Write a program in C++ which generates a maze which has `MROW` rows and `MCOL` columns where `MROW` and `MCOL` are constants specified in the program. Your maze will be generated using text graphics with the `|` character as a wall and `+-` as a floor. A typical maze that has 3 rows and 4 columns will look like this:

```
+--+--+--+--+
|  |  |  |  |
+--+--+--+--+
|  |  |  |  |
+--+--+--+--+
|  |  |  |  |
```

A perfect maze is one in which any two cells are connected by a single unique path and all cells are reachable by some path. Your program should generate a perfect maze.

A maze can be thought of as a two-dimensional array of cells in which each cell has a right wall and a bottom wall. For our purposes, the maze will be made up of a two-dimensional array of elements of the class `Cell` defined on the following page. Note that when printing a maze the first column of cells will have no left wall (since there is no adjoining column to the left and cells have only right walls). Likewise, there will be no top wall on the first row of cells. Your print routine will have to make up for this by printing these items separately.

**How to generate the maze**

Begin by initializing your maze to have all walls in place. Do this by setting all of the `rWall` and `bWall` variables in the class to true. Choose a random column in the first row as a starting point. Write a routine to tell you which of the adjoining cells have not yet been visited. (Note that the Cell class has a `visited` variable which can be set when you visit a cell.) Choose one of the adjoining available adjoining cells and knock out the wall between your cell and the new cell by setting the wall variable to false and marking the new cell as visited. Proceed in this manner until you reach a point where there are no more available cells to choose from. At this point you are at a location where all of the neighboring locations have been visited. Begin backing up along your route until you find a cell which does have available neighbors that have not been visited. When you can no longer find any cells with available neighbors you are done.

The Cell class has two variables called `fromR` and `fromC` which hold the row and column of the previous cell. For example, if I begin at row 0, col 3 and I proceed to row 1, col 3 then I will use `maze[1][3].SetFrom(0, 3)` to set `fromR` and `fromC` in the cell at row 0, col 3 the row and column of the previous cell. This information will allow me to back up along the path when the algorithm gets stuck.
Your program must be modular. Typically you will have *at least* one function which will print the maze, and another which will tell you what cells are available from a given cell.

Turn in a complete copy of your commented source code along with a screen printout that shows that your program works correctly.
const int MROW = 10;
const int MCOL = 20;

class Cell
{
public:
    bool WasVisited();
    void SetVisited(bool b);
    bool HasRWall();
    void SetRWall(bool b);
    bool HasBWall();
    void SetBWall(bool b);
    void SetFrom(int fRow, int fCol);
    void GetFrom(int &fRow, int &fCol);
private:
    bool visited;
    bool rWall; // right wall
    bool bWall; // bottom wall
    // fromR and fromC are the coordinates of the previous cell
    int fromR; // from row
    int fromC; // from col
};

// bool Cell::WasVisited()
// { return visited; }
void Cell::SetVisited(bool b)
{ visited = b; }

bool Cell::HasRWall()
{ return rWall; }
void Cell::SetRWall(bool b)
{ rWall = b; }

bool Cell::HasBWall()
{ return bWall; }
void Cell::SetBWall(bool b)
{ bWall = b; }

void Cell::SetFrom(int fRow, int fCol)
{ fromR = fRow;
  fromC = fCol;
}

void Cell::GetFrom(int &fRow, int &fCol)
{ fRow = fromR;
  fCol = fromC;
}