

## Homework: Class `mat<T>`

You should implement a template parameterized class `mat` (in case someone is not familiar with templates you can implement with only `int` type, e.g. without `template<typename T>`) which is going to be a 2d-array, allocated with operator `new`.

The `mat` class should have the following structure:

```
template <typename T>
class mat
{
    size_t M;
    size_t N;

    T** v;
public:
};
```

You have to implement the following methods:

- `mat(const vector<vector<T>>&)` - a constructor which uses a `const vector<vector<T>>&` parameter to initialize the `mat` object.  
Example:  
`vector<vector<int>> v1 = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} };`  
`mat<int> m(v1);`  
The above line should be compiled and run correctly.
- `mat(size_t m, size_t n)` - a constructor which should initialize the `m`x`n` `mat` object with default values of type `T`.  
Example:  
`mat<int> m(10, 10); // each element of m[i][j] should have the default value zero.`
- `mat(const mat<T>& m2)` - copy constructor. You need to deep copy each value of `m2` into this object.  
Example:  
`vector<vector<int>> v1 = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} };`  
`mat<int> m(v1);`  
`mat<int> m2(m);`  
`//m and m2 should have exactly the same values`
- `~mat()` - Destructor to correctly release (delete) all the allocated memory.
- Two subscript operators `operator[]` and `const operator[]`
- `pair<size_t, size_t> size() const` - to return `M` and `N`.
- `vector<vector<T>> to_vec_of_vec() const` - convert `mat` to `vector<vector<T>>`.
- `friend ostream& operator<<(ostream& os, const mat<T>& m)` - to be able to cout `mat` objects

You also have to solve and implement the following problems in the most efficient way.

- **PROBLEM #1**

```
template <typename T>
mat<T> generate_spiral_mat(int n)
```

Given a positive integer  $n$ , you need to generate an  $n \times n$  mat object filled with elements from 1 to  $n^2$  in spiral order.

**Example 1:**

1 →	2 →	3 ↓
8 →	9 ↓	4 ↓
↑ 7 ←	6 ←	5

Input:  $n = 3$

Output: `[[1,2,3],[8,9,4],[7,6,5]]`

**Example 2:**

Input:  $n = 1$

Output: `[[1]]`

(Note that this function is not going to be a class member)

Code example:

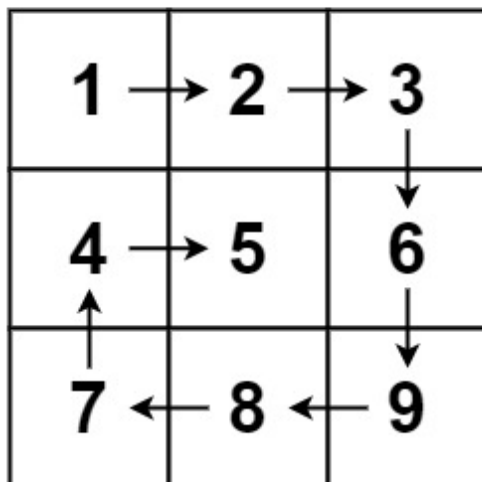
```
mat<int> m = generate_spiral_mat<int>(3);
vector<vector<int>> answ = m.to_vec_of_vec();
answ == { {1, 2, 3}, {8, 9, 4}, {7, 6, 5} }
```

- **PROBLEM #2**

```
vector<T> spiral_order() const
```

Given an  $m \times n$  mat object, return all the elements of the object in spiral order.

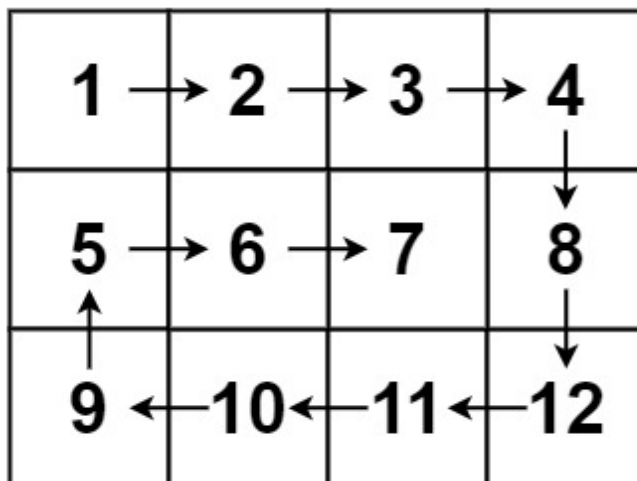
**Example 1:**



Input: matrix = [[1,2,3],[4,5,6],[7,8,9]]

Output: [1,2,3,6,9,8,7,4,5]

**Example 2:**



Input: matrix = [[1,2,3,4],[5,6,7,8],[9,10,11,12]]

Output: [1,2,3,4,8,12,11,10,9,5,6,7]

(Note that this function is going to be a member function and it's not mandatory to be a rectangle matrix: M can be different from N).

Code example:

```
mat<int> m({ {1, 2, 3}, {4, 5, 6}, {7, 8, 9} });  
vector<int> answ = m.spiral_order();  
answ == { 1, 2, 3, 6, 9, 8, 7, 4, 5 };
```

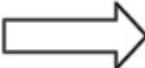
- **PROBLEM #3**

```
void rotate()
```

Add a rotate member function to the mat object and rotate the 2d- array by 90 degrees (clockwise). You have to rotate the object in-place, which means you have to modify the input mat directly. DO NOT allocate another 2D matrix and do the rotation.

**Example 1:**

1	2	3
4	5	6
7	8	9




7	4	1
8	5	2
9	6	3

Input: matrix = [[1,2,3],[4,5,6],[7,8,9]]

Output: [[7,4,1],[8,5,2],[9,6,3]]

**Example 2:**

5	1	9	11
2	4	8	10
13	3	6	7
15	14	12	16



15	13	2	5
14	3	4	1
12	6	8	9
16	7	10	11

Input: matrix = [[5,1,9,11],[2,4,8,10],[13,3,6,7],[15,14,12,16]]

Output: [[15,13,2,5],[14,3,4,1],[12,6,8,9],[16,7,10,11]]

Code example:

```
mat<int> m({ {1, 2, 3}, {4, 5, 6}, {7, 8, 9} });
m.rotate();
vector<vector<int>> answ = m.to_vec_of_vec();
answ == { {7, 4, 1}, {8, 5, 2}, {9, 6, 3} };
```

I implemented some tests to check the correctness of your implementation. You can use those tests to check yourself. If you implement everything right at the end you should see on the screen the following lines:

```
running spiral_order_tests
true
```

`true`  
`true`

*running spiral\_mat\_tests*  
`true`  
`true`

*running rotate\_tests*  
`true`  
`true`  
`true`

This is going to be the first part of our `mat` class on which we're going to build a very huge class with rich functionality by adding more functions.

## Second Part

- `template <typename F>`  
`friend void swap_(mat<F>&, mat<F>&)` - Friend swap function, which should be used in copy constructor, move constructor and move assignment operator.  
Example:  
`swap(*this, other);`
- `mat& operator=(const mat&)` - copy constructor, implemented by the "copy and swap" idiom.
- `mat(mat&&) noexcept` - Move constructor
- `mat& operator=(mat&&)` - Move assignment operator
- `template <typename T>`  
`mat<T> mat_mul(const mat<T>&, const mat<T>&)` - a function to do matrix multiplication. By the rules of matrix multiplication, you need to **throw** `logic_error("Bad shapes!")`; if the shapes of the given matrices are wrong (there is a test case inside the `mat_mul_tests` which checks this condition).

You can use our [previous class's code](#) as some hint. Try to check yourself by adding couts and debugging your code. Call with some different types and see what happens during and after the calls.

- **PROBLEM #4**  
`int max_square() const` (Should be a member function of our `mat` class)  
Given an `m`x`n` matrix (`mat<char>` object) filled with '0's and '1's, you need to find the largest square containing only 1s and return its area.  
Note: This function should work only on `mat<char>` objects, if the template parameter is not `char`, but another type, you need to throw an exception:  
`std::logic_error("incorrect type!")`; (There is a test checking this condition inside `max_square_tests`. To check the template parameter type you can use [std::is\\_same](#)).

**Example 1:**

1	0	1	0	0
1	0	1	1	1
1	1	1	1	1
1	0	0	1	0

Input: matrix = `[["1","0","1","0","0"],["1","0","1","1","1"],`  
`[["1","1","1","1","1"],["1","0","0","1","0"]]`

Output: 4

**Example 2:**

0	1
1	0

Input: matrix = `[["0","1"],["1","0"]]`

Output: 1

**Example 3:**

Input: matrix = `[["0"]]`

Output: 0

You can find more examples inside the `max_square_tests` function.

- **PROBLEM# 5**

`int num_islands()` (Should be a member function. Note that this function is not marked as 'const', so you can change the content when implementing your algorithm).

Given an  $m \times n$  2D matrix (`mat<char>` object) which represents a map of '1's (land) and '0's (water), return *the number of islands*.

An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

As for the previous problem, here too you need to throw

`std::logic_error("incorrect type!");` when the template type is not char.

**Example 1:**

```
Input: grid = [
  ["1","1","1","1","0"],
  ["1","1","0","1","0"],
  ["1","1","0","0","0"],
  ["0","0","0","0","0"]
]
Output: 1
```

**Example 2:**

```
Input: grid = [
  ["1","1","0","0","0"],
  ["1","1","0","0","0"],
  ["0","0","1","0","0"],
  ["0","0","0","1","1"]
]
Output: 3
```

This was the second part. Now you have 5 tests, so if you implemented everything correctly at the and you should see the following lines on your screen.

***running spiral\_order\_tests***

***true***

***true***

***true***

***running spiral\_mat\_tests***

***true***

***true***

*running rotate\_tests*

*true*

*true*

*true*

*running max\_square\_tests*

*true*

*true*

*true*

*true*

*true*

*true*

*true*

*running num\_islands\_tests*

*true*

*true*

*true*

*running mat\_mul\_tests*

*true*

*true*

Next time I'll add some tests for your constructors/overloaded operators and other functions too, to check not only the problems but the C++ implementation part too.