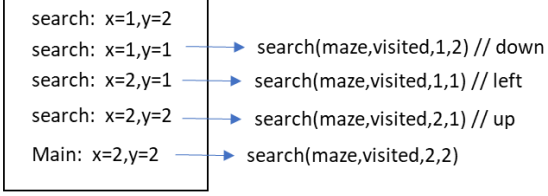
Maze

X	X	X	X
X			X
X	@		X
X	X		X
X	X	E	X

Visited

F	F	F	F
F	T	T	F
F	T	T	F
F	F	F	F
F	F	F	F

Stack



At this point our recursive calls have reached a dead end! Checking up is a cell we have visited already. Checking left is a wall. Checking right is a cell we have visited already. Checking down is a wall. This recursive call must exit and return false. We return to execution of search in the previous stack frame.

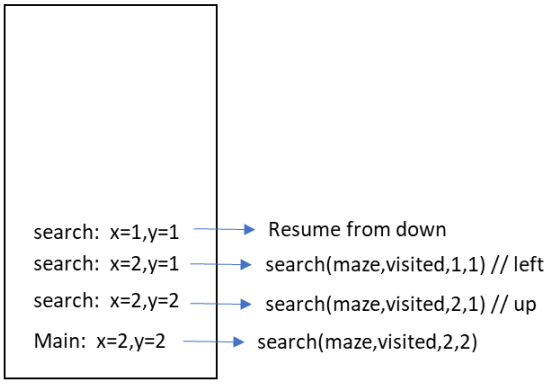
Maze

X	X	X	X
X	@		X
X			X
X	X		X
X	X	E	X

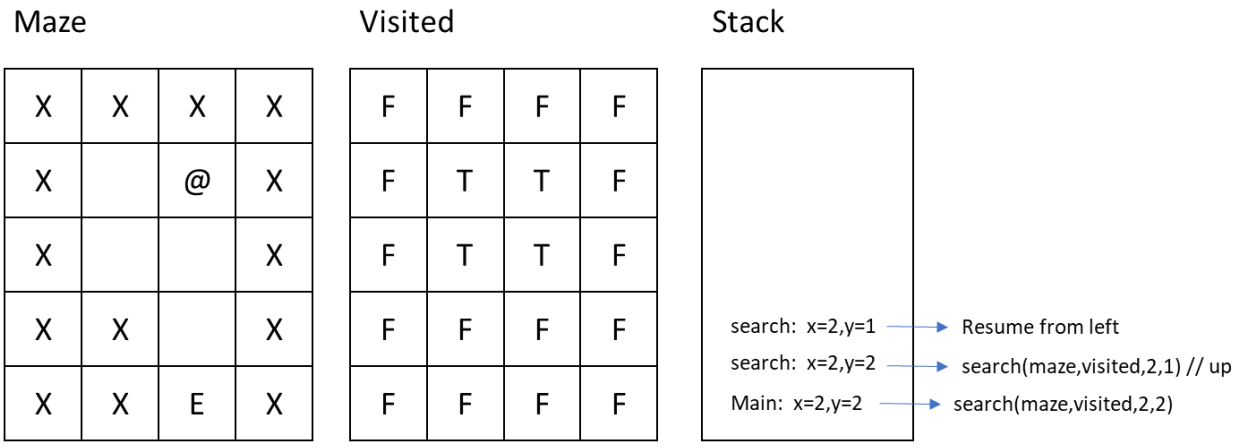
Visited

F	F	F	F
F	T	T	F
F	T	T	F
F	F	F	F
F	F	F	F

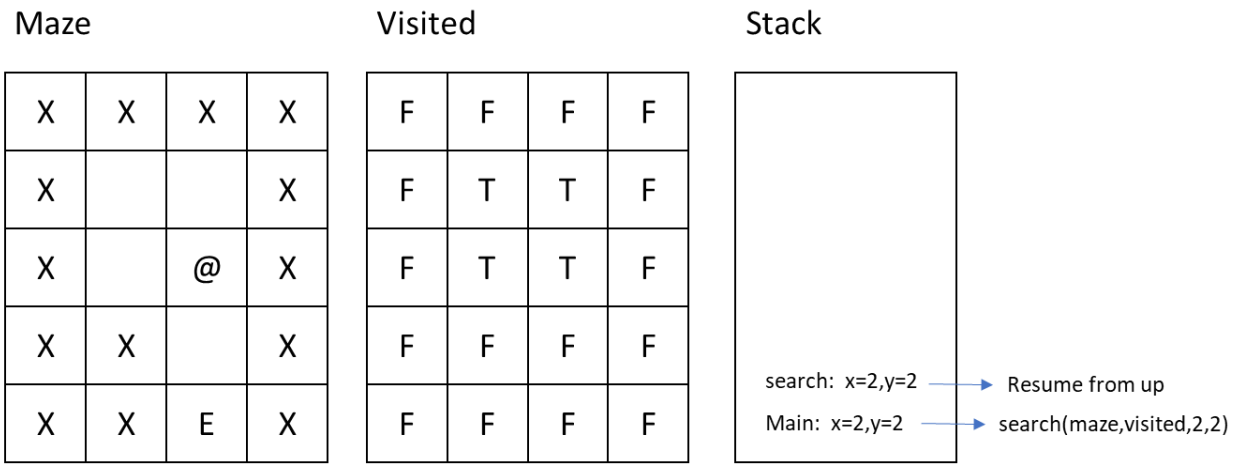
Stack



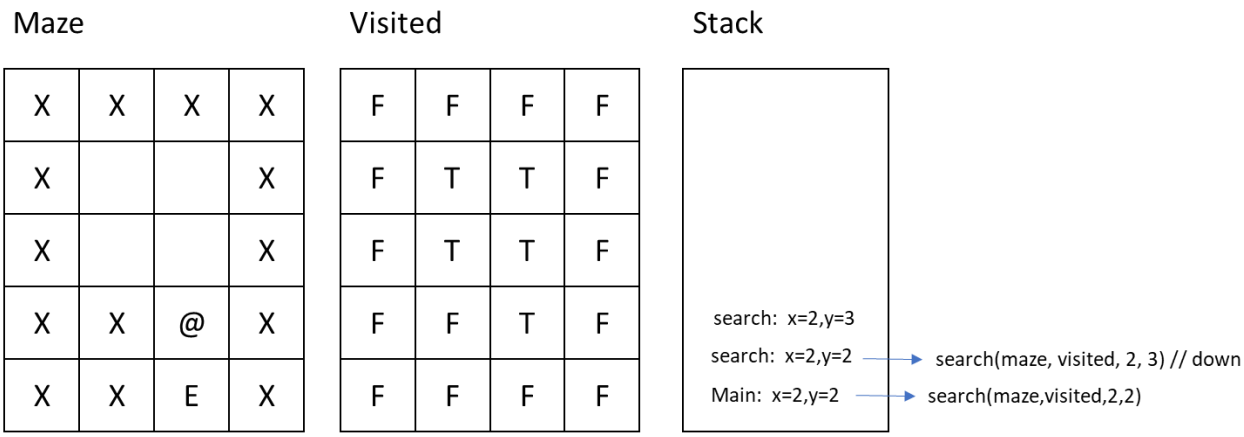
In this recursive call, we resume execution from searching down. It turns out there is nothing left to search, so this recursive call also returns false and we return to execution of search in the previous stack frame.



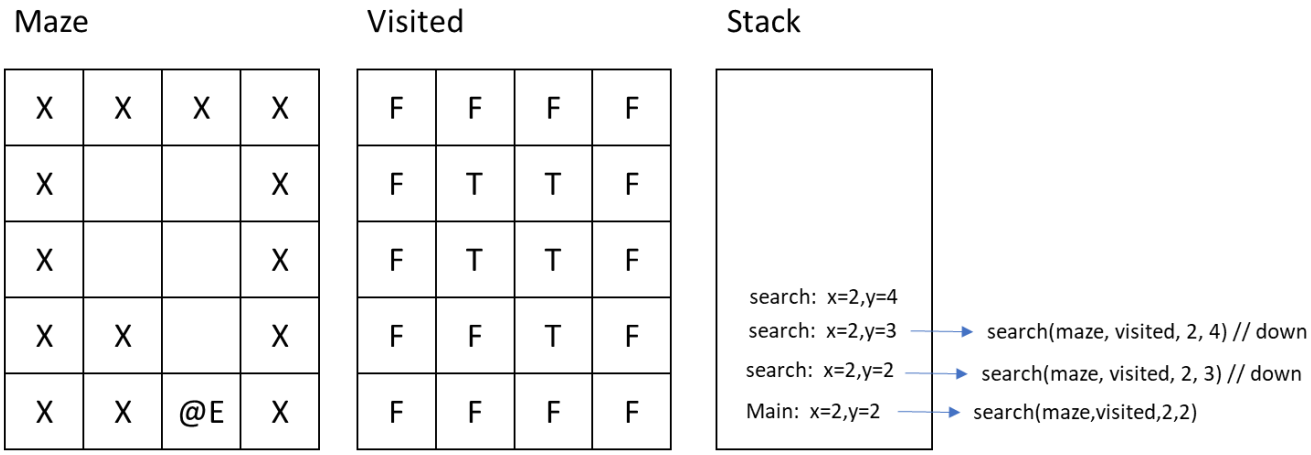
This recursive call would resume execution from looking left, so it will try looking right but we can't move there because of the wall at (3,1). We try looking down but can't move there because we already visited (2,2). So this recursive call also returns false and we resume execution in the previous stack frame.



This recursive call would resume execution from looking up, so next it will look left but we can't move there because we already visited (1,2). Next we look right and can't move there because of the wall at (3,2). Next we look down and we can move there! (2,3) is not visited and is empty.



From this recursive call we mark the cell as visited and check up at (2,2), left at (1,3), right at (3,3) and none of those are not allowed. But we can move down to (2,4). It is empty and not a wall.



At this point the first line of the recursive call kicks in! We are at the exit, so this returns true.

All of the other recursive calls return true until execution resumes in main.

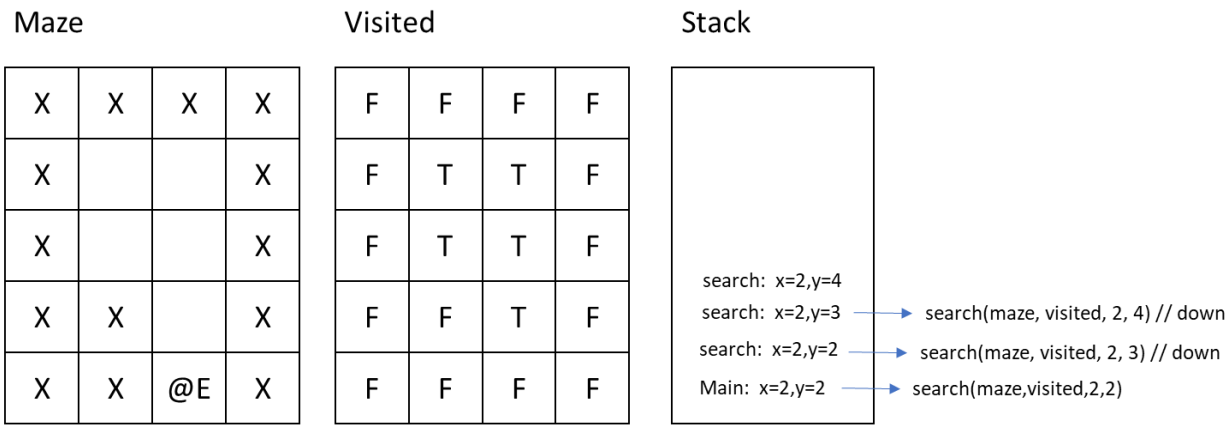
Here is complete code:

```
include <iostream>
#include <ctime>
#include <cstdlib>
using namespace std;

const int WIDTH = 10;
const int HEIGHT = 10;

// Function prototypes
void printMaze(char maze[][WIDTH], int curx, int cury);
bool validMove(char maze[][WIDTH], bool visited[][WIDTH],
               int newX, int newY);
bool move(char maze[][WIDTH], bool visited[][WIDTH],
          int &curX, int &curY, int newX, int newY);
```

```
// Return true or false if moving to the specified coordinate is valid
// Return false if we have been to this cell already
bool validMove(char maze[][WIDTH], bool visited[][WIDTH],
               int newX, int newY)
{
 // Check for going off the maze edges
 if (newX < 0 || newX >= WIDTH)
     return false;
 if (newY < 0 || newY >= HEIGHT)
     return false;
 // Check if target is a wall
 if (maze[newY][newX]=='X')
     return false;
 // Check if visited
 if (visited[newY][newX])
     return false;
 return true;
}

// Make the move on the maze to move to a new coordinate
// I passed curX and curY by reference so they are changed to
// the new coordinates.  Here we assume the move coordinates are
valid.
// This returns true if we move onto the exit, false otherwise.
// Also update the visited array.
bool move(char maze[][WIDTH], bool visited[][WIDTH],
       int &curX, int &curY, int newX, int newY)
{
  bool foundExit = false;
  if (maze[newY][newX]=='E')     // Check for exit
     foundExit = true;
  curX = newX;              // Update location
  curY = newY;
  visited[curY][curX] = true;

  return foundExit;
}

// Display the maze in ASCII
void printMaze(char maze[][WIDTH], int curx, int cury)
{
  for (int y=0; y < HEIGHT;y++)
  {
    for (int x=0; x < WIDTH; x++)
    {
     if ((x==curx) && (y==cury))
           cout << "@";
     else
           cout << maze[y][x];
    }
```

```
    cout << endl;
  }
}

// Recursively search from x,y until we find the exit
bool search(char maze[][WIDTH], bool visited[][WIDTH],
            int x, int y)
{
   bool foundExit;

   if (maze[y][x]=='E')
     return true;
   visited[y][x]=true;
   if (validMove(maze,visited,x,y-1))
     foundExit = search(maze,visited,x,y-1);
   if (!foundExit && (validMove(maze,visited,x,y+1)))
     foundExit = search(maze,visited,x,y+1);
   if (!foundExit && (validMove(maze,visited,x-1,y)))
     foundExit = search(maze,visited,x-1,y);
   if (!foundExit && (validMove(maze,visited,x+1,y)))
     foundExit = search(maze,visited,x+1,y);

   return foundExit;
}

int main()
{
 char maze[HEIGHT][WIDTH] = {
        {'X','X','X','X','X','X','X','X','X','X'},
        {'X',' ',' ',' ',' ',' ','X',' ',' ','X'},
        {'X',' ','X',' ',' ',' ','X',' ',' ','X'},
        {'X',' ','X','X','X',' ','X',' ',' ','X'},
        {'X',' ',' ',' ','X',' ','X',' ',' ','X'},
        {'X',' ',' ',' ','X',' ',' ',' ','X','X'},
        {'X',' ','X','X','X',' ',' ',' ',' ','X'},
        {'X',' ','X',' ',' ',' ','X',' ',' ','X'},
        {'X',' ',' ',' ',' ',' ','X',' ','E','X'},
        {'X','X','X','X','X','X','X','X','X','X'}
 };
 bool visited[HEIGHT][WIDTH];

 int x = 1, y = 1;
 bool foundExit = false;

 // Initialize visited locations to false
 for (int x = 0; x < WIDTH; x++)
  for (int y = 0; y < HEIGHT; y++)
        visited[y][x] = false;
 visited[y][x] = true;

 cout << search(maze,visited,x,y) << endl;
}
```

This program runs, but it doesn't produce very exciting output. It just prints out "1" showing that a path was found to the exit! To print the actual path to the exit, look at the call stack at the point where we found the exit:



The (x,y) coordinates stored on the call stack contain the locations we need to move to in order to reach the goal. If we simply output these coordinates (or the board with the @ at these coordinates) as the recursive search function exits a found exit, then it will output the sequence of mazes to reach the exit.

Here is the modified search function:

```
bool search(char maze[][WIDTH], bool visited[][WIDTH], int x, int y)
{
   bool foundExit;
   if (maze[y][x]=='E')
      return true;
   visited[y][x]=true;
   if (validMove(maze,visited,x,y-1))
      foundExit = search(maze,visited,x,y-1);
   if (!foundExit && (validMove(maze,visited,x,y+1)))
      foundExit = search(maze,visited,x,y+1);
   if (!foundExit && (validMove(maze,visited,x-1,y)))
      foundExit = search(maze,visited,x-1,y);
   if (!foundExit && (validMove(maze,visited,x+1,y)))
      foundExit = search(maze,visited,x+1,y);

   if (foundExit)
   {
      printMaze(maze,x,y);
      cout << endl;
      return true;
   }
   return false;
}
```

Here is sample output for our full maze. Note the path is output in reverse order from the goal first, to the start last, because of the way the values are on the stack.

```
XXXXXXXXXX
X        X
X XX  XX X
X  X  X  X
X  XXXX  X
X        X
XXXXX  XXX
X       @X
X    X  EX
XXXXXXXXXX
```

```
XXXXXXXXXX
X        X
X XX  XX X
X  X  X  X
X  XXXX  X
X        X
XXXXX  XXX
X      @ X
X    X  EX
XXXXXXXXXX
```

```
XXXXXXXXXX
X        X
X XX  XX X
X  X  X  X
X  XXXX  X
X        X
XXXXX  XXX
X        X
X    X @EX
XXXXXXXXXX
```

```
XXXXXXXXXX
X        X
X XX  XX X
X  X  X  X
X  XXXX  X
X        X
XXXXX  XXX
X        X
X    X@ EX
XXXXXXXXXX
```

```
XXXXXXXXXX
X        X
X XX  XX X
X  X  X  X
X  XXXX  X
X        X
XXXXX  XXX
X     @  X
X    X  EX
XXXXXXXXXX
```

```
XXXXXXXXXX
X        X
X XX  XX X
X  X  X  X
X  XXXX  X
X        X
XXXXX  XXX
X    @   X
X    X  EX
XXXXXXXXXX
```

```
XXXXXXXXXX
X        X
X XX  XX X
X  X  X  X
X  XXXX  X
X        X
XXXXX@ XXX
X        X
X    X  EX
XXXXXXXXXX
```

```
XXXXXXXXXX
X        X
X XX  XX X
X  X  X  X
X  XXXX  X
X    @   X
XXXXX  XXX
X        X
X    X  EX
XXXXXXXXXX
```

```
XXXXXXXXXX
X        X
X XX  XX X
X  X  X  X
X  XXXX  X
X   @    X
XXXXX  XXX
X        X
X    X  EX
XXXXXXXXXX

XXXXXXXXXX
X        X
X XX  XX X
X  X  X  X
X  XXXX  X
X  @     X
XXXXX  XXX
X        X
X    X  EX
XXXXXXXXXX

XXXXXXXXXX
X        X
X XX  XX X
X  X  X  X
X  XXXX  X
X @      X
XXXXX  XXX
X        X
X    X  EX
XXXXXXXXXX

XXXXXXXXXX
X        X
X XX  XX X
X  X  X  X
X  XXXX  X
X@       X
XXXXX  XXX
X        X
X    X  EX
XXXXXXXXXX
```

```
XXXXXXXXXX
X        X
X XX  XX X
X  X  X  X
X@ XXXX  X
X        X
XXXXX  XXX
X        X
X    X  EX
XXXXXXXXXX

XXXXXXXXXX
X        X
X XX  XX X
X@ X  X  X
X  XXXX  X
X        X
XXXXX  XXX
X        X
X    X  EX
XXXXXXXXXX

XXXXXXXXXX
X        X
X@XX  XX X
X  X  X  X
X  XXXX  X
X        X
XXXXX  XXX
X        X
X    X  EX
XXXXXXXXXX

XXXXXXXXXX
X@       X
X XX  XX X
X  X  X  X
X  XXXX  X
X        X
XXXXX  XXX
X        X
X    X  EX
XXXXXXXXXX
```

1