21.2 — Overloading the arithmetic operators using friend functions

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 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 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.\n";

	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 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 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.\n";

	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 is inside the class
	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 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.\n";

	return 0;
}
```

This is fine for overloaded operators with trivial implementations.

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 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 c1.m_cents + value;
}

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

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

	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:

```
#include <iostream>

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() const { return m_min; }
	int getMax() const { 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 { 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 { 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+ evaluates from left to right, so m1 + m2 evaluates 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 used to initialize 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 may or may not be worth doing this.

Quiz time

Question #1

a) Write a class named Fraction that has an 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

b) 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

c) Why does the program continue to work correctly if we remove the operators for integer multiplication from the previous solution?

```
// We can remove these operators, and the program continues to work
Fraction operator*(const Fraction& f1, int value);
Fraction operator*(int value, const Fraction& f1);
```

Show Solution

d) If we remove the `const` from the `Fraction * Fraction` operator, the following line from the `main` function no longer works. Why?

```
// The non-const multiplication operator looks like this
Fraction operator*(Fraction& f1, Fraction& f2)

// This doesn't work anymore
Fraction f6{ Fraction{1, 2} * Fraction{2, 3} * Fraction{3, 4} };
```

Show Solution

e) 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.

std::gcd was added to the standard library in C++17 (in the <numeric> header).

If you're on an older compiler, you can use this function to find the GCD:

```
#include <cmath> // for std::abs

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

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
0/1
```

Show Solution