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-proxy.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:

```
// file documentation header
// include guard
// standard library includes

namespace HLP3 {

template <typename T>
class Matrix {
public:
  // provide common standard library container type definitions
  // with using keyword ...

public:
  // To allow clients to access values in an object m of type Matrix m
  // using m[r][c] syntax, define a proxy class.
  // Note that this nested class is simply a type and has no inherent access
  // to any of outer class' members. Therefore, proxy class definition
  // declares 2 data members: a Matrix& data member that references Matrix
  // object instantiating the proxy object and a size_type data member
  // indicating the matrix row. Suppose a call to Matrix object's member
  // function (*this).op[](r) returns a proxy object constructed with a
  // reference to *this and the value of (row) index.
  // The proxy class will then define an overload of op[](size_type c) to
  // return the value stored in the Matrix object's data store data[r].

  // a second nested proxy class definition for Matrix const&

  // ctors, dtor, copy, and move functions ...
  Matrix(size_type nr, size_type nc);
  Matrix(Matrix const& rhs);
  Matrix(Matrix&& rhs) noexcept;
  Matrix(std::initializer_list<std::initializer_list<value_type>>);
  ~Matrix() noexcept;
  Matrix& operator=(Matrix const& rhs);
  Matrix& operator=(Matrix&& rhs) noexcept;

  size_type get_rows() const noexcept;
  size_type get_cols() const noexcept;

  proxy-class-for-Matrix operator[](size_type r);
  proxy-class-for-Matrix-const operator[](size_type r) const;

private:
  size_type rows;
  size_type cols;
  pointer   data;
};

// declare global functions for following operator overloads:
// 1. +:  adding two Matrix<T> objects
// 2. -:  subtracting two Matrix<T> objects
// 3. *:  multiplying two Matrix<T> objects
// 4. ==: compare two Matrix<T> objects for equality
// 5. !=: compare two Matrix<T> objects for inequality
```

> ***Since the 2D arrays are dynamically allocated, you should not include*** `<vector>` ***and*** `<deque>` ***in*** `matrix-proxy.hpp`***.***

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^{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^{th} row and c^{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-proxy.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. Since the 2D arrays are dynamically allocated, you should not include `<vector>` and `<deque>` in `matrix-proxy.hpp`.

Compiling, linking, and testing

A driver file `driver-matrix-proxy.cpp` is provided to test your implementation with 4 unit tests ranging and corresponding correct output files `out0` through `out3`. `outall` contains the correct output for all 4 tests. Practice using makefiles by refactoring the `makefile` from previous assignments 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 the appropriate file(s).
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.
   - F grade if Valgrind detects even a single memory leak or error.
   - A deduction of one letter grade for missing file-level documentation in each submitted file. A deduction of one letter grade for each missing function definition documentation block in files containing class definitions and function declarations. File-level documentation is not necessary in non-member functions and member functions defined outside a class. Your submission must have **one** file-level documentation block and function-level documentation blocks for ***every function you're declaring***. 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.