

# Assignment: `Matrix` type with a proxy class

## Learning Outcomes

This assignment will provide you with the knowledge and practice required to develop and implement software involving:

1. Data abstraction and encapsulation techniques using C++ classes.
2. RAII
3. Move semantics
4. Operator overloading
5. Class templates
6. Proxy class

## Task

In this assignment, you'll implement a C++ style API for matrices in `matrix.hpp`. You'll use some of the techniques discussed in class: dynamically allocated 2D arrays, subscripting, exception handling, move semantics, initializer lists, and proxy classes. The partial definition of the class template will look like this:

```

1  // file documentation header
2  // include guard
3  // standard library includes
4
5  namespace csd2125 {
6
7  template <typename T>
8  class Matrix {
9  public:
10     // provide common standard library container type definitions
11     // with using keyword ...
12
13  public:
14     // To allow clients to access values in an object m of type Matrix m
15     // using m[r][c] syntax, define a proxy class.
16     // Note that this nested class is simply a type and has no inherent access
17     // to any of outer class' members. Therefore, proxy class definition
18     // declares 2 data members: a Matrix& data member that references Matrix
19     // object instantiating the proxy object and a size_type data member
20     // indicating the matrix row. Suppose a call to Matrix object's member
21     // function (*this).op[](r) returns a proxy object constructed with a
22     // reference to *this and the value of (row) index.
23     // The proxy class will then define an overload of op[](size_type c) to
24     // return the value stored in the Matrix object's data store data[r].
25
26     // a second nested proxy class definition for Matrix const&
27
28     // ctors, dtor, copy, and move functions ...
29     Matrix(size_type nr, size_type nc);
30     Matrix(Matrix const& rhs);

```

```

31 Matrix(Matrix&& rhs) noexcept;
32 Matrix(std::initializer_list<std::initializer_list<value_type>>);
33 ~Matrix() noexcept;
34 Matrix& operator=(Matrix const& rhs);
35 Matrix& operator=(Matrix&& rhs) noexcept;
36
37 size_type get_rows() const noexcept;
38 size_type get_cols() const noexcept;
39
40 proxy-class-for-Matrix operator[](size_type r);
41 proxy-class-for-Matrix-const operator[](size_type r) const;
42
43 private:
44     size_type rows;
45     size_type cols;
46     pointer    data;
47 };
48
49 // declare global functions for following operator overloads:
50 // 1. +=: adding two Matrix<T> objects
51 // 2. -=: subtracting two Matrix<T> objects
52 // 3. *=: multiplying two Matrix<T> objects
53 // 4. ==: compare two Matrix<T> objects for equality
54 // 5. !=: compare two Matrix<T> objects for inequality

```

Here are the details of the class member functions.

`Matrix(nr, nc)` : (constructor) creates  $nr \times nc$  matrix. That is, a matrix with  $nr$  rows and  $nc$  columns. The storage for this matrix should be dynamically allocated.

`Matrix(rhs)` : (copy constructor, including move version) creates a matrix that is a deep copy of matrix `rhs`.

`Matrix(list)` : (constructor) creates  $nr \times nc$  matrix from an initializer list that has  $nr$  rows and  $nc$  columns. A `runtime_error` exception should be thrown if the initializer list doesn't have rows of equal size. The `runtime_error` exception object must be initialized with string "bad initializer list".

`~Matrix()` : (destructor) destroys the matrix by explicitly returning storage to free store.

`operator=(rhs)` : (assignment operator, including move version) replaces the matrix with a deep copy (or move) of matrix `rhs`.

`get_rows()` : returns number of rows in matrix.

`get_cols()` : returns number of columns in matrix.

`operator[](r)` : (subscripting operator, both const and non-const versions) returns index where  $r^{\text{th}}$  row of matrix data. The `const` version will return an object of nested proxy class that keeps references to `Matrix const` objects. The `non-const` version will return an object of nested proxy class that keeps references to `Matrix` objects.

The helper functions of the API are described below:

`operator+(M, N)` : returns matrix with sum  $M + N$  of matrices `M` and `N`. If dimensions of `M` and `N` are different, a `runtime_error` exception should be thrown. The `runtime_error` exception object must be initialized with string "operands for matrix addition must have same dimensions".

`operator-(M, N)` : returns matrix with difference  $M - N$  of matrices `M` and `N`. If dimensions of `M` and `N` are different, a `runtime_error` exception should be thrown. The `runtime_error` exception object must be initialized with string "operands for matrix subtraction must have same dimensions".

`operator*(M, N)` : returns matrix that is product  $MN$  of matrices `M` and `N`. Recall that if  $M$  has dimensions  $mr \times mc$  and  $N$  has dimensions  $nr \times nc$ , then  $MN$  is only defined if  $mc = nr$ . In this case, product  $MN$  has dimensions  $mr \times nc$ , and the element at the  $r^{\text{th}}$  row and  $c^{\text{th}}$  column of  $MN$  is given by the formula

$$(MN)_{rc} = \sum_{k=0}^{mc-1} M_{rk} N_{kc}$$

Use [this](#) tutorial if you need practice with matrix multiplication.

If matrices  $M$  and  $N$  cannot be multiplied, a `runtime_error` exception should be thrown. The `runtime_error` exception object must be initialized with string "number of columns in left operand must match number of rows in right operand".

`operator(r, M)` : returns matrix which is obtained by scaling every element of matrix `M` by scale factor `r`.

`operator==(M, N)` : returns `true` if matrices `M` and `N` are exactly equivalent; otherwise the function returns `false`.

`operator!=(M, N)` : return `true` if matrices `M` and `N` are not equivalent; otherwise the function returns `false`.

## Submission Details

---

Please read the following details carefully and adhere to all requirements to avoid unnecessary deductions.

### Submission files

You will submit `matrix.hpp` containing definition of class template `Matrix`, declarations of non-member functions, definitions of member functions outside definition class `Matrix`, and definitions of non-member functions.

### Compiling, linking, and testing

A driver file `matrix-test.cpp` is provided to test your implementation. `matrixtest.txt` contains the correct output generated by the driver. Practice using makefiles by refactoring the makefile from the previous assignment to compile, link, run the executable, and test the output.

### Valgrind is required

### Documentation

This module will use [Doxygen](#) to tag source and header files for generating html-based documentation. Every source and header file *must* begin with *file-level* documentation block. Every function that you declare and define and submit for assessment must contain *function-level documentation*. This documentation should consist of a description of the function, the inputs, and return value.

## Automatic evaluation

1. In the course web page, click on the appropriate submission page to submit `matrix.hpp`.
2. Please read the following rubrics to maximize your grade:
  - Your submission will receive an *F* grade if your submission doesn't compile with the full suite of `g++` options.
  - *F* grade if your submission doesn't link to create an executable.
  - Your implementation's output doesn't match correct output of the grader (you can see the inputs and outputs of the auto grader's tests). The auto grader will provide a proportional grade based on how many incorrect results were generated by your submission. *A+* grade if output of function matches correct output of auto grader.
  - A maximum of *D* grade if Valgrind detects even a single memory leak or error. A teaching assistance will check you submission for such errors.
  - A deduction of one letter grade for missing file-level documentation in `matrix.hpp`. A deduction of one letter grade for each missing function definition documentation block in `matrix.hpp`. Your submission must have **one** file-level documentation block and function-level documentation blocks for **every function you're defining**. A teaching assistant will physically read submitted source files to ensure that these documentation blocks are authored correctly. Each missing or incomplete or copy-pasted (with irrelevant information from some previous assessment) block will result in a deduction of a letter grade. For example, if the automatic grader gave your submission an *A+* grade and one documentation block is missing, your grade will be later reduced from *A+* to *B+*. Another example: if the automatic grade gave your submission a *C* grade and the two documentation blocks are missing, your grade will be later reduced from *C* to *E*.