

Operator Overloading

Simple Fraction class

```
class Fraction
{
    private:
        int numerator;
        int denominator;

    public:
        Fraction() : numerator(1), denominator(1) {};
        Fraction(int num, int denom=1) : numerator(num),
        denominator(denom) {};
};
```

What will happen if we compile this line?

```
f3 = f1 + f2;
```

Simple Fraction class

```
class Fraction
{
private:
    int numerator;
    int denominator;

public:
    Fraction() : numerator(1), denominator(1) {};
    Fraction(int num, int denom=1) : numerator(num),
denominator(denom) {};
};
```

What will happen if we compile this line?

```
f3 = f1 + f2;
```

main.cpp:27:13: Invalid operands to binary expression ('Fraction' and 'Fraction')

What do we expect to happen with $f1+f2$?

- We expect to add the two fractions and put the result in the fraction $f3$
- Adding two fractions:
 - $f3.numerator = (f1.numerator * f2.denominator) + f2.numerator * f1.denominator$
 - $f3.denominator = f1.denominator * f2.denominator$
- However, C++ only sees a class. It does not know what this class represent or how to deal with adding two objects of this class together.
- But we can teach C++ how to do it!!

Operator Overloading

- Enables C++ to use operators with class objects
- The program will deal with the operator based on how we define it
- Which means it is our job to define meaningful operators because otherwise our programs can become unreadable!

Not all operators can be overloaded

Operators that can be overloaded							
+	-	*	/	%	^	&	
~	!	=	<	>	+=	-=	*=
/=	%=	^=	&=	=	<<	>>	>>=
<<=	==	!=	<=	>=	&&		++
--	->*	,	->	[]	()	new	delete
new[]	delete[]						

Operators that cannot be overloaded				
.	.*	::	?:	sizeof

Yes sizeof is an operator and not a function

We overload the operator + in Fraction class

```
class Fraction
{
```

```
private:
```

```
    int numerator;  
    int denominator;
```

```
public:
```

```
    Fraction() : numerator(1), denominator(1) {};  
    Fraction(int num, int denom=1) : numerator(num), denominator(denom) {};
```

```
    Fraction operator+(const Fraction& f){  
        Fraction r;  
        r.numerator=(numerator*f.denominator)+(denominator*f.numerator);  
        r.denominator=denominator*f.denominator;  
        return r;  
    }  
};
```

Return type is an object of Fraction after the adding occurs

The keyword operator specifies that this is an operator we are overloading followed by the operator which is + in this case

We are passing an object of Fraction as a const reference.

```
f3 = f1 + f2;
```

When running this again:

The operator + overload will be called on f1

And f2 will be passed by constant reference to our defined overloaded operator

The operator overload will return a new Fraction object and this new object will be assigned to f3

The values inside f3 after this line is executed will be: 10 and 12

Will the values of f1 and f2 change?

Exercise: overload the following operators to the Fraction class - $*$ and $/$

We can also overload the operator +=

```
f3 += f1;
```

```
Fraction& operator+=(const Fraction& f)
{
    *this = *this + f;
    return *this;
}
```

- Instead of redefining how to add two fractions, let's use the already overloaded operator +
- We are not creating a new object of Fraction and returning that as we want to store the values in f3 already.
- If we create a new object like we did before we will be creating a copy of f3 then adding f1 to it by making a new object and then replacing the existing f3
- In this case we are returning a reference to this object

If we add print at every constructor call: regular, default and copy constructor and then we run the following. What will be the output?

In other words, how many objects of Fraction are we creating and when?

```
Fraction f1(2,4), f2(1,3), f3;  
f3 += f1;
```

If we add print at every constructor call: regular, default and copy constructor and then we run the following. What will be the output?

In other words, how many objects of Fraction are we creating and when?

```
Fraction f1(2,4), f2(1,3), f3;  
f3 += f1;
```

```
regular constructor //for f1(2,4)  
regular constructor //for f2(1,3)  
default constructor //for f3  
default constructor //for f3 += f1  
//specifically when inside the += operator we are doing *this + f  
//the + operator overload creates a new object and returns it
```

We can show that by modifying our operator

```
Fraction& operator+=(const Fraction& f)
{
    std::cout << "before\n";
    *this = *this + f;
    std::cout << "after\n";

    return *this;
}
```

The output of the previous call is:

```
regular constructor
regular constructor
default constructor
before
default constructor
after
```

Can we make this overload perform better? Without making any new objects of Fraction?
Our goal is to put the values directly in f3

The new operator += overload

```
Fraction& operator+=(const Fraction& f)
{
    std::cout << "before\n";
    this->numerator = (numerator*f.denominator)+(denominator*f.numerator);
    this->denominator = denominator*f.denominator;
    std::cout << "after\n";
    return *this;
}
```

Output of previous program:

```
regular constructor
regular constructor
default constructor
before
after
```

We are rewriting the same code more or less
But it is more efficient

Exercise: Overload the operator -=

Now we want to print the values of the fraction


- We can define a `print()` function in the class that does that
- And every time we want to print an object of `Fraction` (call it `f`) we will have to call
- `f.print()`
- However, why can't we just pass our object to `std::cout` like this
- `std::cout << f;`
- We can! But we need to tell C++ what to do when an object of **`Fraction` is on the right hand side of the `<<` operator**
- **What is on the Left hand side of `<<` ??**

Let's start overloading the << operator

The operator is a friend
to be able to access the members

The LHS is ostream object like cout

The RHS is our Fraction object that
we are passing by constant reference



```
friend void operator<<(std::ostream& os, const Fraction& f)
{
    os << f.numerator << "/" << f.denominator;
}
```

Running these two lines:

```
Fraction f(1,4);
std::cout << f;
```

Will output:

```
1/4
```

What will the output be in each of these cases?

```
Fraction f(1,4);  
std::cout << "The value of the fraction" << f;
```

```
Fraction f(1,4);  
std::cout << f << "is the value of the fraction";
```

What will the output be in each of these cases?

```
Fraction f(1,4);  
std::cout << "The value of the fraction" << f;
```

The value of the fraction1/4

```
Fraction f(1,4);  
std::cout << f << "is the value of the fraction";
```

This will give an error! After the << f we are not returning anything
The LHS of the second << is basically void now

How can we fix that?

We will change the return of the operator to return the output stream so it can be used with the second <<

```
friend std::ostream& operator<<(std::ostream& os, const Fraction& f)
{
    os << f.numerator << "/" << f.denominator;
    return os;
}
```

Exercise: overload the >> operator

Now can we do the same thing for boolean operators?

- For example we want to do this:

```
Fraction f1(1,4), f2(1,4);  
if(f1==f2)  
{  
    std::cout << "equal";  
}
```

Overloading == operator

```
bool operator==(const Fraction& f)
{
    if(numerator == f.numerator && denominator == f.denominator)
        return true;
    else
        return false;
}
```

This operator is member of our class Fraction

The left hand side of the operator is the object itself on the left hand side

Is this a good comparison between two fractions?

Exercise: fix it and overload >, <, <=, >= and != operators

Can we do this?

```
Fraction f1(1,3);  
  
if(2==f1)  
    std::cout << "equal";  
else  
    std::cout << "not equal";
```


Can we do this?

```
Fraction f1(1,3);  
  
if(2==f1)  
    std::cout << "equal";  
else  
    std::cout << "not equal";
```

The operator we overloaded takes an object of Fraction on the right hand side
And the operator needs to be called on an object of Fraction, which on the left of the operator

And in this case the left hand side of type int
There isn't an overload of the == operator that takes
int on the LHS and Fraction on the RHS

How about this?

```
Fraction f1(1,3);  
  
if(f1==2)  
    std::cout << "equal";  
else  
    std::cout << "not equal";
```

Can we do this?

```
Fraction f1(1,3);  
  
if(f1==2)  
    std::cout << "equal";  
else  
    std::cout << "not equal";
```

You will probably say that it will not work! Because the operator we overloaded takes an object of Fraction on the RHS and not int

And in this case the right hand side of type int

HOWEVER, an implicit call to the regular constructor will be called and the 2 will become implicitly casted to a Fraction object using the regular constructor Fraction(int num, denom=1)

The output of this program with the prints from the constructors is:

```
regular constructor //first constructor for f1  
regular constructor //second constructor call is done implicitly to create a Fraction(2,1)  
not equal
```

SIDE NOTE

We can specify that the constructor can only be called explicitly and not implicitly by using the `explicit` keyword

```
explicit Fraction(int num, int denom=1) : numerator(num), denominator(denom)
{
    std::cout << "regular constructor" << std::endl;
};
```

In this case the constructor can only be called explicitly and the program before will not compile

We can still use that operator and do a cast ourselves if we want

```
Fraction f1(1,3);  
  
if(f1 == static_cast<Fraction>(2))  
    std::cout << "equal";  
else  
    std::cout << "not equal";
```

This will call the regular constructor explicitly as we are doing the cast

We can define an operator overload as not a member function. It can be a friend function

```
friend bool operator==(const Fraction& LF, const Fraction& RF)
{
    return !(LF.numerator*RF.denominator - RF.numerator*LF.denominator);
}
friend bool operator==(const int& num, const Fraction& RF)
{
    return Fraction(num)==RF;
}
```

Note that we are creating a new Fraction object when inside the overload that takes int as Fraction

Can we do it better?

Yes we can

```
friend bool operator==(const int& num, const Fraction& RF)
{
    if(RF.numerator % RF.denominator)
        return false; //the fraction is not an integer
    else
        return num == RF.numerator / RF.denominator;
}
```

Note that we are not constructing a new object of Fraction anymore
Can we do this now:

```
if(f1 == 2)
    std::cout << "equal";
else
    std::cout << "not equal";
```

Our operator takes int on the left side and Fraction on the right side. But this is the opposite. We can overload the == operator again

```
friend bool operator==(const Fraction& LF, const int& num)
{
    return num == LF;
}
```

We are simply using the operator overload that we defined before

Exercise: overload the operators \geq , \leq , \neq , $<$, and $>$ that takes `int` on one side and `Fraction` on the other side

Code can be found here:

- cpp.sh/847mo