21.5 — Overloading operators using member functions

In lesson 21.2 -- Overloading the arithmetic operators using friend functions, you learned how to overload the arithmetic operators using friend functions. You also learned you can overload operators as normal functions. Many operators can be overloaded in a different way: as a member function.

Overloading operators using a member function is very similar to overloading operators using a friend function. When overloading an operator using a member function:

- The overloaded operator must be added as a member function of the left operand.
- The left operand becomes the implicit *this object
- All other operands become function parameters.

As a reminder, here’s how we overloaded operator+ using a friend function:

```
#include <iostream>

class Cents
{
private:
    int m_cents {};

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

    // Overload Cents + int
    friend Cents operator+(const Cents& cents, int value);

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

// note: this function is not a member function!
Cents operator+(const Cents& cents, int value)
{
    return Cents(cents.m_cents + value);
}

int main()
{
        const Cents cents1 { 6 };
        const Cents cents2 { cents1 + 2 };
        std::cout << "I have " << cents2.getCents() << " cents.\n";

        return 0;
}
```

Converting a friend overloaded operator to a member overloaded operator is easy:

1. The overloaded operator is defined as a member instead of a friend (Cents::operator+ instead of friend operator+)
2. The left parameter is removed, because that parameter now becomes the implicit *this object.
3. Inside the function body, all references to the left parameter can be removed (e.g. cents.m_cents becomes m_cents, which implicitly references the *this object).

Now, the same operator overloaded using the member function method:

```
#include <iostream>

class Cents
{
private:
    int m_cents {};

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

    // Overload Cents + int
    Cents operator+(int value) const;

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

// note: this function is a member function!
// the cents parameter in the friend version is now the implicit *this parameter
Cents Cents::operator+ (int value) const
{
    return Cents { m_cents + value };
}

int main()
{
        const Cents cents1 { 6 };
        const Cents cents2 { cents1 + 2 };
        std::cout << "I have " << cents2.getCents() << " cents.\n";

        return 0;
}
```

Note that the usage of the operator does not change (in both cases, `cents1 + 2`), we've simply defined the function differently. Our two-parameter friend function becomes a one-parameter member function, with the leftmost parameter in the friend version (cents) becoming the implicit *this parameter in the member function version.

Let's take a closer look at how the expression `cents1 + 2` evaluates.

In the friend function version, the expression `cents1 + 2` becomes function call operator+(cents1, 2). Note that there are two function parameters. This is straightforward.

In the member function version, the expression `cents1 + 2` becomes function call `cents1.operator+(2)`. Note that there is now only one explicit function parameter, and cents1 has become an object prefix. However, in lesson 15.1 -- The hidden "this" pointer and member function chaining, we mentioned that the compiler implicitly converts an object prefix into a hidden leftmost parameter named *this. So in actuality, `cents1.operator+(2)` becomes `operator+(&cents1, 2)`, which is almost identical to the friend version.

Both cases produce the same result, just in slightly different ways.

So if we can overload an operator as a friend or a member, which should we use? In order to answer that question, there's a few more things you'll need to know.

Not everything can be overloaded as a friend function

The assignment (=), subscript ([]), function call (()), and member selection (->) operators must be overloaded as member functions, because the language requires them to be.

Not everything can be overloaded as a member function

In lesson 21.4 -- Overloading the I/O operators, we overloaded operator<< for our Point class using the friend function method. Here's a reminder of how we did that:

```
#include <iostream>

class Point
{
private:
    double m_x {};
    double m_y {};
    double m_z {};

public:
    Point(double x=0.0, double y=0.0, double z=0.0)
        : m_x { x }, m_y { y }, m_z { z }
    {
    }

    friend std::ostream& operator<< (std::ostream& out, const Point& point);
};

std::ostream& operator<< (std::ostream& out, const Point& point)
{
    // Since operator<< is a friend of the Point class, we can access Point's members
directly.
    out << "Point(" << point.m_x << ", " << point.m_y << ", " << point.m_z << ")";

    return out;
}

int main()
{
    Point point1 { 2.0, 3.0, 4.0 };

    std::cout << point1;

    return 0;
}
```

However, we are not able to overload operator<< as a member function. Why not? Because the overloaded operator must be added as a member of the left operand. In this case, the left operand is an object of type std::ostream. std::ostream is fixed as part of the standard library. We can't modify the class declaration to add the overload as a member function of std::ostream.

This necessitates that operator<< be overloaded as a normal function (preferred) or a friend.

Similarly, although we can overload operator+(Cents, int) as a member function (as we did above), we can't overload operator+(int, Cents) as a member function, because int isn't a class we can add members to.

Typically, we won't be able to use a member overload if the left operand is either not a class (e.g. int), or it is a class that we can't modify (e.g. std::ostream).

When to use a normal, friend, or member function overload

In most cases, the language leaves it up to you to determine whether you want to use the normal/friend or member function version of the overload. However, one of the two is usually a better choice than the other.

When dealing with binary operators that don't modify the left operand (e.g. operator+), the normal or friend function version is typically preferred, because it works for all parameter types (even when the left operand isn't a class object, or is a class that is not modifiable). The normal or friend function version has the added benefit of "symmetry", as all operands become explicit parameters (instead of the left operand becoming *this and the right operand becoming an explicit parameter).

When dealing with binary operators that do modify the left operand (e.g. operator+=), the member function version is typically preferred. In these cases, the leftmost operand will always be a class type, and having the object being modified become the one pointed to by *this is natural. Because the rightmost operand becomes an explicit parameter, there's no confusion over who is getting modified and who is getting evaluated.

Unary operators are usually overloaded as member functions as well, since the member version has no parameters.

The following rules of thumb can help you determine which form is best for a given situation:

- If you're overloading assignment (=), subscript ([]), function call (()), or member selection (->), do so as a member function.
- If you're overloading a unary operator, do so as a member function.
- If you're overloading a binary operator that does not modify its left operand (e.g. operator+), do so as a normal function (preferred) or friend function.

- If you’re overloading a binary operator that modifies its left operand, but you can’t add members to the class definition of the left operand (e.g. operator<<, which has a left operand of type ostream), do so as a normal function (preferred) or friend function.
- If you’re overloading a binary operator that modifies its left operand (e.g. operator+=), and you can modify the definition of the left operand, do so as a member function.