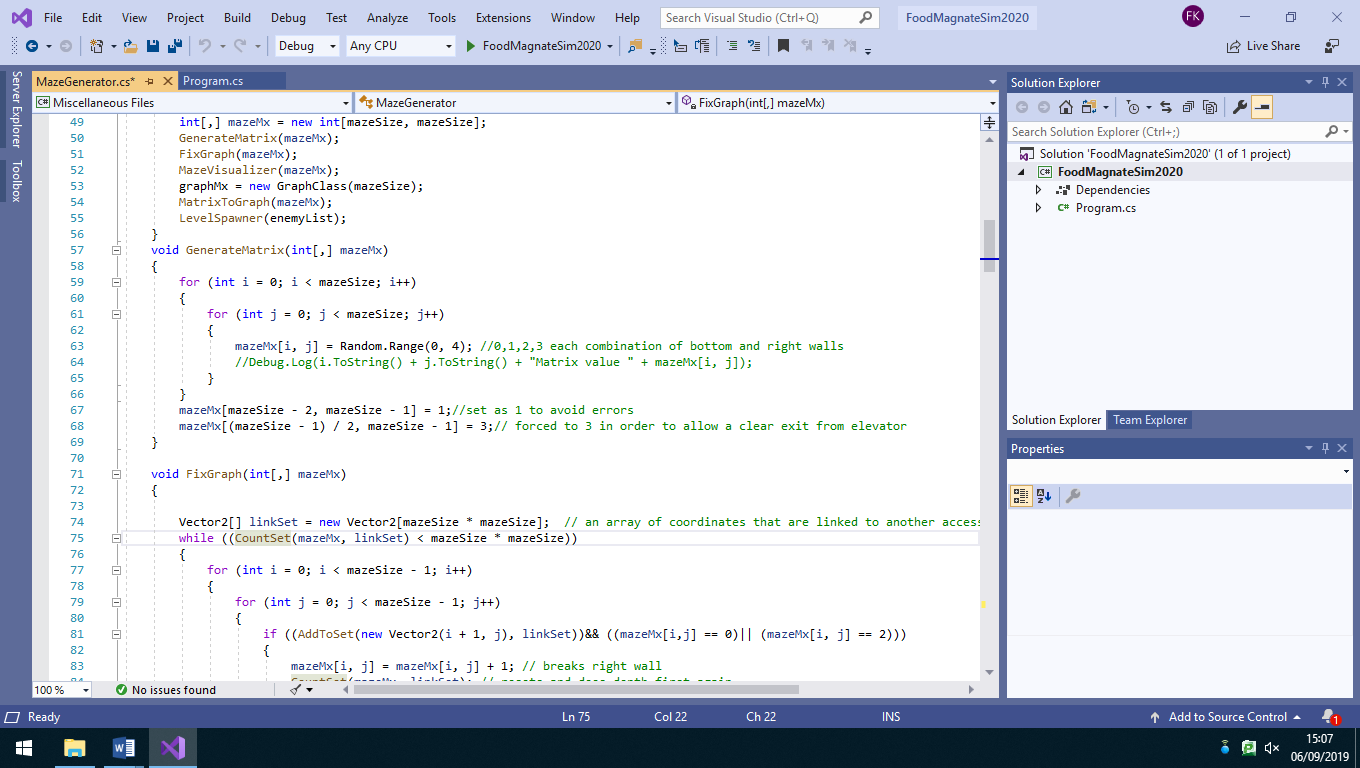
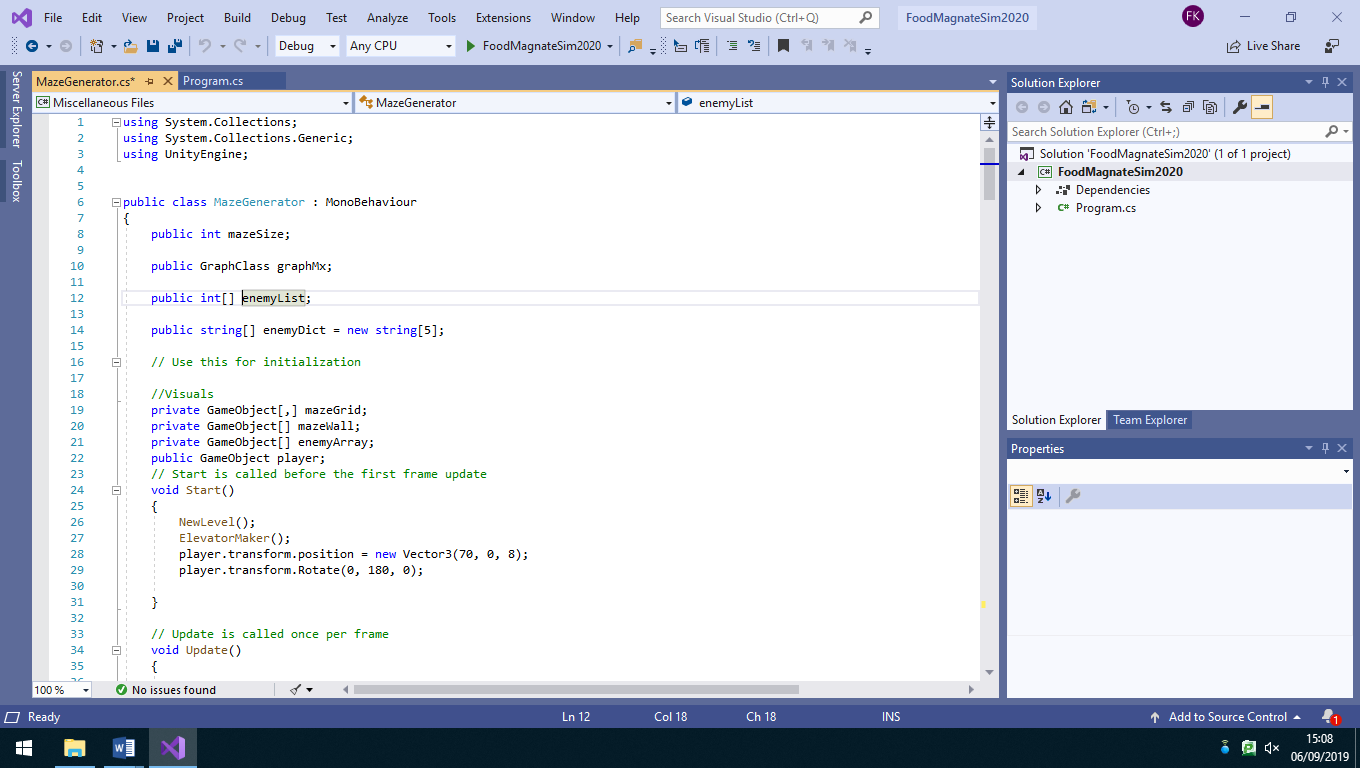
Explaining Maze Generator



This fills every single square in the matrix with either a: 0,1,2,3. The number put in the square is completely randomly generated from the choice of 4 numbers. Each number is designated to a specific type of wall

0

1

2

3

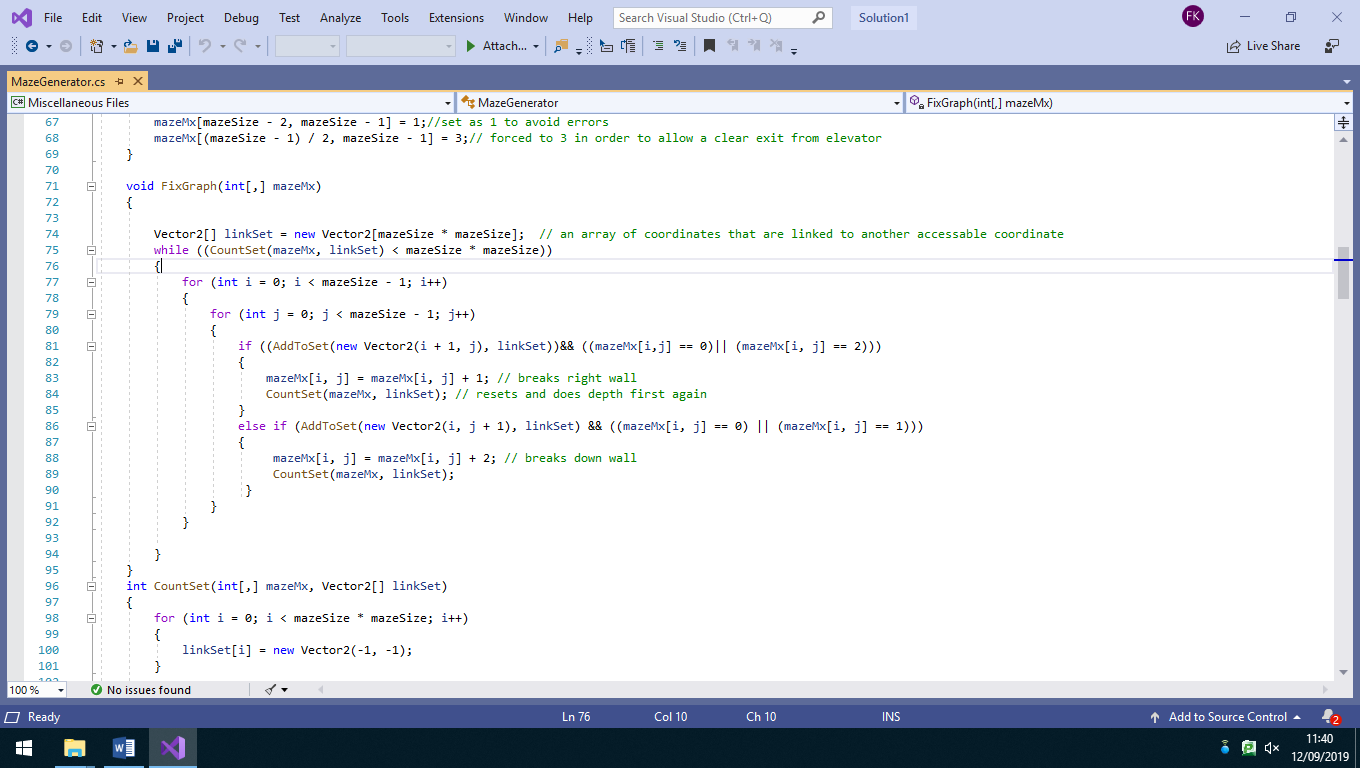
No wall

wall

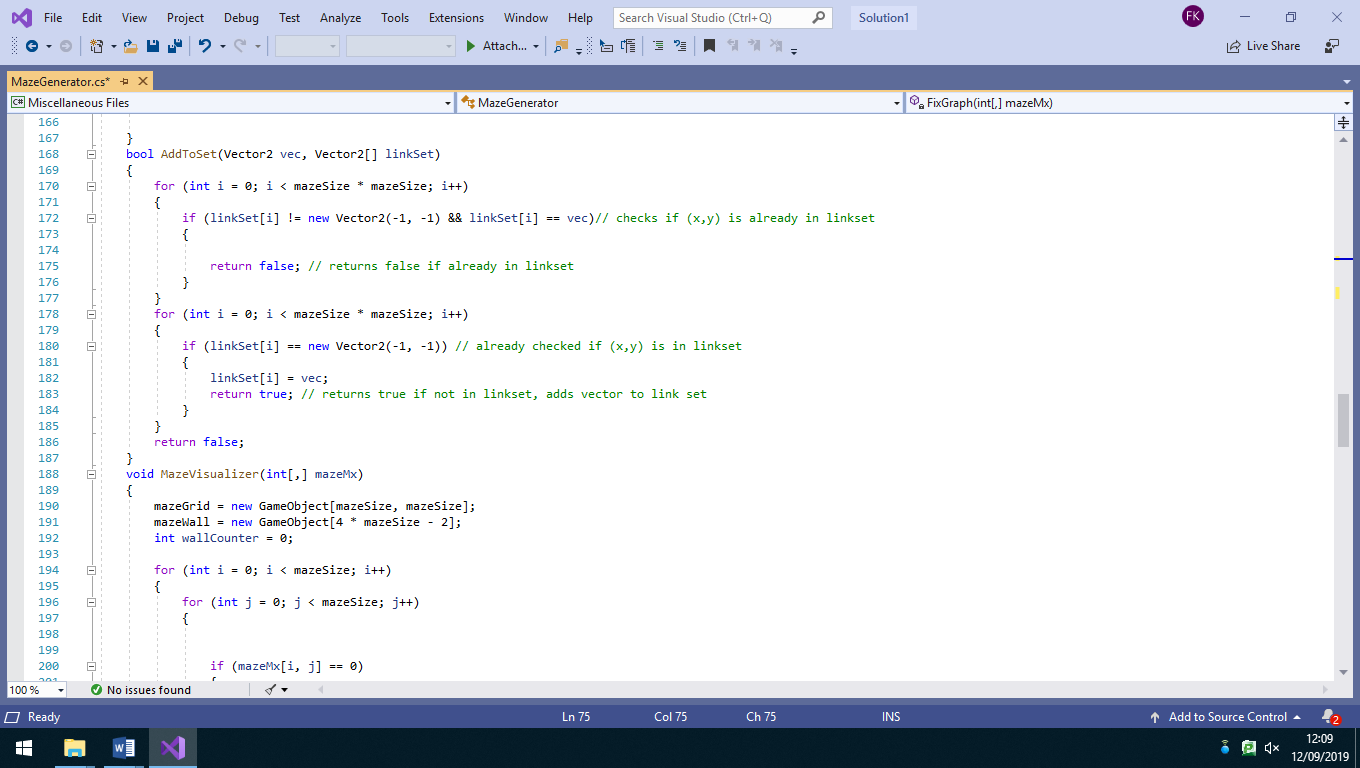
I had to force some squares to have a certain type of wall, in order to allow the player to exit spawn.

The way did this, is by creating an array, which stores all the coordinates of the squares that linked to an accessible square. I also made a depth first traversal method which starting from the top left of the matrix see’s how many squares it can go to. If the Number of squares accessible from by the depth first traversal method is equal to the area of the matrix (mazeSize \* mazeSize), this would mean that the issue is fixed and every square is accessible someway.

Since every square will have a random type of wall, not every square will be accessible. In order to fix this issue, I created several different methods. The 4 methods I created to fix this issue are: FixGraph, CountSet, DepthFirst and AddToSet.



First of all, in the method FixGraph I created an empty vector 2 array called linkset which will store all the coordinates of squares that are linked to another accessible square



The Boolean method AddToSet checks whether the coordinate of a square is already in the linkset array or if they need to be added. If the coordinates are not in the array AddToSet returns true indicating it needs to be added to the set. If they are in the array it returns false.

Before AddToSet is run, in the method CountSet all the elements in the vector array are set to (-1, -1), since (0, 0) is a used vector in this program

After already checking whether the coordinates are in the array, the program then checks whether the coordinates at index of the array are (-1, -1). If they are, this means that this part of the array has been untouched and the coordinates of the square can replace the (-1, -1)

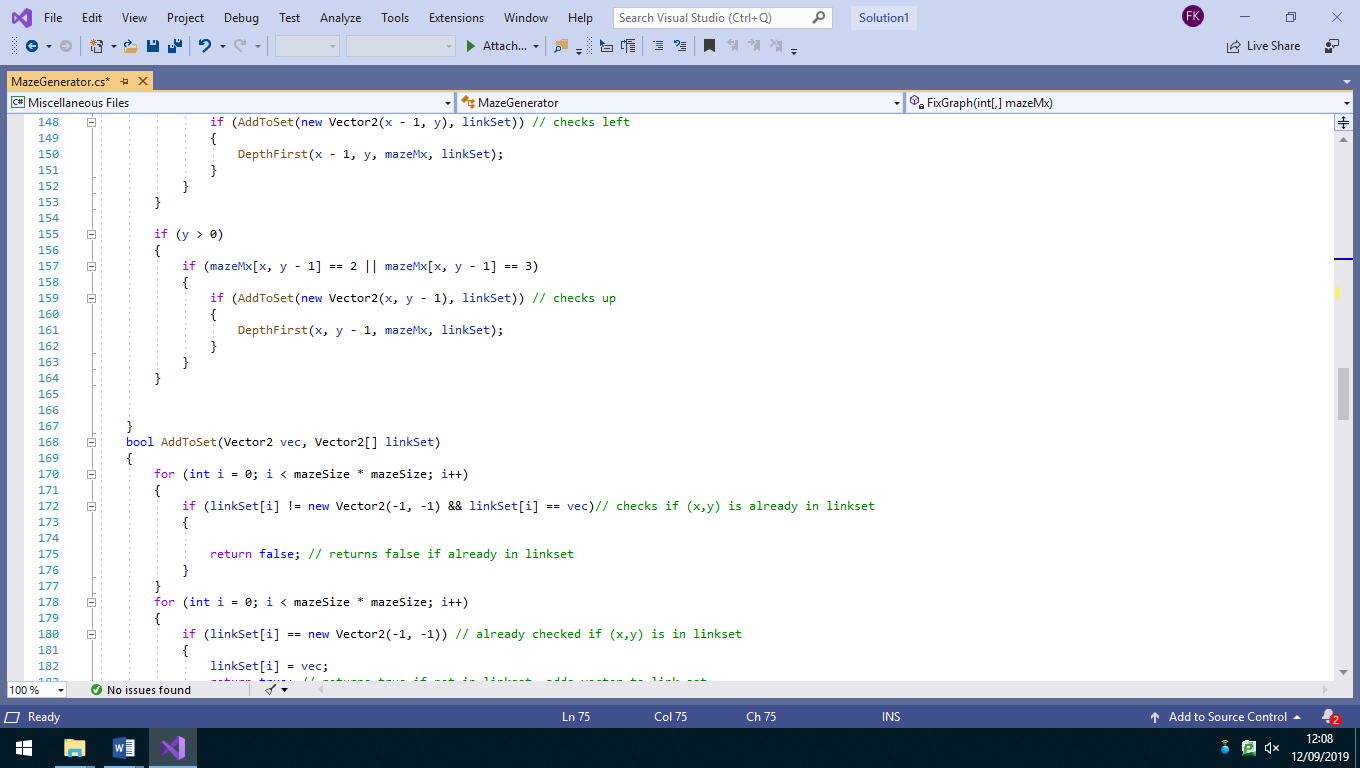
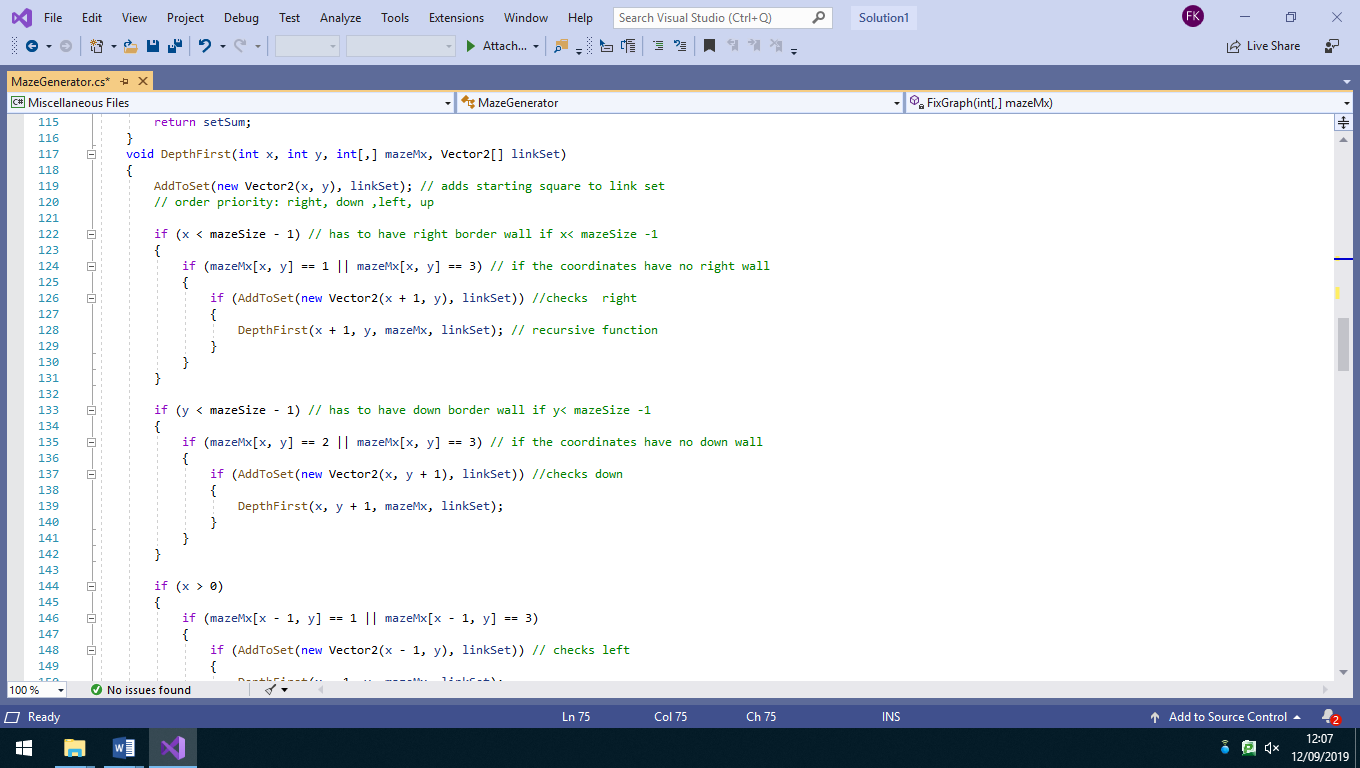
After going through all the rows, while x<mazeSize the program then repeats all this for the columns while y<mazeSize. . After one y value has been traversed it then calls the recursive Depth First method with the same parameters except the y value is incremented by 1, causing the square to move down by 1 square each time.

I created a recursive function to implement depth first traversal on the maze. I called this method DepthFirst. Depth first checks the all the squares accessible from the origin (0,0).

y = mazeSize -1

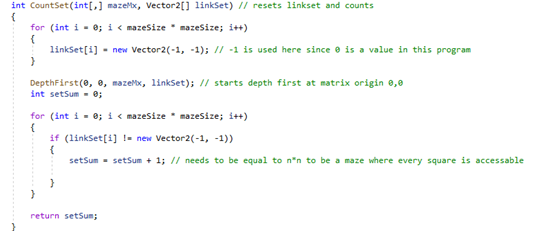
It starts at the origin (0,0) and first checks the square right of that. If that square is accessible (has no right wall in between) it runs the AddToSet method in order to add that coordinate to linkSet. Then it calls the recursive DepthFirst method with the same parameters except the x value is incremented by 1, causing the square to move right by 1 square each time.

x = mazeSize -1



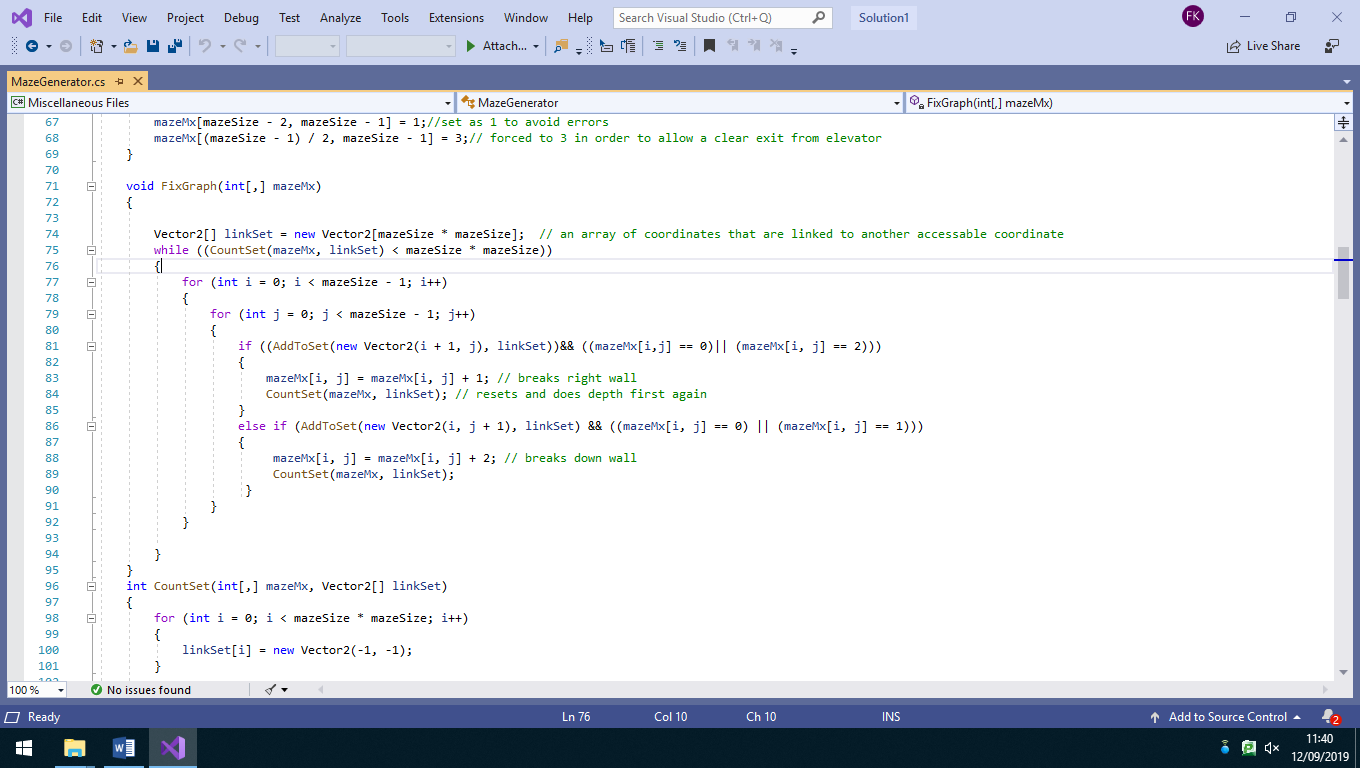
Then, for all the squares where y>0, it checks whether the square above has a down wall. If there is no down wall the square above can be accessible and the coordinates are added to linkSet. DepthFirst is called again with the same parameter except y is decreased by 1, causing the square to be 1 to the up each time

After, for all squares where x>0, it checks whether the square to the left has a right wall. If there is no wall, the square to the left can be accessible and the coordinates are added to linkSet. DepthFirst is called again with the same parameter except x is decreased by 1, causing the square to be 1 to the left each time

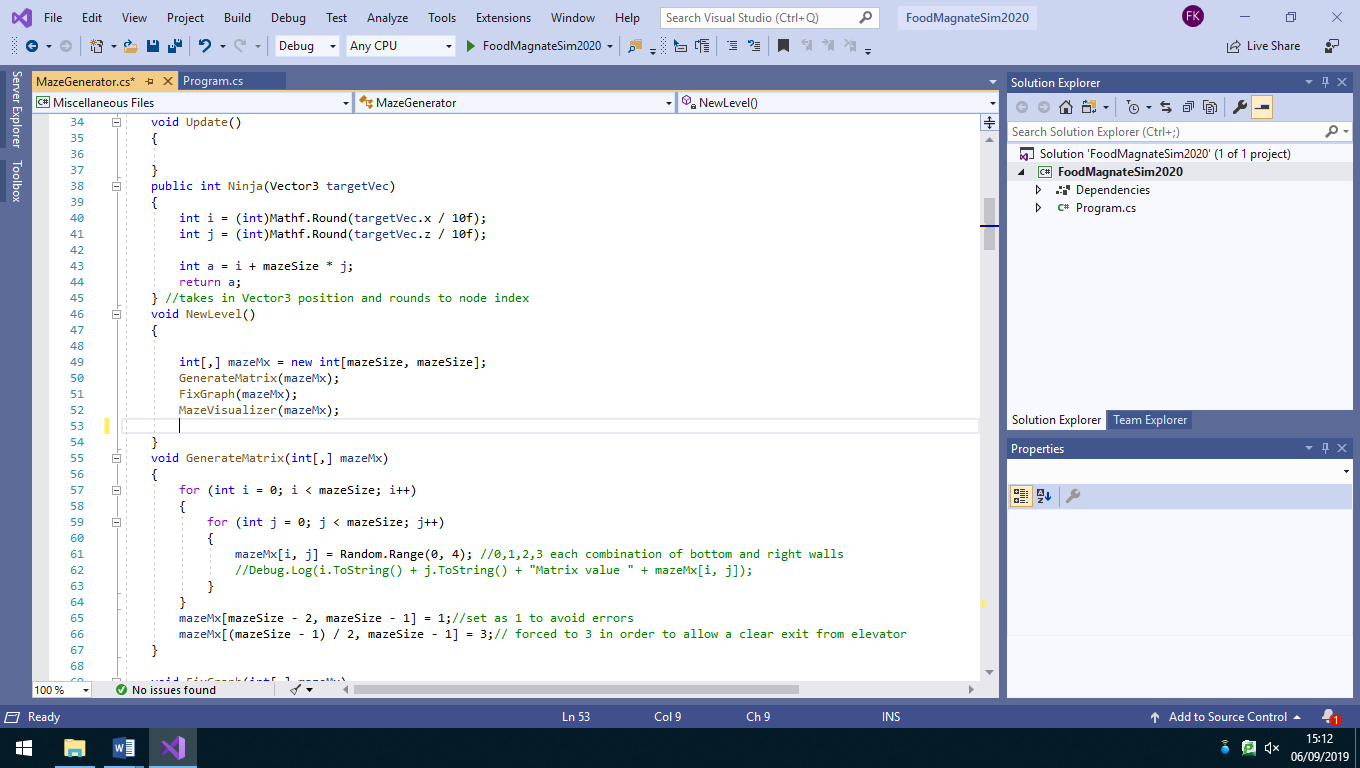


It then goes through linkSet again and if the coordinates at the index are not (-1, -1) they are accessible from the origin. For every accessible square it adds 1 to the integer variable setSum which counts the number of accessible squares. For every square to be accessible, setSum needs to be equal to the area of the matrix which would be mazeSize\*mazeSize.

The method CountSet is an integer method which returns the integer variable setSum. CountSet resets linkSet so that every element in the vector array is set to (-1, -1). It then runs DepthFirst starting with the x and y coordinates set to the origin (0,0).



FixGraph is the method which starts the fix to make every square accessible. As mentioned before it starts by making an empty vector 2 array called linkSet. Then it goes through the entire matrix, where first it runs the AddToSet method to check whether the coordinate is in linkSet, if not, depending on the number it breaks a wall. corresponding to the square (if 0 or 2 it breaks right wall) (0 or 1 breaks down wall) and the runs CountSet to reset linkSet and refill it with every square accessible from origin.



Creates a matrix (table/graph) for the base for the maze. The dimensions are mazeSize by mazeSize which is a variable which value is assigned before program runs

MazeVisualizer is a method which actually creates the playable maze. As mentioned before that every square is assigned to a specific type of wall (either 0,1,2 or 3), MazeVisualizer loads the actual prefab for the specific wall from the Prefabs folder from the games file in unity.

MazeVisualizer also creates all the border walls for the matrix.

