Programming Assignment: Adding Exceptions to `matrix` Class Template

Learning Outcomes

- Gain experience in generic programming by implementing class templates
- Gain experience in exception handling
- Gain practice in implementing derived classes.

Task

When you call a member function that modifies an object's state, you typically want one of two things to happen. Usually, of course, the function fully succeeds and brings the object into its desired new state. As soon as any error prevents a complete success, however, you really do not want to end up with an object in an unpredictable halfway state. Leaving a function's work half-finished mostly means that the object becomes unusable. Once anything goes wrong you instead prefer the object to remain in, or revert to its initial state. This all-or-nothing quality of a function is formally known as *strong exception safety*. Working with functions offering strong exception safety is easier because after calling a function offering the strong guarantee, there are only two possible program states: as expected following successful execution of the function, or the state that existed at the time the function was called. In contrast, after calling a function that doesn't offer this guarantee, the program could be in any valid state.

You'll be modifying your `matrix.hpp` from Assignment 8 to make it exception safe. Your `matrix.hpp` needs to handle the following 3 types of exceptions:

1. `InvalidDimension`
2. `IndexOutOfBounds`
3. `IncompatibleMatrices`

> ***All these 3 exception classes should inherit from base class `std::exception` declared in header file `<exception>`. You should define these 3 classes in file `matrix.hpp`.***

Throwing exception of type `InvalidDimension`

When a `hlp2::matrix<>` is defined with non-positive number of rows or non-positive number of columns, then your `matrix<>` constructor should throw an exception as shown below:

```
throw InvalidDimension(nrows, ncols);
```

where `nrows` and `ncols` are the rows and columns of the `hlp2::matrix<>` object that the client is trying to create and have type `int`. When the client catches this exception and prints the information of this exception object using member function `InvalidDimension::what()` then the client should see the error messages shown below.

1. **Case** 1**:** *Rows is negative*, as in:

```
try {
  hlp2::matrix<int> m(-2, 4);
} catch (hlp2::InvalidDimension const& e) {
  std::cout << e.what() << std::endl;
}
```

the message printed would look like this:

```
Invalid Dimension Exception: -2 is an invalid dimension for rows

```

2. **Case 2:** *Columns is negative*, as in:

```
try {
  hlp2::matrix<int> m(2, -4);
} catch (hlp2::InvalidDimension const& e) {
  std::cout << e.what() << std::endl;
}
```

the message printed would look like this:

```
Invalid Dimension Exception: -4 is an invalid dimension for columns
```

3. **Case 3:** *Both rows and columns are negative*, as in:

```
try {
  hlp2::matrix<int> m(-2, -4);
} catch (hlp2::InvalidDimension const& e) {
  std::cout << e.what() << std::endl;
}
```

the message printed would look like this:

```
Invalid Dimension Exception: -2 and -4 are invalid dimensions for rows and columns respectively
```

> ***Notice that there is no newline returned by member function*** `hlp2::InvalidDimension::what()`.

Throwing exception of type `IndexOutOfBounds`

When someone tries to access an out-of-bounds row, then `hlp2::matrix<>` class should thrown an exception as shown below:

```
throw IndexOutOfBounds(row_index);
```

where `row_index` is an invalid row index for accessing a `hlp2::matrix<>` object and is of type `int`. When the client catches this exception and prints the information of this exception object using the `what()` method then the client should see the error messages as shown below:

1. **Case 1:** The row index is *greater than the maximum row index*. When a client tries to access a row with index 3 in a 3×3 matrix:

```
try {
  hlp2::matrix<int> m(3, 3);
  m[3][2] = 10;
} catch (hlp2::IndexOutOfBounds const& e) {
  std::cout << e.what() << '\n';
}
```

the message would look like this:

```
Index Out Of Bounds Exception: 3 is an invalid index for rows

```

2. **Case 2:** The row index is *lower than the minimum row index*. When a client tries to access a row with index -1 in a 3×3 matrix:

```
try {
  hlp2::matrix<int> m(3, 3);
  m[-1][2] = 10;
} catch (std::exception const& e) {
  std::cout << e.what() << '\n';
}
```

the message would look like this:

```
Index Out Of Bounds Exception: -1 is an invalid index for rows

```

> ***Notice that there is no newline returned by member function*** `hlp2::IndexOutOfBounds::what()`.

Important Note: You need ***NOT*** worry about throwing an exception when the number of columns is out of range. When you store matrix elements using container `std::vector<std::vector<T>>`, your `matrix<>` class cannot detect if the number of columns is out of range when the array subscript notation is used. Can you think why this is not possible? If you can understand this, then you have understood operator overloading well! :) If you want to understand how can we check range for both the rows and columns in a matrix class, then read this.

Throwing exception of type `IncompatibleMatrices`

When arithmetic operations [addition, subtraction, multiplication] are performed between incompatible matrices, then your `matrix<>` class should throw an exception as shown below:

```
throw IncompatibleMatrices(operation, lRows, lCols, rRows, rCols);
```

where

- `operation` has type `std::string` representing operation [*addition* or *subtraction* or *multiplication*] type
- `lRows` has type `int` representing the number of rows in the left `matrix<>` operand
- `lCols` has type `int` representing the number of columns in the left `matrix<>` operand
- `rRows` has type `int` representing the number of rows in the right `matrix<>` operand
- `rCols` has type `int` representing the number of columns in the right `matrix<>` operand

When the client catches this exception and prints the information of this exception object using the `what()` method then the client should see some error messages as shown below:

1. **Case 1:** When a client writes an *addition* expression between incompatible matrices:

   ```cpp
   try {
     hlp2::matrix<int> m(3, 3);
     hlp2::matrix<int> n(4, 4);
     hlp2::matrix<int> r = m + n;
   } catch (hlp2::IncompatibleMatrices const& e) {
      std::cout << e.what() << '\n';
   }
   ```

   the message would look like this:

   ```
   Incompatible Matrices Exception: Addition of LHS matrix with dimensions 3 X 3 and RHS matrix with dimensions 4 X 4 is undefined

   ```

2. **Case 2:** When a client writes a *subtraction* expression between incompatible matrices:

   ```cpp
   try {
     hlp2::matrix<int> m(3, 3);
     hlp2::matrix<int> n(4, 4);
     hlp2::matrix<int> r = m - n;
   } catch (std::exception const& e) {
      std::cout << e.what() << '\n';
   }
   ```

   the message would look like this:

   ```
   Incompatible Matrices Exception: Subtraction of LHS matrix with dimensions 3 X 3 and RHS matrix with dimensions 4 X 4 is undefined

   ```

3. **Case 3:** When a client writes a *multiplication* expression between incompatible matrices:

   ```cpp
   try {
     hlp2::matrix<int> m(3, 4);
     hlp2::matrix<int> n(3, 4);
     hlp2::matrix<int> r = m * n;
   } catch (hlp2::IncompatibleMatrices const& e) {
      std::cout << e.what() << std::endl;
   }
   ```

   the message would look like this:

   ```
   Incompatible Matrices Exception: Multiplication of LHS matrix with dimensions 3 X 4 and RHS matrix with dimensions 3 X 4 is undefined

   ```

> ***Notice that there is no newline returned by member function*** `hlp2::IncompatibleMatrices::what()`.

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 class template definition, definitions of member/non-member functions, and the definitions of the three exception classes.

Compiling, executing, and testing

Download driver source file `matrixexp-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.