Homework

30

A Fraction Class

Fractions are similar to floating-point numbers. Both can represent `1.5`, which is the fraction `3/2`. However, fractions (or **rational numbers**) are exact, while floating-point numbers **are approximations**.

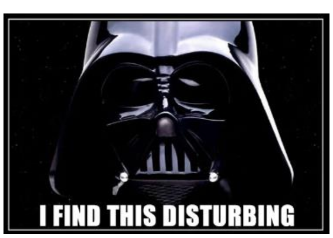
Consider adding together the following fractions:

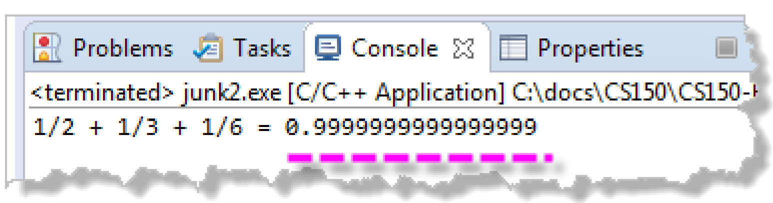
$$\frac{1}{2} + \frac{1}{3} + \frac{1}{6}$$

Even a fifth-grader can see that the answer is `1`, but you'd never know that if you used floating-point numbers on even the most advanced computer.

```
int main()
{
    double sum = 1.0 / 2.0 + 1.0 / 3.0 + 1.0 / 6.0;
    cout << fixed << setprecision(16);
    cout << "1/2 + 1/3 + 1/6 = " << sum << endl;
}
```

The result of running this program is a **little disturbing**:





The memory used to store floating-point numbers is limited, which in turn restricts the precision they can offer. Within the limits of floating-point numbers, the sum is closer to `0.9999999999999999` than it is to `1.0`. Worse still, at the end of the run, the value of the expression `sum < 1` would be `true`, and the value of `sum == 1` would be `false`.

Your Turn

Fractions are not subject to rounding errors; there are no approximations involved. Fractions have their own arithmetic rules which we can implement. Athough C++ does not include fractions as a library type, we can write our own.

Upload the starter code for `H30` and let's get started. You'll find some sample code that uses the `Fraction` class in `h30.cpp` in the `run()` function.

```
int main()
{
    Fraction a(3, 6); // 1/2 after reducing
    Fraction b(1, 3); // 1/3
    Fraction c(1, 6); // 1/6

    Fraction sum = a + b + c;

    cout << a << " + " << b << " + " << c
        << " = " << sum << endl;
}
```

Look in `h30.h` to find the interface for the `Fraction` class. You will need to implement:

- **Two constructors.** The default construct uses in-class initialization.
- Four **side-effect assignment operators** as member functions.
- Four **arithmetic non-member operators**
- A `toString()` conversion member function
- A non-member **output operator**.

Writing the Stubs

Start with the tried-and-true old programmer's trick, called a stub. You must declare the correct number of arguments, and give each method the correct name. Make sure that your stub method returns a value of the correct type.

1. **Copy all of the prototypes** from the header file (member and non-member) and past them into the implementation file.
2. **Qualify the names** by adding the class name with the scope resolution operator to the names of all of the **constructors** and **member functions**.
3. **Complete the stubs:** remove the semicolons at the end of each prototype, and, for those member functions that return a value, add a default or dummy return value and return it.

The member functions need full qualification, while the non-members do not. Your code should compile, link and test at this point.

Planning your Steps

Now that you have the stubs written, add an outline of the steps necessary to perform that particular operation. There are two advantages to doing this.

1. It makes sure you understand the algorithm; if you can't write the steps in English, it's not going to get any easier when you have to write it in C++.
2. The comments in the code will help you to remember what you intended when you wrote the program.

Here, for instance, are my comments for the `Fraction` arithmetic methods:

```
// Implementation notes for all arithmetic.
// 1. Get the 4 parts a, b, c, and d
//     a->numerator, b->denominator,
//     c->rhs.numerator, d->rhs.denominator
// 2. Construct a new Fraction number using
//    math calculations with a, b, c, d for numerator
//    and denominator.
// 3. Return the result
// add->Fraction(a*d + b*c, b*d);
```

You've undoubtedly been lectured about commenting your code since your very first programming class, and, perhaps, you've found it a burden. Commenting before you write the code--instead of trying to remember what you did last week--creates better comments and helps you create better designs.

Constructors & Invariants

The following constructor code is **obvious, but inadequate**:

```
Fraction::Fraction(int n, int d)
{
    numerator_ = n;
    denominator_ = d;
}
```

Inadequate constructor.

The rules of arithmetic place invariant constraints on the values of the numerator and denominator; the most obvious constraint is that the value of the denominator cannot be zero; the constructor should throw an exception.

There is a second issue. There are different ways to represent the same fractionional number. For example, one-third can be written in any of the following ways:

$$\frac{1}{3} \qquad \frac{2}{6} \qquad \frac{100}{300} \qquad \frac{-1}{-3}$$

All of these **represent the same Fraction**; they should have the same, unique representation. We can do that by following these rules:

1. The fraction is **expressed in lowest terms**. Divide both the numerator and the denominator by their **greatest common divisor**.
2. The **denominator is always positive**; the sign is stored with the numerator.
3. The **`Fraction 0`** is represented as **`0/1`**.

The Greatest Common Divisor

A **helper** function is one that is used as part of your implementation; it is not part of the interface, so it does not go in the header file. Using Euclid's algorithm to write a **`gcd()`** function like this is an obvious candidate:

```
static int gcd(int x, int y)
{
    int r = x % y;
    while (r != 0)
    {
        x = y;
        y = r;
        r = x % y;
    }
    return y;
}
```

The gcd helper function.

Making the function **`static`** means that it will only be used by the implementation of the **`Fraction`** class. It is not visible elsewhere. Place it first thing in your cpp file.

The Working Constructor

Here is the pseudocode for the working constructor:

```
Fraction(n, d):
   If d is 0 Then throw invalid_argument
   If n is 0 Then numerator(0), denominator(1)
   Else
      Let g = gcd(|n|, |d|)
      Let numerator = n / g
      Let denominator = |d| / g;
      If d is negative Then
         Let numerator be negative
```

Default & Conversion Constructors

You don't need a default constructor. The conversion constructor can just use the initializer list to set the denominator to 1 and the numerator to n.

C++11 delegating constructors can greatly simplify multiple overloaded constructors. Here is the code for our four constructors, using delegation. One point to note is that delegation can make instrumentation a little more difficult.

```
Fraction::Fraction(int n, int d) { /* working */ }
Fraction::Fraction(int n) : Fraction(n, 1) { }
```

Implementing the Methods

The code for the four arithmetic operations operators follows directly from the mathematical definitions. All four extract the values for **a**, **b**, **c** and **d** from the implicit and explicit parameters, and then apply the specific formulas from the mathematical definitions shown below.

Addition

$$\frac{a}{b} + \frac{c}{d} = \frac{ad + bc}{bd}$$

Multiplication

$$\frac{a}{b} \times \frac{c}{d} = \frac{ac}{bd}$$

Subtraction

$$\frac{a}{b} - \frac{c}{d} = \frac{ad - bc}{bd}$$

Division

$$\frac{a}{b} \div \frac{c}{d} = \frac{ad}{bc}$$

Overloaded Operators

Write the four side-effect member operators first, and then use those operators to write the four non-member addition and subratction operators. The CS 150 Reader shows you how to do it, but I'll show you a short example for addition:

```
Fraction& Fraction::operator+=(const Fraction& rhs)
{
    int a = numerator_;
    int b = denominator_;
    int c = rhs.numerator_;
    int d = rhs.denominator_;
    *this = Fraction(a * d + b * c, b * d);
    return *this;
}
```

The operator+=.

Once you have that, you can implement **`operator+()`** using it:

```
const Fraction
operator+(const Fraction& lhs, const Fraction& rhs)
{
    Fraction result(lhs);
    return result += rhs;
}
```

The operator+.

Writing *toString()*

To implement **`toString()`**, use the **`ostringstream`** class. Here's the pseudocode:

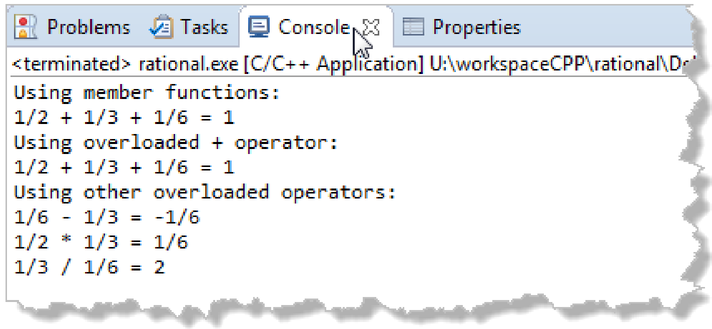
```
Let out be an ostringstream
Send numberator to out
If denominator is != 1 Then
   Send "/" and denominator to out
Return out.str()
```

Alternatively, since **`numerator_`** and **`denominator_`** are integers, you can use the C++11 standard library function ***`to_string()`***.

Overloaded Output Operator

All the overloaded operator does is to call the member function **`toString()`**, like this:

Once you've completed this operator, you can test them by expanding your original **`run()`** function using **`make stest`**. Here's the "smoke test" that I ran on my machine.



Run the instructor tests with **`make test`**. Use **`make submit`** to turn in your assignment. If you have difficulty, ask questions on Piazza or come to my office hours.