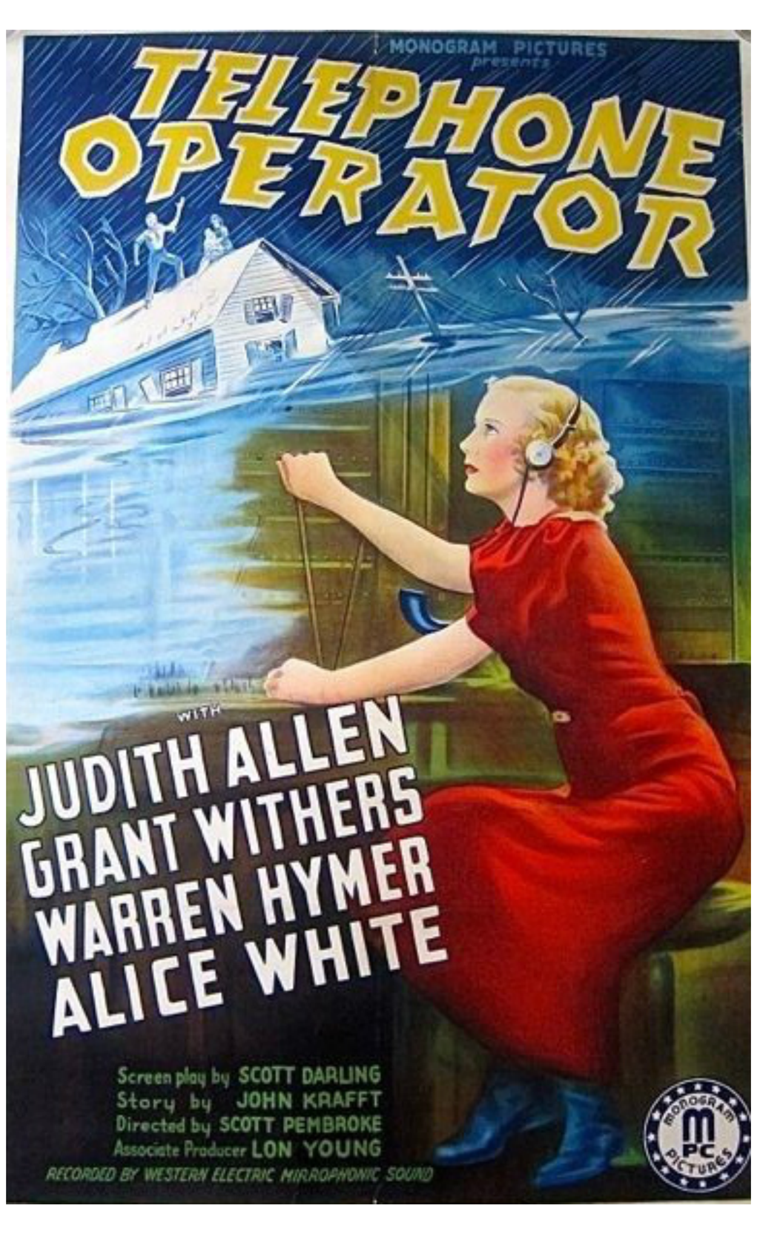
CS253 HW6: Operators!



Description

For this assignment, you will improve your `Ratio` class, replacing clumsy methods such as `.add()` with operators.

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 `int`s rather than `long`s.

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 `0.75L`.

`Ratio + Ratio`
`Ratio - Ratio`
`Ratio * Ratio`
`Ratio / Ratio`
: Add/subtract/multiply/divide the two ratios, yielding another `Ratio`, returned by value. Do not modify either operand. If division results in a divisor of zero, throw a `runtime_error` with an appropriate string. Either operand can also be `short`, `int`, or `long`. For example, for a `Ratio r`, `r+3` and `435L/r` are valid.

`Ratio += Ratio`
`Ratio -= Ratio`
`Ratio *= Ratio`
`Ratio /= Ratio`
: Combine the two operands with the corresponding arithmetic operation, and replace the left-hand side with the resulting value. Do not modify the right-hand operand. The right-hand operand can also be `short`, `int`, or `long`.

`Ratio == Ratio`
`Ratio != Ratio`
`Ratio <  Ratio`
`Ratio <= Ratio`
`Ratio >  Ratio`
`Ratio >= Ratio`
: Compare two ratios, return `true` if the condition is true. Ether operand can also be any of `short`, `int`, `long`, `float`, `double` or `long double`.

Non-methods:

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

`istream >> Ratio`
: Read a `long` numerator, a slash, and a `long` 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, operands, 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

Every operation must result in a *normalized* `Ratio`:

- 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 `throw`ing 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.

Hints

"OMG! int, short, long double! There are totally too many types! I have to write like nine thousand methods!!" Don't panic. You *don't* have to write methods for all of those. The various arithmetic types naturally promote, as needed. For example, if you write a method that takes a `double`, it will also take a `float`.

Libraries

`libhw6.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 `libhw6.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`hw6.a`. Particularly, do **not** put `main()` in `Ratio.h` or `Ratio.cc`. You will also have to create `Ratio.h`, and put it into `hw6.tar`. We will test your program by doing something like this:

```
mkdir a-new-directory
cd the-new-directory
tar -x < /some/where/else/hw6.tar
cmake . && make
cp /some/other/place/test-program.cc .@@
g++ -Wall test-program.cc libhw6.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(hw6 Ratio.cc)
add_executable(test test.cc)
target_link_libraries(test hw6)

# Create a tar file every time:
add_custom_target(hw6.tar ALL COMMAND tar cf hw6.tar Ratio.cc Ratio.h test.cc CMakeLists.txt)
% cat test.cc
#include "Ratio.h"
#include <cassert>
#include <sstream>
#include <iostream>
#include <stdexcept>

using std::cout;
using std::cerr;
using std::istringstream;
using std::runtime_error;

int main() {
    Ratio half(-100,-200), third(half);
    const Ratio five(10/2);
    third.numerator(24);             // now 24/2 ⇒ 12/1
    third.denominator(72);           // now 12/72 ⇒ 1/6
    third *= 2;                      // now 2/6 ⇒ 1/3
    assert(0.3333333333 < third && third < 0.3333333334);
    assert(0.3333333333 < third.ratio() && third.ratio() < 0.3333333334);
    assert(half == half);
    assert(0.5 == half);
    assert(five == 5);
    assert(6 != five);
    assert(half > third);
    assert(half >= third);
    assert(third < half);
    assert(third <= half);
    assert(third != half);
    assert(third + half == Ratio(50,60));
    assert(1/(third * half) == 6.0);
    Ratio a(1), b(2), c(3), d(4);
    a = b = c = d = half;
    assert(a == 0.5);
    assert(0+1/b+0 == 2);
    b /= d;
    assert(b == 1);
    assert(d == half);

    istringstream in(" 1/7\n \n123 / 456 4q5");    // 123/456 ⇒ 41/152
    if (!(in >> a >> b))
        cerr << "read failure\n";
    assert(a.numerator() == 1);
    assert(a.denominator() == 7);
    assert(b.denominator() == 152);
    assert(b.numerator() == 41);
    if (in >> c)
        cerr << "unexpected success\n";
    assert(1 == 2*c);              // must contain old value

    bool caught = false;
    try {
        Ratio bad(1, 0);
    }
    catch (const runtime_error &err) {
        caught = true;
    }
    assert(caught);

    cout << "Hooray!\n";
    return 0;
}
% cmake .
… cmake output appears here …
% make
… make output appears here …
% ./test
Hooray!
```

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.
- 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()`, `execv()`, …
- You may not use C-style I/O facilities such as `printf()`, `scanf()`, `fopen()`, `getchar()`, `getc()`, etc.
  - 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`.
- 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: `hw6.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`hw6.a`.
- These commands must produce the library lib`hw6.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 HW6 hw6.tar
```

How to receive *negative* points:

Turn in someone else's work.