3. A treasure map is represented as a rectangular grid. Each grid location contains either a single treasure or nothing. The grid is represented using a matrix of Boolean values. If a cell in the grid contains a treasure then the value `true` is stored in the corresponding matrix location; otherwise, the value `false` is stored.

Consider the following declaration for the `TreasureMap` class.

```
class TreasureMap
{
  public:

    // ... constructors not shown

    bool HasTreasure(int row, int col) const;
    // postcondition: returns true if the cell at location (row, col)
    //                contains a treasure;
    //                returns false if location (row, col) is not within
    //                the bounds of the grid or if there is no treasure
    //                at that location

    int NumAdjacent(int row, int col) const;
    // precondition:  0 <= row < NumRows(); 0 <= col < NumCols()
    // postcondition: returns a count of the number of treasures in the
    //                cells adjacent to the location (row, col),
    //                horizontally, vertically, and diagonally

    int NumRows() const;
    // postcondition: returns the number of rows in the treasure map

    int NumCols() const;
    // postcondition: returns the number of columns in the treasure map

  private:

    apmatrix<bool> myGrid;
       // myGrid[r][c] being true indicates a treasure at (r, c)
       // the matrix is sized by the constructor
};
```

For example, suppose that the 6-by-9 grid shown below is a treasure map where the symbol 💰 in a cell indicates a treasure. In this example, `myGrid[2][3]` is `true` and `myGrid[1][2]` is `false`.

	0	1	2	3	4	5	6	7	8
0		💰	💰		💰		💰		
1		💰					💰		
2		💰		💰	💰			💰	💰
3	💰		💰		💰	💰			
4		💰			💰			💰	
5	💰			💰		💰			

(a) Write the `TreasureMap` member function `HasTreasure`, which is described as follows. `HasTreasure` returns `true` if there is a treasure at the location `(row, col)`. If `(row, col)` is not within the bounds of the grid or if there is no treasure at that location, `HasTreasure` returns `false`.

For example, if `TreasureMap theMap` represents the treasure map shown at the beginning of the question, the following table gives the results of several calls to `HasTreasure`.

Function call	Value returned
`theMap.HasTreasure(0, 2)`	`true`
`theMap.HasTreasure(0, -1)`	`false`
`theMap.HasTreasure(2, 3)`	`true`
`theMap.HasTreasure(2, 2)`	`false`
`theMap.HasTreasure(4, 9)`	`false`

Complete function `HasTreasure` below.

```
bool TreasureMap::HasTreasure(int row, int col) const
// postcondition: returns true if the cell at location (row, col)
//                contains a treasure;
//                returns false if location (row, col) is not within
//                the bounds of the grid or if there is no treasure
//                at that location
```

GO ON TO THE NEXT PAGE.

(b) Write the `TreasureMap` member function `NumAdjacent`, which is described as follows. `NumAdjacent` returns the number of treasures that are adjacent to a given location specified by `row` and `col`. To be adjacent, a treasure must be in one of the (at most) eight cells that border the given location horizontally, vertically, or diagonally; a treasure in the given location does not count as being adjacent.

The treasure map below is repeated for your convenience.

	0	1	2	3	4	5	6	7	8
0		💰	💰		💰		💰		
1		💰					💰		
2		💰		💰	💰			💰	💰
3	💰		💰		💰	💰			
4		💰			💰			💰	
5	💰			💰		💰			

For example, if `TreasureMap theMap` represents the treasure map shown above, the following table gives the results of several calls to `NumAdjacent`.

Function call	Value returned
`theMap.NumAdjacent(3, 3)`	5
`theMap.NumAdjacent(2, 4)`	3
`theMap.NumAdjacent(4, 7)`	0

In writing `NumAdjacent`, you may call `HasTreasure` specified in part (a). Assume that `HasTreasure` works as specified, regardless of what you wrote in part (a).

Complete function `NumAdjacent` below.

```
int TreasureMap::NumAdjacent(int row, int col) const
// precondition:  0 <= row < NumRows(); 0 <= col < NumCols()
// postcondition: returns a count of the number of treasures in the
//                cells adjacent to the location (row, col),
//                horizontally, vertically, and diagonally
```

GO ON TO THE NEXT PAGE.

(c) Write free function `ComputeCounts`, which is described as follows. `ComputeCounts` returns a matrix of integers where the value at `(row, col)` is `9` if there is a treasure at location `(row, col)` in `theMap`. Otherwise, the value at `(row, col)` is the number of treasures adjacent to location `(row, col)`.

For example, the following shows the matrix that is returned as a result of calling `ComputeCounts` with the `TreasureMap aMap`.

<u>`aMap`</u>

	0	1	2	3	4
0		💰		💰	💰
1	💰				
2		💰	💰		

<u>Matrix returned by the call `ComputeCounts(aMap)`</u>

	0	1	2	3	4
0	2	9	2	9	9
1	9	4	4	3	2
2	2	9	9	1	0

In writing `ComputeCounts`, you may call any `TreasureMap` member function. Assume that all member functions of `TreasureMap` work as specified, regardless of what you wrote in parts (a) and (b).

Complete function `ComputeCounts` below.

```
apmatrix<int> ComputeCounts(const TreasureMap & theMap)
```