CS253 HW4: Time to get classy!

Description

For this assignment, you will write a standalone class called **`Ratio`**, which will represent a fraction, e.g., 2/3 or −6/5. Specifically, you will provide **`Ratio.h`**, which will contain the interface of that class, and the library **`libhw4.a`**, which will contain the implementation of that class.

Methods

Some methods are forbidden:

no default ctor
: The default (no-argument) ctor for **`Ratio`** must fail to compile. This is *not* a run-time error; it's a compile-time error.

no floating-point ctors
: **`Ratio(float)`**, **`Ratio(double)`**, and **`Ratio(long double)`** must all fail to compile.

`Ratio` must have the following public methods:

`Ratio(long numerator, long denominator)`
: Create a ratio representing the fraction numerator/denominator. If the denominator is not given, assume a denominator of one. If the denominator is zero, throw a `runtime_error` with an appropriate string.

`Ratio(int numerator, int denominator)`
: Same as above, but using `ints` rather than `longs`.

Copy constructor
: Copy all information from another object of the same class.

Assignment operator
: Copy all information from another object of the same class, replacing any previous information.

Destructor
: Destroy.

`.numerator()`
: Return the numerator as a `long`.

`.numerator(long)`
: Set the numerator.

`.denominator()`
: Return the denominator as a `long`.

`.denominator(long)`
: Set the denominator. If the denominator is zero, throw a `runtime_error` with an appropriate string, and leave the object unchanged.

`.ratio()`
: Return a **`long double`** representing the fraction. For example, **`Ratio(3,4).ratio()`** would return a **`long double`** with the value 0.75.

`.add(Ratio, …)`
: Add all the arguments to the current **`Ratio`**, resulting in yet another **`Ratio`**, returned by value. This method will not modify the current object, or any arguments. It is possible for this method to have one to eight **`Ratio`** arguments.

`.subtract(Ratio)`
: Subtract the argument from the current **`Ratio`**, resulting in a third **`Ratio`**, returned by value. This method will not modify the current object, or the argument.

`.multiply(Ratio)`
: Multiply the argument by the current **`Ratio`**, resulting in a third **`Ratio`**, returned by value. This method will not modify the current object, or the argument.

`.divide(Ratio)`
: Divide the current **`Ratio`** by the argument, resulting in a third **`Ratio`**, returned by value. This method will not modify the current object, or the argument. If this results in a divisor of zero, throw a `runtime_error` with an appropriate string.

`.compare(Ratio)`
: Compare the current object to another **`Ratio`**. If the current object is smaller, return an `int` less than zero. If the current object is larger, return an `int` greater than zero. If they're equal, return zero.

`.compare(long double)`
: Compare the **`.ratio()`** of the current object to a **`long double`**. If the current object is smaller, return an `int` less than zero. If the current object is larger, return an `int` greater than zero. If they're equal, return zero.

Non-methods:

`ostream << Ratio`
: Write the numerator, a slash, and the denominator to the `ostream`. Nothing else—no whitespace.

`istream >> Ratio`
: Read a **`long int`** numerator, a slash, and a **`long int`** denominator from the `istream`, skipping optional whitespace before each one. If an error occurs, set the state of the `istream` to failure, and leave the **`Ratio`** object unchanged. A zero divisor can either cause `istream` failure, `throw` a `runtime_error`, or both.

Const-correctness, for arguments, methods, and operators, is your job. For example, it must be possible to call **`.ratio()`** on a **`const Ratio`**, or to add a two **`const Ratio`** objects together.

You may define other methods or data, public or private, as you see fit. You may define other classes, as you see fit. However, to use the **`Ratio`** class, the user need only **`#include "Ratio.h"`**, not any other header files.

Normalization

A **`Ratio`** must be *normalized*:

- It must be reduced to lowest terms (66660/88880 becomes 3/4).
- The denominator must be positive (−4/−3 becomes 4/3, and 9/−100 becomes −9/100).
- If the numerator is zero, a non-zero denominator must be one (0/12554 becomes 0/1).

Errors

- All errors are indicated by `throwing` a `runtime_error` with an explanatory message. The exact string thrown is up to you, but should be descriptive and understandable by the TA.
- No **`Ratio`** method should call **`exit()`**, or produce any output.
- Numeric overflow (e.g., by multiplying two very large values) results in undefined behavior.

Libraries

`libhw4.a` is a *library file*. It contains a number of **`*.o`** (object) files. It must contain **`Ratio.o`**, but it may also contain whatever other **`*.o`** files you need. The **`CMakeLists.txt`** shown creates **`libhw4.a`**. It does *not* contain `main()`.

Testing

You will have to write a `main()` function to test your code. Put it in a separate file, and do **not** make it part of lib**`hw4.a`**. Particularly, do **not** put `main()` in **`Ratio.h`** or **`Ratio.cc`**. You will also have to create **`Ratio.h`**, and put it into **`hw4.tar`**. We will test your program by doing something like this:

```
mkdir a-new-directory
cd the-new-directory
tar -x </some/where/else/hw4.tar
cmake . && make
cp /some/other/place/test-program.cc .
g++ -Wall test-program.cc libhw4.a
./a.out
```

We will supply a main program to do the testing that we want. You should do something similar.

Sample Run

Here is a sample run, where `%` is my shell prompt:

```
% cat CMakeLists.txt
cmake_minimum_required(VERSION 3.14)
project(Homework)

# Using -Wall is required:
add_compile_options(-Wall)

# These compile flags are highly recommended, but not required:
add_compile_options(-Wextra -Wpedantic)

# Optional super-strict mode:
add_compile_options(-fmessage-length=80 -fno-diagnostics-show-option)
add_compile_options(-fstack-protector-all -g -O3 -std=c++14 -Walloc-zero)
add_compile_options(-Walloca -Wctor-dtor-privacy -Wduplicated-cond)
add_compile_options(-Wduplicated-branches -Werror -Wfatal-errors -Winit-self)
add_compile_options(-Wlogical-op -Wold-style-cast -Wshadow)
add_compile_options(-Wunused-const-variable=1 -Wzero-as-null-pointer-constant)

# add_compile_options must be BEFORE add_executable or add_library.

add_library(hw4 Ratio.cc)
add_executable(test test.cc)
target_link_libraries(test hw4)

# Create a tar file every time:
add_custom_target(hw4.tar ALL COMMAND tar cf hw4.tar Ratio.cc Ratio.h test.cc CMakeLists.txt)
% cat test.cc
#include "Ratio.h"
#include <cassert>
#include <fstream>
#include <iostream>
#include <stdexcept>

using std::cerr;
using std::cout;
using std::ifstream;
using std::runtime_error;

int main() {
    Ratio f12(-2,-4), f34(3L,4L), seven_eights = Ratio(7).divide(Ratio(8));
    const Ratio f78(seven_eights), one(f34.divide(f34));
    assert(f12.numerator() == 1);
    assert(f12.denominator() == 2);
    assert(f12.ratio() == 0.5);
    assert(f34.ratio() == 0.75);
    assert(f78.ratio() == 0.875);
    assert(f12.add(f34).multiply(f78).ratio() == 1.09375);
    assert(f12.ratio() == 0.5);
    assert(f34.ratio() == 0.75);
    assert(f78.ratio() == 0.875);
    assert(Ratio(1,5).add(Ratio(1,7)).compare(Ratio(12,35)) == 0);

    assert(f12.compare(f34) < 0);
    assert(f34.compare(f12) > 0);
    assert(f34.compare(Ratio(-66,-88)) == 0);
    assert(f12.add(f34).compare(Ratio(5,4)) == 0);
    assert(f12.add(f34).ratio() == 1.25);
    assert(f12.add(f34).compare(1.25L) == 0);
    assert(f12.add(f34, f78, f12, f34, f78).ratio() == 4.250);
    assert(f34.add(one,one,one,one,one,one,one,one).ratio() == 8.75);

    Ratio r(42, 666);
    r.numerator(0);
    r.numerator(700);
    r.denominator(-3000);
    assert(r.numerator() == -7);
    assert(r.denominator() == 30);

    cout << "Should be |-7/30|: |" << r << "|\n";

    ifstream in("data");
    if (!in) {
        cerr << "Can't open data\n";
        return 1;
    }
    while (in >> r)
        cout << "»»» " << r << '\n';

    try {
        r.denominator(0);
        cout << "Should not get here.\n";
    }
    catch (const runtime_error &err) {
        cout << "Error detected: "  << err.what() << '\n';
    }

    return 0;
}
% cat data
03/-4           -0/999

 0100 / +8
% cmake .
… cmake output appears here …
% make
… make output appears here …
% ./test
Should be |-7/30|: |-7/30|
»»» -3/4
»»» 0/1
»»» 25/2
Error detected: divisor of zero
```

Hints

- Fractions are tricky: ½ + ⅓ ≠ ⅖. Read Wikipedia for a refresher on fractions.
- This is an `int`: **`24`**. This is a `long`: **`68L`**. This is a `float`: **`1.2F`**. This is a `double`: **`3.4`**. This is a **`long double`**: **`5.6L`**.
- If you use `try`…`catch` in your **`Ratio`** code, you probably don't understand exceptions—seek help.
- "whitespace" is not just a fancy way of saying "space". It's what `isspace()` says it is.
- See https://en.cppreference.com/w/cpp/io/basic_ios/setstate for how to put a stream in a failed state.
- The foolish student will put `main()` in **`Ratio.cc`**, and try to remember to remove it before turning in the homework. Good luck with that. Just put it in a separate file.

Requirements

- You have permission to copy GCD (Greatest Common Divisor) or LCM (Least Common Multiple) code from a book or the internet. However, you **must** have a comment before the copied code citing the book (Author, title, page) or URL where you found the code.
- Only **`.add()`** takes from one to eight arguments, not **`.subtract()`**, **`.multiply()`**, or **`.divide()`**.
- In case of a failed input operation (>>), the position of the input stream is unspecified.
- You may use the **`CMakeLists.txt`** shown, or create your own.
- Do not put **`using namespace std;`** in any header file.
- All copies (copy ctor, assignment operator) are "deep". Do *not* share data between copies—that's not making a copy.

- You may not use any external programs via `system()`, `fork()`, `popen()`, `execl()`, `execvp()`, etc.
- You may not use C-style I/O, such as `printf()`, `scanf()`, `fopen()`, and `getchar()`.
  - Instead, use C++ facilities such as `cout`, `cerr`, and `ifstream`.
- You may not use dynamic memory via `new`, `delete`, `malloc()`, `calloc()`, `realloc()`, `free()`, `strdup()`, etc.
  - It's ok to *implicitly* use dynamic memory via containers such as `string` or `vector`.
- You may not use the `istream::eof()` method.
- No global variables.
- For readability, don't use ASCII `int` constants (**`65`**) instead of `char` constants (**`'A'`**) for printable characters.

- We will compile your program like this: **`cmake . && make`**
  - If that generates warnings, you will lose a point.
  - If that generates errors, you will lose *all* points.
- There is no automated testing/pre-grading/re-grading.
  - Test your code yourself. It's your job.
  - Even if you only change it a little bit.
  - Even if all you do is add a comment.

If you have any questions about the requirements, **ask**. In the real world, your programming tasks will almost always be vague and incompletely specified. Same here.

Tar file

- The tar file for this assignment must be called: **`hw4.tar`**
- It must contain:
  - source files (**`*.cc`**), including **`Ratio.cc`**
  - header files (**`*.h`**), including **`Ratio.h`**
  - **`CMakeLists.txt`**, which will create the library file lib**`hw4.a`**.
- These commands must produce the library lib**`hw4.a`**:
  `cmake . && make`
- Your **`CMakeLists.txt`** must use *at least* **`-Wall`** when compiling.

How to submit your homework:

- Use web checkin, or Linux checkin:

```
~cs253/bin/checkin HW4 hw4.tar
```

How to receive *negative* points:

Turn in someone else's work.