Data Structures and Objects
CSIS 3700

Lab 2 — Classes and Objects

Goal

Create a **`Fraction`** class whose objects can be used much like integers. The class must support the following:

- Creation of **`Fraction`** objects, specifying an initial numerator and denominator, just a numerator with a default denominator of `1`, or neither numerator nor denominator with a default value of `0/1`
- Assignment of one **`Fraction`** object to another using **`=`**
- Apply the four basic arithmetic operations to **`Fraction`** objects using **`+`**, **`-`**, **`*`** and **`/`**
- Compare two **`Fraction`** objects using **`==`**, **`<`**, *etc.*
- I/O using **`<<`** and **`>>`**

The fractions that are stored should always be in lowest terms; in addition, the denominator should never be negative.

Warning: Thou shalt not use any floating-point division in this lab!

What is a Class?

You will often encounter a situation where you have:

- Data that should only be manipulated in certain ways
- Manipulation functions that work with that data but are not useful in other contexts

These situations call for a *class*, which is a combination of data and code that manipulates the data into one entity.

Some examples:

- A **`Die`**, which consists of a count of the number of faces and the face that is currently the "top" face of the die; it also has a *roll* action to generate a random top face and a *look* action to look at the top face
- A **`Stack`**, which consists of a storage space — often an array — and a count of items in the space; it has a *push* action which adds a new value to the end of the list and a *pop* action which removes the most recently added item still in the container

Note that the top face, face count, backing store and item count are only meant to be used with their attendant functions, and roll, look, push and pop are only meant to work with their corresponding data. Since both code and data need one another, it makes sense to combine them into one entity.

▹ *Public and Private Regions*

Inside a class there are two regions: a *public* region which is accessible to all parts of a program, and a *private* region which is only accessible to functions inside the class.

Typically, data are in the private region and functions are in the public region. This hides the data so that the user of a class cannot access the data except through the class functions.

▹ *What's in a Fraction Class?*

To figure out what goes into a class, consider two types of things:

- The properties of the entity that you need to remember
- The actions that need to be performed on the data

Ponder these for a moment before reading on. . .

Obviously, a fraction needs a numerator and a denominator. Thinking ahead a bit, both of these should be signed integers even though the denominator is not going to be negative; negative denominators can be handled as they appear. No other information needs to be stored for a fraction.

The actions are the operations listed in the *Goal* section at the top of this document.

Interface and Implementation

To the extent possible, it is good practice to split a class into an *interface* and an *implementation*. These generally live in separate files.

The interface provides the user of a class with the list of allowed actions. It does not, however, explain how those actions are performed. The interface consists of the **`class`** definition, with the data declared and the function prototypes but not their bodies. Typically, the interface is placed in a header file with a **`.h`** extension.

The implementation contains the bodies for the functions — *methods* in object-oriented jargon — contained within the class. These are typically in a separate file; this provides for easier maintenance and reuse. Further, the user ordinarily doesn't need to know about the details of how the class functions work; they only need to know what functions are available and how to access them. For this, a prototype is sufficient.

Setting up the Interface File

Make sure you are in your course's work directory and not the course repo folder. Create a new folder for this lab and make it your current directory.

Edit the file **`fraction.h`** and add the following lines:

```
// defines int32_t and related aliases
#include <cstdint>

#ifndef _FRACTION_H
#define _FRACTION_H

// see how the _FRACTION_H looks like the file name? Not a coincidence!

// stuff goes here and nowhere else!

#endif
```

What you've set up is a guard; it prevents whatever is between **`#define`** and **`#endif`** from being shown to the compiler more than once. Telling the compiler about code and data once is good and necessary. The compiler will yell if you tell it about the same item twice.

Guards like this — *ifndef guards* — are only used in header files. Thus, your interface is placed inside the guard, but the implementation is not.

Next, set up the **`class`** skeleton:

```
// defines int32_t and related aliases
#include <cstdint>

#ifndef _FRACTION_H
#define _FRACTION_H

class Fraction {
 public:
 private:
};

#endif
```

Next, add the data. For our purposes, the numerator and denominator can be ordinary integers.

```
// defines int32_t and related aliases
#include <cstdint>

#ifndef _FRACTION_H
#define _FRACTION_H

class Fraction {
 public:
 private:
  int32_t
    num,
    den;
};

#endif
```

▹ *Constructors and Destructors*

Every class has two special functions — a *constructor* and a *destructor*. The constructor initializes data within an object, while a destructor performs any cleanup necessary when an object dies. Both functions use the same name as the class; the destructor has a tilde in front of the name.

Note that the destructor does not have anything to do; nothing has been dynamically allocated, no files have been opened, no actions are taken that need to be finished when a **`Fraction`** object dies. In this situation, best practice is to use a *default destructor*; the syntax is shown below.

Note: Default destructors originated with the C++ 11 standard. Your compiler may need to add a setting to use this standard (or a more recent one). I recommend adding the option `--std=c++17` to your command line; this will use the C++ 17 standard.

```
// defines int32_t and related aliases
#include <cstdint>

#ifndef _FRACTION_H
#define _FRACTION_H

class Fraction {
 public:
  Fraction(int32_t n=0,int32_t d=1);
  ~Fraction() = default;

 private:
  int32_t
    num,
    den;
};

#endif
```

Note the parameters for the constructor. These are called *default parameters*. They can be omitted; if they are, the given values are used in their place. Suppose we create the following three **`Fraction`** objects:

```
Fraction f1(2,3);
Fraction f2(5);
Fraction f3;
```

The object **`f1`** will have the value 2/3, **`f2`** will hold 5/1 since its denominator is missing and **`f3`** will be initialized to 0/1 since both numerator and denominator are missing.

▹ *Prototyping the Operations*

Class methods are prototyped in the same manner as any other function. For example, we could have:

```
  Fraction add(Fraction f1,Fraction f2);
```

With this, the statement

```
  result.add(a,b);
```

adds fractions **a** and **b**, placing the sum in the **`result`** object.

One of the goals of the project is to enable the use of arithmetic operators to perform arithmetic on **`Fraction`** objects. Some languages allow you to define how its operators work with user-defined objects, while others don't. C++ does allow redefinition, so we will take advantage of it.

The following adds the prototypes for addition and equality comparison. You will need to add prototypes for the other operations; they are similar to the examples provided. The prototype for assignment should look like the prototypes for the arithmetic operations.

```
#ifndef _FRACTION_H
#define _FRACTION_H

class Fraction {
 public:
  Fraction(int32_t n=0,int32_t d=1);
  ~Fraction() = default;

  // rhs == right hand side, as in the right hand side of an operator
  Fraction operator+(Fraction rhs);

  bool operator==(Fraction rhs);
 private:
  int32_t
    num,
    den;
};

#endif
```

▹ *Prototypes for I/O functions*

I/O operations are not part of the class; given the syntax for I/O in C++, I/O operations cannot be part of a class.

This presents a dilemma: I/O isn't part of the class, but it needs to access the data inside an object and the data is only accessible inside the class.

There are two solutions to this. One is to use *getter* and *setter* methods to provide the data values to the outside world.

The other solution is to use *friendship*. A class can specify that another class or outside function is allowed to access private data; that is, the other class or function is a friend.

In this exercise, we'll use two getter methods. Add these lines to the public region of the class:

```
  int32_t getNum() { return num; }  // getter for numerator
  int32_t getDen() { return den; }  // getter for denominator
```

Also, add these lines *outside* of the class but *inside* the guard.

```
  std::istream &operator>>(std::istream &,Fraction &);
  std::ostream &operator<<(std::ostream &,Fraction);
```

You'll also want to include **`iostream`** at the top of the file.

That's it for the interface. If you're unsure about anything, please show me what you have before continuing.

Implementation

For the implementation, edit the file **`fraction.cc`** and begin by including the header file.

The implementation file has the function bodies for all of the functions inside the class.

Tip: Copy and paste the prototypes into the implementation file and individually comment them out until you're ready to write the code for them. This gives you a list of what needs to be written and the prototypes are already there for you.

Let's start with the code for the constructor. It needs to do the following:

1. Make sure **d** ≥ 0
2. Make sure **n / d** is in lowest terms
3. Copy **n** and **d** to **num** and **den**, respectively

The first item is easy; if **d** < 0, flip the sign of both **n** and **d**.

For the second item, find the greatest common divisor of **n** and **d** and divide both by that value.

That leads to the following:

```
#include "fraction.h"

Fraction(int32_t n,int32_t d) {
  int32_t
    g;

  if (d < 0) {
    n = -n;
    d = -d;
  }

  g = gcd(n,d);

  num = n / g;
  den = d / g;
}
```

Save the files and try to compile **fraction.cc**; use the **-c** option to stop before linking, or you'll get an error about no **main()**.

If you had any errors, the compiler will tell you about them. If you had no errors, the compiler is still going to complain about **num** and **den** not being declared.

Why is this, if the variables are part of the class?

The reason is that the compiler does not know that this function is also part of the **Fraction** class. To do this, we must add the *namespace* to the function name. All functions in the **Fraction** class are in the **Fraction** namespace.

The correct form is:

```
#include "fraction.h"

Fraction::Fraction(int32_t n,int32_t d) {
  int32_t
    g;

  if (d < 0) {
    n = -n;
    d = -d;
  }

  g = gcd(n,d);

  num = n / g;
  den = d / g;
}
```

Note that the default parameter values are not given here. They are only given once, in the prototype inside the class.

The destructor does not need to do anything; it is an empty function. Generally, unless dynamic memory allocation or files are involved, a destructor is empty.

Before continuing, note that the constructor uses a **gcd** function that doesn't exist. We'll need to write it; place it above the constructor.

Note: If you are using the C++ 17 standard or later, there is already a **gcd** function available; add **#include <numeric>** to your header files to use it.

```
#include "fraction.h"

// static here tells the compiler that only functions in this file can access
// the function; functions in other files don't see it.
static int32_t gcd(int32_t a,int32_t b) {
  int32_t
    r;

  // ternary operator... go look it up, it's sometimes useful.
  a = (a >= 0) ? a : -a;
  b = (b >= 0) ? b : -b;

  while (b != 0) {
    r = a % b;
    a = b;
    b = r;
  }

  return a;
}

Fraction::Fraction(int32_t n,int32_t d) {
  int32_t
    g;

  if (d < 0) {
    n = -n;
    d = -d;
  }

  g = gcd(n,d);

  num = n / g;
  den = d / g;
}
```

Next, create the addition function. Given fractions $f_1 = \frac{a}{b}$ and $f_2 = \frac{c}{d}$, calculate

$$s = a \cdot d + b \cdot c \text{ and } t = b \cdot d$$

and return a new **Fraction** using s and t as its initial values.

Note that this uses the constructor, so $t \geq 0$ and the fraction being in lowest terms will be taken care of.

The implementation looks like this and is placed underneath the destructor:

```
Fraction Fraction::operator+(Fraction rhs) {
  int32_t
    s,
    t;

  s = num * rhs.den + den * rhs.num;
  t = den * rhs.den;

  return Fraction(s,t);
}
```

Given **`Fraction`** objects **`f1`** and **`f2`**, the calculation **`f1 + f2`** is converted to **`f1.operator+(f2)`**. Inside the **`operator+`** method, **`num`** and **`den`** belong to **`f1`**, **`rhs`** is a copy of **`f2`**, and **`s`** and **`t`** are local variables inside the method.

Subtraction is the same as addition, except **`+`** becomes **`-`** in the two places it appears. Multiplication multiplies numerators and denominators separately to create the resulting numerator and denominator. Division inverts the right hand operand and does a multiply. Don't actually invert the values; just swap **`rhs.num`** and **`rhs.den`** in the division code.

Implement those functions now.

▹ *Comparison Operations*

Comparing via **`==`** or **`!=`** is straightforward; compare numerators and compare denominators and return a result accordingly.

Hint: Learn DeMorgan's laws!

The code for equality looks like this; **`!=`** is similar:

```
bool Fraction::operator==(Fraction rhs) {

  return num == rhs.num && den == rhs.den;
}
```

To check for the inequalities, it is easiest to cross-multiply; after all, if $a/b < c/d$, isn't $a \cdot d < b \cdot c$?

Unfortunately, there's a problem with this. If $b \cdot d < 0$, then you're multiplying by a negative number, which flips the direction of the inequality. However, our fractions guarantee both $b \geq 0$ and $d \geq 0$, so this situation cannot occur. This simplifies our processing, removing the need to check for negative denominators and inequality flipping.

```
bool Fraction::operator<(Fraction rhs) {

  return num * rhs.den < den * rhs.num;
}
```

The remaining comparisons look almost exactly like this; just change the comparison operators accordingly.

▹ *I/O Operations*

Finally, we have the I/O operations. While the syntax for the prototype may look intimidating, it's an illusion; these are short, simple functions.

For output, we simply output the numerator, a slash character and the denominator. For input, we read the numerator, pass over the slash and read the denominator. In both cases, the function returns the stream object given as the first parameter. This allows input and output operations to be chained together. It's also the reason why you can't include these functions inside the class; they would expect a **`Fraction`** on the left side of the operator, not a stream.

```
istream &operator>>(istream &is,Fraction &f) {
  int32_t
    n,d;
  char
    slash;

  is >> n >> slash >> d; // reads numerator, the slash which is ignored, then
                         // denominator

  f = Fraction(n,d);     // set the fraction

  return is;             // I/O always returns the stream, for chaining >>
}

ostream &operator<<(ostream &os,Fraction f) {

  // those are spaces next to the slash!
  os << f.getNum() << "␣/␣" << f.getDen();

  return os;
}
```

Testing

Now, it's time to test your code. I've provided a test file **`ftest.cc`** that tests `Fraction` class. Compile both the **`fraction.cc`** and **`ftest.cc`** files.

Warning: Don't compile header files! This creates a compiled header file with a **`.gch`** extension that is the source of many troubles.

Run the program. If all of the tests pass, you're ready to submit. Otherwise, go back and fix the issues that the test program indicates.

▹ *A source of some bonus points*

Create a **`Makefile`** for this lab.

What to turn in

Turn in a .zip file of the lab 2 folder.