


## 21.14 — Overloading operators and function templates

 [learncpp.com/cpp-tutorial/overloading-operators-and-function-templates/](http://learncpp.com/cpp-tutorial/overloading-operators-and-function-templates/)

In lesson [11.7 -- Function template instantiation](#), we discussed how the compiler will use function templates to instantiate functions, which are then compiled. We also noted that these functions may not compile, if the code in the function template tries to perform some operation that the actual type doesn't support (such as adding integer value `1` to a `std::string`).

In this lesson, we'll take a look at a few examples where our instantiated functions won't compile because our actual class types don't support those operators, and show how we can define those operators so that the instantiated functions will then compile.

Operators, function calls, and function templates

First, let's create a simple class:

```
class Cents
{
private:
    int m_cents{};
public:
    Cents(int cents)
        : m_cents { cents }
    {
    }

    friend std::ostream& operator<< (std::ostream& ostr, const Cents& c)
    {
        ostr << c.m_cents;
        return ostr;
    }
};
```

and define a `max` function template:

```
template <typename T>
const T& max(const T& x, const T& y)
{
    return (x < y) ? y : x;
}
```

Now, let's see what happens when we try to call `max()` with object of type `Cents`:

```

#include <iostream>

class Cents
{
private:
    int m_cents{};
public:
    Cents(int cents)
        : m_cents { cents }
    {
    }

    friend std::ostream& operator<< (std::ostream& ostr, const Cents& c)
    {
        ostr << c.m_cents;
        return ostr;
    }
};

template <typename T>
const T& max(const T& x, const T& y)
{
    return (x < y) ? y : x;
}

int main()
{
    Cents nickel{ 5 };
    Cents dime{ 10 };

    Cents bigger { max(nickel, dime) };
    std::cout << bigger << " is bigger\n";

    return 0;
}

```

C++ will create a template instance for `max()` that looks like this:

```

template <>
const Cents& max(const Cents& x, const Cents& y)
{
    return (x < y) ? y : x;
}

```

And then it will try to compile this function. See the problem here? C++ has no idea how to evaluate `x < y` when `x` and `y` are of type `Cents`! Consequently, this will produce a compile error.

To get around this problem, simply overload `operator<` for any class we wish to use `max` with:

```

#include <iostream>

class Cents
{
private:
    int m_cents {};
public:
    Cents(int cents)
        : m_cents { cents }
    {
    }

    friend bool operator< (const Cents& c1, const Cents& c2)
    {
        return (c1.m_cents < c2.m_cents);
    }

    friend std::ostream& operator<< (std::ostream& ostr, const Cents& c)
    {
        ostr << c.m_cents;
        return ostr;
    }
};

template <typename T>
const T& max(const T& x, const T& y)
{
    return (x < y) ? y : x;
}

int main()
{
    Cents nickel{ 5 };
    Cents dime { 10 };

    Cents bigger { max(nickel, dime) };
    std::cout << bigger << " is bigger\n";

    return 0;
}

```

This works as expected, and prints:

```
10 is bigger
```

Another example

Let's do one more example of a function template not working because of missing overloaded operators.

The following function template will calculate the average of a number of objects in an array:

```

#include <iostream>

template <typename T>
T average(const T* myArray, int numValues)
{
    T sum { 0 };
    for (int count { 0 }; count < numValues; ++count)
        sum += myArray[count];

    sum /= numValues;
    return sum;
}

int main()
{
    int intArray[] { 5, 3, 2, 1, 4 };
    std::cout << average(intArray, 5) << '\n';

    double doubleArray[] { 3.12, 3.45, 9.23, 6.34 };
    std::cout << average(doubleArray, 4) << '\n';

    return 0;
}

```

This produces the values:

```

3
5.535

```

As you can see, it works great for built-in types!

Now let's see what happens when we call this function on our **Cents** class:

```

#include <iostream>

template <typename T>
T average(const T* myArray, int numValues)
{
    T sum { 0 };
    for (int count { 0 }; count < numValues; ++count)
        sum += myArray[count];

    sum /= numValues;
    return sum;
}

class Cents
{
private:
    int m_cents {};
public:
    Cents(int cents)
        : m_cents { cents }
    {
    }
};

int main()
{
    Cents centsArray[] { Cents { 5 }, Cents { 10 }, Cents { 15 }, Cents { 14 } };
    std::cout << average(centsArray, 4) << '\n';

    return 0;
}

```

The compiler goes berserk and produces a ton of error messages! The first error message will be something like this:

```
error C2679: binary << : no operator found which takes a right-hand operand of type Cents (or there is no acceptable conversion)
```

Remember that `average()` returns a `Cents` object, and we are trying to stream that object to `std::cout` using `operator<<`. However, we haven't defined the `operator<<` for our `Cents` class yet. Let's do that:

```

#include <iostream>

template <typename T>
T average(const T* myArray, int numValues)
{
    T sum { 0 };
    for (int count { 0 }; count < numValues; ++count)
        sum += myArray[count];

    sum /= numValues;
    return sum;
}

class Cents
{
private:
    int m_cents {};
public:
    Cents(int cents)
        : m_cents { cents }
    {
    }

    friend std::ostream& operator<< (std::ostream& out, const Cents& cents)
    {
        out << cents.m_cents << " cents ";
        return out;
    }
};

int main()
{
    Cents centsArray[] { Cents { 5 }, Cents { 10 }, Cents { 15 }, Cents { 14 } };
    std::cout << average(centsArray, 4) << '\n';

    return 0;
}

```

If we compile again, we will get another error:

error C2676: binary += : Cents does not define this operator or a conversion to a type acceptable to the predefined operator

This error is actually being caused by the function template instance created when we call `average(const Cents*, int)`. Remember that when we call a templated function, the compiler “stencils” out a copy of the function where the template type parameters (the placeholder types) have been replaced with the actual types in the function call. Here is the function template instance for `average()` when `T` is a `Cents` object:

```

template <>
Cents average(const Cents* myArray, int numValues)
{
    Cents sum { 0 };
    for (int count { 0 }; count < numValues; ++count)
        sum += myArray[count];

    sum /= numValues;
    return sum;
}

```

The reason we are getting an error message is because of the following line:

```
sum += myArray[count];
```

In this case, `sum` is a `Cents` object, but we have not defined `operator+=` for `Cents` objects! We will need to define this function in order for `average()` to be able to work with `Cents`. Looking forward, we can see that `average()` also uses the `operator/=`, so we will go ahead and define that as well:

```

#include <iostream>

template <typename T>
T average(const T* myArray, int numValues)
{
    T sum { 0 };
    for (int count { 0 }; count < numValues; ++count)
        sum += myArray[count];

    sum /= numValues;
    return sum;
}

class Cents
{
private:
    int m_cents {};
public:
    Cents(int cents)
        : m_cents { cents }
    {
    }

    friend std::ostream& operator<< (std::ostream& out, const Cents& cents)
    {
        out << cents.m_cents << " cents ";
        return out;
    }

    Cents& operator+= (const Cents &cents)
    {
        m_cents += cents.m_cents;
        return *this;
    }

    Cents& operator/= (int x)
    {
        m_cents /= x;
        return *this;
    }
};

int main()
{
    Cents centsArray[] { Cents { 5 }, Cents { 10 }, Cents { 15 }, Cents { 14 } };
    std::cout << average(centsArray, 4) << '\n';

    return 0;
}

```

Finally, our code will compile and run! Here is the result:



11 cents

Note that we didn't have to modify `average()` at all to make it work with objects of type `Cents`. We simply had to define the operators used to implement `average()` for the `Cents` class, and the compiler took care of the rest!