9.2 — Overloading the arithmetic operators using friend functions

BY ALEX ON SEPTEMBER 26TH, 2007 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

Some of the most commonly used operators in C++ are the arithmetic operators -- that is, the plus operator (+), minus operator (-), multiplication operator (*), and division operator (/). Note that all of the arithmetic operators are binary operators -- meaning they take two operands -- one on each side of the operator. All four of these operators are overloaded in the exact same way.

It turns out that there are three different ways to overload operators: the member function way, the friend function way, and the normal function way. In this lesson, we'll cover the friend function way (because it's more intuitive for most binary operators). Next lesson, we'll discuss the normal function way. Finally, in a later lesson in this chapter, we'll cover the member function way. And, of course, we'll also summarize when to use each in more detail.

Overloading operators using friend functions

Consider the following trivial class:

```
class Cents
{
private:
    int m_cents;

public:
    Cents(int cents) { m_cents = cents; }
    int getCents() const { return m_cents; }
};
```

The following example shows how to overload operator plus (+) in order to add two "Cents" objects together:

```
#include <iostream>

class Cents
{
private:
    int m_cents;

public:
    Cents(int cents) { m_cents = cents; }

    // add Cents + Cents using a friend function
    friend Cents operator+(const Cents &c1, const Cents &c2);

    int getCents() const { return m_cents; }
};

// note: this function is not a member function!
Cents operator+(const Cents &c1, const Cents &c2)
{
    // use the Cents constructor and operator+(int, int)
    // we can access m_cents directly because this is a friend function
    return Cents(c1.m_cents + c2.m_cents);
}

int main()
{
```

```
    Cents cents1(6);
    Cents cents2(8);
    Cents centsSum = cents1 + cents2;
    std::cout << "I have " << centsSum.getCents() << " cents." << std::endl;

    return 0;
}
```

This produces the result:

```
I have 14 cents.
```

Overloading the plus operator (+) is as simple as declaring a function named operator+, giving it two parameters of the type of the operands we want to add, picking an appropriate return type, and then writing the function.

In the case of our Cents object, implementing our operator+() function is very simple. First, the parameter types: in this version of operator+, we are going to add two Cents objects together, so our function will take two objects of type Cents. Second, the return type: our operator+ is going to return a result of type Cents, so that's our return type.

Finally, implementation: to add two Cents objects together, we really need to add the m_cents member from each Cents object. Because our overloaded operator+() function is a friend of the class, we can access the m_cents member of our parameters directly. Also, because m_cents is an integer, and C++ knows how to add integers together using the built-in version of the plus operator that works with integer operands, we can simply use the + operator to do the adding.

Overloading the subtraction operator (-) is simple as well:

```
#include <iostream>

class Cents
{
private:
    int m_cents;

public:
    Cents(int cents) { m_cents = cents; }

    // add Cents + Cents using a friend function
    friend Cents operator+(const Cents &c1, const Cents &c2);

    // subtract Cents - Cents using a friend function
    friend Cents operator-(const Cents &c1, const Cents &c2);

    int getCents() const { return m_cents; }
};

// note: this function is not a member function!
Cents operator+(const Cents &c1, const Cents &c2)
{
    // use the Cents constructor and operator+(int, int)
    // we can access m_cents directly because this is a friend function
    return Cents(c1.m_cents + c2.m_cents);
}

// note: this function is not a member function!
Cents operator-(const Cents &c1, const Cents &c2)
{
    // use the Cents constructor and operator-(int, int)
    // we can access m_cents directly because this is a friend function
```

```
    return Cents(c1.m_cents - c2.m_cents);
}

int main()
{
    Cents cents1(6);
    Cents cents2(2);
    Cents centsSum = cents1 - cents2;
    std::cout << "I have " << centsSum.getCents() << " cents." << std::endl;

    return 0;
}
```

Overloading the multiplication operator (*) and the division operator (/) is as easy as defining functions for operator* and operator/ respectively.

Friend functions can be defined inside the class

Even though friend functions are not members of the class, they can still be defined inside the class if desired:

```
#include <iostream>

class Cents
{
private:
    int m_cents;

public:
    Cents(int cents) { m_cents = cents; }

    // add Cents + Cents using a friend function
        // This function is not considered a member of the class, even though the definition i
    friend Cents operator+(const Cents &c1, const Cents &c2)
    {
        // use the Cents constructor and operator+(int, int)
        // we can access m_cents directly because this is a friend function
        return Cents(c1.m_cents + c2.m_cents);
    }

    int getCents() const { return m_cents; }
};

int main()
{
    Cents cents1(6);
    Cents cents2(8);
    Cents centsSum = cents1 + cents2;
    std::cout << "I have " << centsSum.getCents() << " cents." << std::endl;

    return 0;
}
```

We generally don't recommend this, as non-trivial function definitions are better kept in a separate .cpp file, outside of the class definition. However, we will use this pattern in future tutorials to keep the examples concise.

Overloading operators for operands of different types

Often it is the case that you want your overloaded operators to work with operands that are different types. For example, if we have Cents(4), we may want to add the integer 6 to this to produce the result Cents(10).

When C++ evaluates the expression x + y, x becomes the first parameter, and y becomes the second parameter. When x and y have the same type, it does not matter if you add x + y or y + x -- either way, the same version of

operator+ gets called. However, when the operands have different types, x + y does not call the same function as y + x.

For example, `Cents(4) + 6` would call operator+(Cents, int), and `6 + Cents(4)` would call operator+(int, Cents). Consequently, whenever we overload binary operators for operands of different types, we actually need to write two functions -- one for each case. Here is an example of that:

```
#include <iostream>

class Cents
{
private:
    int m_cents;

public:
    Cents(int cents) { m_cents = cents; }

    // add Cents + int using a friend function
    friend Cents operator+(const Cents &c1, int value);

    // add int + Cents using a friend function
    friend Cents operator+(int value, const Cents &c1);


    int getCents() const { return m_cents; }
};

// note: this function is not a member function!
Cents operator+(const Cents &c1, int value)
{
    // use the Cents constructor and operator+(int, int)
    // we can access m_cents directly because this is a friend function
    return Cents(c1.m_cents + value);
}

// note: this function is not a member function!
Cents operator+(int value, const Cents &c1)
{
    // use the Cents constructor and operator+(int, int)
    // we can access m_cents directly because this is a friend function
    return Cents(c1.m_cents + value);
}

int main()
{
    Cents c1 = Cents(4) + 6;
    Cents c2 = 6 + Cents(4);

    std::cout << "I have " << c1.getCents() << " cents." << std::endl;
    std::cout << "I have " << c2.getCents() << " cents." << std::endl;

    return 0;
}
```

Note that both overloaded functions have the same implementation -- that's because they do the same thing, they just take their parameters in a different order.

Another example

Let's take a look at another example:

```
class MinMax
```

```
{
private:
    int m_min; // The min value seen so far
    int m_max; // The max value seen so far

public:
    MinMax(int min, int max)
    {
        m_min = min;
        m_max = max;
    }

    int getMin() { return m_min; }
    int getMax() { return m_max; }

    friend MinMax operator+(const MinMax &m1, const MinMax &m2);
    friend MinMax operator+(const MinMax &m, int value);
    friend MinMax operator+(int value, const MinMax &m);
};

MinMax operator+(const MinMax &m1, const MinMax &m2)
{
    // Get the minimum value seen in m1 and m2
    int min = m1.m_min < m2.m_min ? m1.m_min : m2.m_min;

    // Get the maximum value seen in m1 and m2
    int max = m1.m_max > m2.m_max ? m1.m_max : m2.m_max;

    return MinMax(min, max);
}

MinMax operator+(const MinMax &m, int value)
{
    // Get the minimum value seen in m and value
    int min = m.m_min < value ? m.m_min : value;

    // Get the maximum value seen in m and value
    int max = m.m_max > value ? m.m_max : value;

    return MinMax(min, max);
}

MinMax operator+(int value, const MinMax &m)
{
    // call operator+(MinMax, int)
    return m + value;
}

int main()
{
    MinMax m1(10, 15);
    MinMax m2(8, 11);
    MinMax m3(3, 12);

    MinMax mFinal = m1 + m2 + 5 + 8 + m3 + 16;

    std::cout << "Result: (" << mFinal.getMin() << ", " <<
        mFinal.getMax() << ")\n";

    return 0;
}
```

The MinMax class keeps track of the minimum and maximum values that it has seen so far. We have overloaded the + operator 3 times, so that we can add two MinMax objects together, or add integers to MinMax objects.

This example produces the result:

```
Result: (3, 16)
```

which you will note is the minimum and maximum values that we added to mFinal.

Let's talk a little bit more about how "MinMax mFinal = m1 + m2 + 5 + 8 + m3 + 16" evaluates. Remember that operator+ has higher precedence than operator=, and operator+ evaluates from left to right, so m1 + m2 evaluate first. This becomes a call to operator+(m1, m2), which produces the return value MinMax(8, 15). Then MinMax(8, 15) + 5 evaluates next. This becomes a call to operator+(MinMax(8, 15), 5), which produces return value MinMax(5, 15). Then MinMax(5, 15) + 8 evaluates in the same way to produce MinMax(5, 15). Then MinMax(5, 15) + m3 evaluates to produce MinMax(3, 15). And finally, MinMax(3, 15) + 16 evaluates to MinMax(3, 16). This final result is then assigned to mFinal.

In other words, this expression evaluates as "MinMax mFinal = (((((m1 + m2) + 5) + 8) + m3) + 16)", with each successive operation returning a MinMax object that becomes the left-hand operand for the following operator.

Implementing operators using other operators

In the above example, note that we defined operator+(int, MinMax) by calling operator+(MinMax, int) (which produces the same result). This allows us to reduce the implementation of operator+(int, MinMax) to a single line, making our code easier to maintain by minimizing redundancy and making the function simpler to understand.

It is often possible to define overloaded operators by calling other overloaded operators. You should do so if and when doing so produces simpler code. In cases where the implementation is trivial (e.g. a single line) it's often not worth doing this, as the added indirection of an additional function call is more complicated than just implementing the function directly.

Quiz time

1a) Write a class named Fraction that has a integer numerator and denominator member. Write a print() function that prints out the fraction.

The following code should compile:

```
#include <iostream>

int main()
{
    Fraction f1(1, 4);
    f1.print();

    Fraction f2(1, 2);
    f2.print();

    return 0;
}
```

This should print:

```
1/4
1/2
```

Show Solution

1b) Add overloaded multiplication operators to handle multiplication between a Fraction and integer, and between two Fractions. Use the friend function method.

Hint: To multiply two fractions, first multiply the two numerators together, and then multiply the two denominators together. To multiply a fraction and an integer, multiply the numerator of the fraction by the integer and leave the denominator alone.

The following code should compile:

```
#include <iostream>

int main()
{
    Fraction f1(2, 5);
    f1.print();

    Fraction f2(3, 8);
    f2.print();

    Fraction f3 = f1 * f2;
    f3.print();

    Fraction f4 = f1 * 2;
    f4.print();

    Fraction f5 = 2 * f2;
    f5.print();

    Fraction f6 = Fraction(1, 2) * Fraction(2, 3) * Fraction(3, 4);
    f6.print();

    return 0;
}
```

This should print:

```
2/5
3/8
6/40
4/5
6/8
6/24
```

Show Solution

1c) Extra credit: the fraction 2/4 is the same as 1/2, but 2/4 is not reduced to the lowest terms. We can reduce any given fraction to lowest terms by finding the greatest common divisor (GCD) between the numerator and denominator, and then dividing both the numerator and denominator by the GCD.

The following is a function to find the GCD:

```
int gcd(int a, int b) {
    return (b == 0) ? (a > 0 ? a : -a) : gcd(b, a % b);
}
```

Add this function to your class, and write a member function named reduce() that reduces your fraction. Make sure all fractions are properly reduced.

The following should compile:

```
#include <iostream>

int main()
{
    Fraction f1(2, 5);
    f1.print();

    Fraction f2(3, 8);
    f2.print();

    Fraction f3 = f1 * f2;
    f3.print();

    Fraction f4 = f1 * 2;
    f4.print();

    Fraction f5 = 2 * f2;
    f5.print();

    Fraction f6 = Fraction(1, 2) * Fraction(2, 3) * Fraction(3, 4);
    f6.print();

    Fraction f7(0, 6);
    f7.print();


    return 0;
}
```

And produce the result:

```
2/5
3/8
3/20
4/5
3/4
1/4
0/6
```

Show Solution

9.2a -- Overloading operators using normal functions

Index

9.1 -- Introduction to operator overloading

C++ TUTORIAL | PRINT THIS POST

280 comments to 9.2 — Overloading the arithmetic operators using friend functions

« Older Comments 1 2 3 4

nucod3r
January 29, 2020 at 5:22 pm · Reply

Why does this work

```
void reduce()
    {
        if (m_numerator != 0 && m_denominator != 0)
        {
            int gcd = Fraction::gcd(m_numerator, m_denominator);
            m_numerator /= gcd;
            m_denominator /= gcd;
        }
    }
```

But this doesn't

```
void reduce()
    {
        if (m_numerator != 0 && m_denominator != 0)
        {
            int gcd = gcd(m_numerator, m_denominator);
            m_numerator /= gcd;
            m_denominator /= gcd;
        }
    }
```

nascardriver
January 30, 2020 at 2:32 am · Reply

Because there are 2 things called "gcd". The `int gcd` is the closest `gcd` (Closest scope) to that line, so it `gcd` resolves to the `int gcd`. You can't call an `int`, so you get an error. If you rename either of the `gcd`s, the code works.

Ged
January 6, 2020 at 6:19 am · Reply

Suggestion number 1

This was my first code. I tried to do it a bit differently, I did the division when multiplying. The program works, but it has a bug. If you create a dividable object, it will not divide it. So my suggestion is to add an 8th object. I corrected my code already, but someone may not see this point.

```
Fraction f8(3, 6);
f8.print();
```

```
class Fraction
{
private:
    int m_numerator;
    int m_detonator;

public:
    Fraction(int numerator, int detonator)
        : m_numerator(numerator), m_detonator(detonator) {}

    void print() const
    {
        std::cout << m_numerator << '/' << m_detonator << '\n';
    }

    friend int gcd(int a, int b);


    friend Fraction operator*(const Fraction &m1, const Fraction &m2);
    friend Fraction operator*(const Fraction &m, int value);
    friend Fraction operator*(int value, const Fraction& m);
};

int gcd(int a, int b)
{
    if (a == 0)
        return b;
    return gcd(b % a, a);
}

Fraction operator*(const Fraction& m1, const Fraction& m2)
{
    int numerator = m1.m_numerator * m2.m_numerator;
    int detonator = m1.m_detonator * m2.m_detonator;

    int division = gcd(numerator, detonator);

    if (division > 1)
        return Fraction(numerator / division, detonator / division);
    else
        return Fraction(numerator, detonator);
}

Fraction operator*(const Fraction& m, int value)
{
    int numerator = m.m_numerator * value;

    int division = gcd(numerator, m.m_detonator);

    if (division > 1)
```

```
        return Fraction(numerator / division, m.m_detonator / division);
    else
        return Fraction(numerator, m.m_detonator);
}

Fraction operator*(int value, const Fraction& m)
{
    // Calls the (Fraction m * int value) operator*
    return m * value;
}
```

Question number 1

What is the difference here? Both codes seem to work fine. One has static, the other one doesn't.

```
int gcd(int a, int b)
    {
        if (a == 0)
            return b;
        return gcd(b % a, a);
    }

    void reduce()
    {
        if (m_numerator != 0 && m_detonator != 0)
        {
            int division = gcd(m_numerator, m_detonator);
            m_numerator /= division;
            m_detonator /= division;
        }
    }
```

```
static int gcd(int a, int b)
    {
        if (a == 0)
            return b;
        return gcd(b % a, a);
    }

    void reduce()
    {
        if (m_numerator != 0 && m_detonator != 0)
        {
            int division = Fraction::gcd(m_numerator, m_detonator); // We can even ignore F
            m_numerator /= division;
            m_detonator /= division;
        }
    }
```

Question number 2

Shouldn't this function have const? Cause if the function does not change the value it should always be const. Also at the start 2 examples of getCents() fucntion have a const and the third one does not. This may confuse some people.

```
void print() const
{
  std::cout << m_numerator << '/' << m_detonator << '\n';
}
```

nascardriver
January 8, 2020 at 4:38 am · Reply

> Question number 1
`static int gcd` has internal linkage, it can only be accessed by the file its defined in.
`int gcd` has external linkage, it can be accessed from the entire program (via a forward declaration).
This is covered in lesson 6.6 and 6.7.

> Question number 2
You're right! I added more `const` to the lesson.

Bojan
December 11, 2019 at 8:45 am · Reply

For the 5'th example from the top, in the class "Cents" there is no + operator overload function that takes two integers as paramaters, but if in main I type, Cents c3 = 100 + 50;
there will be no error, and the value of 150 will be recorded to the private variable c3.m_cents. Even if I stated the following in main, Cents c4 = 20; it would still be recorded to c4.m_cents as a value of 20. How does the compiler know, that the value of 20 needs to be saved to m_cents, as there might be more than one int private member variable? The only thing that comes to mind is that Cents c4 = 20; calls a constructor function, and places 20 as paramter, so it would be the same as stating Cents c4(20);. Could you please elaborate on this? Thanks.

nascardriver
December 11, 2019 at 8:54 am · Reply

100 is an integer, 50 is an integer, you're doing integer + integer. The result is then converted to a Cents.

sito
December 8, 2019 at 11:47 am · Reply

hello! So i've been trying to do exercise 1b but I can't get it to work. I've compared my code to the the sulllution and examples in this chapter and I can't spot what's wrong. I've also tried to search for the explenation on the internet but no luck getting it to work. the error i'm getting is 1>C:\Users\Nibar Ahmed\Google Drive\Documents\c++ practis\overLoadingArithmeticOperatorsFriendFunctionsQ1\overLoadingArithmeticOperatorsFriendFunctionsQ1\overLoadingArithmeticOperatorsFriendFunctionsQ1.cpp(37,22): error C2676: binary '*': 'Fraction' does not define this operator or a conversion to a type acceptable to the predefined operator. Here is the code.

```
#include <iostream>
class Fraction {
private:
    int m_numerator;
    int m_denominator;
public:
    Fraction(int numerator=2, int denominator=10):
        m_numerator{ numerator },
        m_denominator{ denominator }
    {
}
    void print() {
        std::cout << m_numerator << '/' << m_denominator << '\n';

    }
    friend Fraction multiplyFraction(const Fraction& fraction1, const Fraction& fraction2);
    friend Fraction multiplyFractionInt(const Fraction& f1, const int value);

```

```
};
Fraction multiplyFraction(const Fraction& fraction1, const Fraction& fraction2) {
    return Fraction(fraction1.m_numerator * fraction2.m_numerator, fraction1.m_denominator
}
Fraction multiplyFractionInt(const Fraction& f1, const int value) {
    return Fraction(f1.m_numerator * value, f1.m_denominator);
}


int main()
{
    Fraction f1{ 2, 5};
    f1.print();
    Fraction f2{ 3, 8 };
    f2.print();
    Fraction f3 = f1 * f2;
    f3.print();
    Fraction f4 = f1 * 2;
    f4.print();
    Fraction f5 = f2*2;
    f5.print();
    Fraction f6 = Fraction(1, 2) * Fraction(2, 3) * Fraction(3, 4);
    f6.print();

    return 0;
}
```

nascardriver
December 9, 2019 at 3:47 am · Reply

You didn't overload any operators. You're never used `multiplyFraction` or `multiplyFractionInt`. Those functions should be called `operator*`.

Samira Ferdi
November 25, 2019 at 5:49 pm · Reply

Hi, Alex and Nascardriver!

Is a good idea to make gcd() and reduce() private? I think there is no any reason to use them publicly. So, if they are private member, then gcd() must be non-static method. But, what do you think?

I hope Alex would revise the solution number 1c, so this class works with cases: 0/0, n/0, 0/n, when n > 0. Because he said "Make sure all fractions are properly reduced", so, I think it should works with those cases.

Alex
November 25, 2019 at 10:50 pm · Reply

My 2c:
* No on GCD, since it provides useful utility even if you don't have a Fraction object. There's no harm in letting the public use it, since it doesn't modify the object state.
* For reduce(), it depends on whether you think anybody will ever have a reason to call reduce() explicitly. They don't in the example, so in this use case, it could be made private.

Fractions with a zero denominator are invalid. As noted in a prior lesson, these examples typically omit error handling because error handling adds clutter to the concepts being taught.

I updated the example to handle 0 numerators though. Thanks for the suggestion.

Samira Ferdi
November 26, 2019 at 4:25 pm · Reply

Hi, Alex. Thanks for your thought! I appreciate that!

Samira Ferdi
November 24, 2019 at 6:14 pm · Reply

Hi, Alex and Nascardriver!

In this lesson, I just realize that we can do this

```
1 std::cout << int{ 5 } + int{ 7 };
```

and it works fine! Interesting! Now, it's clear to me to think a class just a data type!

Samira Ferdi
November 24, 2019 at 5:54 pm · Reply

Hi, Alex and Nascardriver!

Do you prefer this?

```
1 Cents operator+(const Cents &cent1, const Cents &cent2)
2 {
3     return Cents{ cent1.getCents() + cent2.getCents() };
4 }
```

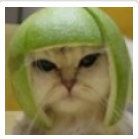
or this

```
1 Cents operator+(const Cents &cent1, const Cents &cent2)
2 {
3     return Cents{ cent1.m_cents + cent2.m_cents };
4 }
```

in our operator overloading, we're just take two arguments. Why I can do this?

```
1 Cents sumCents = cent1 + cent2 + cent2 + cent1;
```

In normal function, we cannot do this, right? Are there any explanations?



Alex
November 25, 2019 at 10:18 pm · Reply

I'd prefer the getCents() version as a non-friend.

`Cents sumCents = cent1 + cent2 + cent2 + cent1;` evaluates as `Cents sumCents = (((cent1 + cent2) + cent2) + cent1)`

With the result of each call to operator+ becoming the left hand operand of the next call to operator+.

You can do this with normal functions: See lesson 8.8 (the hidden this pointer), the example that contains `calc.add(5).sub(3).mult(4);`

« Older Comments 1 2 3 4