PA: `matrix` Class Template using DRY Principle

Learning Outcomes

- Gain experience in implementing ADTs using DRY principle.
- Gain experience in implementing class templates.
- Gain practice in overloading operators.
- Gain experience in reading and understanding code.

Task

In this exercise, you'll implement a `matrix<>` class inside header file `matrix.hpp`. Your `matrix` class should be capable of storing any type that supports the following: user-defined constructor [explained below], copy constructor, `=`, `==`, and `<<`. For numerical types [such as `int`, `double`, `std::complex<>`, and so on], `matrix<>` must additionally support operators `+`, `+=`, `-`, `-=`, `*`, `*=`. The unit tests in the driver check for matrices with elements of type `int`, `double`, `std::complex<int>`, `std::complex<double>`, and `std::string`.

The detailed interface for class `matrix<>` is:

> ***Member functions must be defined outside the class definition!!!***

1. You must provide constructors that allow clients to define and initialize matrices like this:

```
matrix<int> mi1;       // ERROR: default construction
matrix<int> mi2(2, 4); // OK: matrix of ints with 2 rows and 4 columns
                       // with each element initialized to zero
matrix<int> mi3(mi2);  // OK: mi3 is copy constructed
// matrix of std::strings with 3 rows and 6 columns with each element
// default constructed to ""
matrix<std::string> ms(3, 6); // OK
// matrix of std::complex<int>s with 3 rows and 4 columns with each
// element initialized to (0, 0)
matrix<std::complex<int>> mc(3, 4); // OK
```

> ***The arguments to the constructor will always be positive integers. It doesn't make sense to have a matrix with, for example, -2 rows and -1 columns.***

   The DRY principle dictates that you must use `std::vector<std::vector<T>>` where the inner `std::vector<T>` encapsulates a row of elements in the matrix.

2. You should overload the following operators:
   - Array subscript operator `[]`. Once you overload this operator, clients should be able to access the elements in a `matrix<T>` using the following syntax:

```
matrix<int> mi(2, 2);
mi[0][0] = 1; mi[0][1] = 2;
mi[1][0] = 3; mi[1][1] = 4;
```

- Equal to operator `==`. The function must return `true` if the two matrices have the same dimensions *and* the exact same rectangular arrangement of values.
- Not equal to operator `!=`. The function must return `true` if the two matrices have different dimensions or if they don't have the exact same rectangular arrangement of values.
- Left shift operator `<<`. Once you overload this operator, clients should be able to print the values encapsulated by the `matrix<T>` class, like this:

```
std::cout << "matrix mi:" << '\n' << mi;
```

The output printed to standard output will look like this:

```
matrix m:
1 2
3 4
```

> ***Pay attention to spacing between elements in the sample output file. Your output must exactly match the output in the sample - otherwise, the auto grader will not accept your submission!!!***

3. You should provide accessor member functions `Rows` and `Cols`:

```
matrix<int> mi(2, 2);
mi[0][0] = 1; mi[0][1] = 2;
mi[1][0] = 3; mi[1][1] = 4;
matrix<int> const mi2(mi);
std::cout << "mi2 has " << mi2.Rows() << " and "
          << mi2.Cols() << "columns.\n"
```

4. Clients of `matrix` class should be able to perform the following arithmetic operations [only for fundamental C++ numeric types and `std::complex<T>` where `T` is a fundamental C++ numeric type] that are well defined in matrix algebra:
   - Compound addition of 2 matrices using operator `+=`
   - Addition of 2 matrices using operator `+`
   - Compound subtraction of 2 matrices using operator `-=`
   - Subtraction of 2 matrices using operator `-`
   - Compound multiplication of 2 matrices using operator `*=`
   - Multiplication of 2 matrices using operator `*`
   - Multiplication of a matrix with a scalar
   - Multiplication of a scalar with a matrix

   Note that a class template usually supplies multiple operations on the template arguments it is instantiated for [including construction and destruction]. This might lead to the impression that these template arguments would have to provide all operations necessary for all member functions of a class template. But this is not the case. Template arguments only have to provide all necessary operations that *are* needed [instead of that *could* be needed]. Suppose, for example, class `matrix<>` provides a member function `*=` to implement matrix multiplication. You can still use `matrix<>` for elements such as `std::string` that don't have `operator*=` defined. Only if you call `operator*=` for such a matrix, the code will produce an error, because it can't instantiate the call to `operator*=` for this specific element type.

> All of this means that neither the driver nor your clients will expect matrix multiplication to work for non-numeric types such as `std::string`.

How to complete exercise

Comment out the entire body of `main` in driver `matrix-driver.cpp`. Create a file `matrix.hpp` with only the include guard and a comment at the top of the file. You should be able to compile and run.

In `matrix.hpp`, write a very simple *non-templated* outline of class `matrix`, as follows:

```
class matrix {
public:
  matrix(int rows, int cols);
};

matrix::matrix(int rows, int cols) {
  // **TODO**: fill this in later
}
```

Make sure you can still compile and run. In the next several steps, we will assume that class `matrix` can only represent `int`s. You will add support for other types towards the end.

`matrix-driver.cpp` contains several blocks of code, separated by blank lines. Uncomment the first block of code, and change every occurrence of `matrix<int>` to `matrix`. This is important because the `matrix` class is currently non-templated. The body of `main` should be as follows:

```
int main() {
  hlp2::matrix a(2, 4);
  std::cout << "matrix a has " << a.Rows() << " rows and "
            << a.Cols() << " columns" << "\n\n";
  std::cout << "matrix a:" << '\n';
  std::cout << a << '\n';
}
```

Add the required data members, define the constructor(s), `Rows`, and `Cols` in class `matrix`. After this, the code should compile and run correctly. You should also pass the first test, which is very similar to the block of code in `main` that you just uncommented.

Uncomment the next block of code in `main`, changing `matrix<int>` to `matrix`, and repeat the previous steps. As you do this multiple times, you will incrementally add methods to class `matrix` in `matrix.hpp`. The names of the tests should make it clear which tests should pass after you uncomment each block of code. You should be able to uncomment half of the body of `main` by the time this is complete. Be mindful of `const`-correctness; if you neglect it early on, it will cause problems later.

Once you have uncommented everything related to `matrix<int>`, you should be able to pass everything except the tests related to `matrix<std::complex<int>>` and `matrix<std::string>`. Now, you can add template information everywhere. Modify `matrix.hpp` so that class `matrix` is templated, and change all occurrences of `matrix` in `main` back to `matrix<int>`. You should be able to compile and run, and all tests should pass.

The only thing remaining is to deal with `matrix<std::complex<int>>` and `matrix<std::string>`. Learn about the interface provided by `std::complex<>` here.

Math

A matrix is nothing more than a rectangular arrangement of a set of elements of the *same type* `T`. Rectangles have sizes that are described by the number of rows of elements and columns of elements that they contain. The number of rows and columns of a matrix are called the *dimension* of the matrix. A matrix with dimension 3×4 has three rows and four columns; an $m \times m$ matrix has m rows and m columns. You may find these videos helpful if you're not aware of matrices.

Consider two matrices A and B with elements of the same type `T` that are of numerical type [as in the fundamental C++ types and the `std::complex<>` type]. If A and B have the same dimension $m \times n$, then their *sum* $A + B$ is defined and has dimension $m \times n$. Otherwise, their sum is not defined. The calculation of the sum of two matrices is explained here.

The *difference* of two matrices [for elements of type `T` that are of numerical type] is very similar to the sum of two matrices.

Consider a matrix A with dimension $m \times n$ and elements of type `T` (that are of numerical type) and a scalar s of type `T`. The *left scalar product* $s * A$ is defined and has dimension $m \times n$. The *right scalar product* $A * s$ is defined and has dimension $m \times n$. Then $s * A = A * s$, and we call this result the *scalar product*. The calculation of the scalar product is explained here.

Consider two matrices A and B with elements of the same type such that A has dimension $m_1 \times n_1$, while B has dimension $m_2 \times n_2$. If $n_1 == m_2$, then *matrix product* $A * B$ is defined and the resultant matrix will have dimension $m_1 \times n_2$. Otherwise, their matrix product is not defined. Matrix multiplication is associative, but is not necessarily commutative. The calculation of matrix product is explained here.

Submission Details

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

Submission files

You will be submitting file `matrix.hpp` containing definitions of both class template definition and member/non-member functions.

Compiling, executing, and testing

Download driver source file `matrix-driver.cpp` containing unit tests for `hlp2::matrix<T>` and corresponding output files containing correct output for these unit tests. Refactor a `makefile` from previous exercises to test your program.

File-level and function-level documentation

Every source and header file *must* begin with a *file-level* documentation block. This module will use Doxygen to tag source and header files for generating html-based documentation. In addition, 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.

Submission and automatic evaluation

1. In the course web page, click on the appropriate submission page to submit the necessary files.
2. Please read the following rubrics to maximize your grade. Your submission will receive:
    - 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 must exactly match correct output of the grader (you can see the inputs and outputs of the auto grader's tests). There are only two grades possible: $A+$ grade if your output matches correct output of auto grader; otherwise F.
    - 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 each missing documentation block in your submissions. Each source file must have **one** file-level documentation block and a function-level documentation block for each defined function. 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 F.