# 13.2 — Function template instances

BY ALEX ON APRIL 29TH, 2008 | LAST MODIFIED BY ALEX ON NOVEMBER 5TH, 2017

It's worth taking a brief look at how template functions are implemented in C++, because future lessons will build off of some of these concepts. It turns out that C++ does not compile the template function directly. Instead, at compile time, when the compiler encounters a call to a template function, it replicates the template function and replaces the template type parameters with actual types. The function with actual types is called a **function template instance**.

Let's take a look at an example of this process. First, we have a templated function:

```
template <typename T> // this is the template parameter declaration
const T& max(const T& x, const T& y)
{
    return (x > y) ? x : y;
}
```

When compiling your program, the compiler encounters a call to the templated function:

```
int i = max(3, 7); // calls max(int, int)
```

The compiler says, "oh, we want to call max(int, int)". The compiler replicates the function template and creates the template instance max(int, int):

```
const int& max(const int &x, const int &y)
{
    return (x > y) ? x : y;
}
```

This is now a "normal function" that can be compiled into machine language.

Now, let's say later in your code, you called max() again using a different type:

```
double d = max(6.34, 18.523); // calls max(double, double)
```

C++ automatically creates a template instance for max(double, double):

```
const double& max(const double &x, const double &y)
{
    return (x > y) ? x : y;
}
```

and then compiles it.

The compiler is smart enough to know it only needs to create one template instance per set of unique type parameters (per file). It's also worth noting that if you create a template function but do not call it, no template instances will be created.

**Operators, function calls, and function templates**

Template functions will work with both built-in types (e.g. char, int, double, etc…) and classes, with one caveat. When the compiler compiles the template instance, it compiles it just like a normal function. In a normal function, any operators or function calls that you use with your types must be defined, or you will get a compiler error. Similarly, any operators or function calls in your template function must be defined for any types the function template is instantiated for. Let's take a look at this in more detail.

First, we'll create a simple class:

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

```
};
```

Now, let's see what happens when we try to call our templated max() function with the Cents class:

```
template <typename T> // this is the template parameter declaration
const T& max(const T& x, const T& y)
{
    return (x > y) ? x : y;
}

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

int main()
{
    Cents nickle(5);
    Cents dime(10);

    Cents bigger = max(nickle, dime);

    return 0;
}
```

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

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

And then it will try to compile this function. See the problem here? C++ has no idea how to evaluate `x > y`! Consequently, this will produce a fairly-tame looking compile error, like this:

```
1>c:\consoleapplication1\main.cpp(4): error C2676: binary '>': 'const Cents' does not define this o
1>  c:\consoleapplication1\main.cpp(23): note: see reference to function template instantiation 'co
1>          with
1>          [
1>              T=Cents
1>          ]
```

The top error message points out the fact that there is no overloaded operator > for the Cents class. The bottom error points out the templated function call that spawned the error, along with the type of the templated parameter.

To get around this problem, simply overload the > operator for any class we wish to use max() with:

```
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);
    }
};
```

Now C++ will know how to compare x > y when x and y are objects of the Cents class! As a result, our max() function will now work with two objects of type Cents.

**Another example**

Let's do one more example of a function template. The following function template will calculate the average of a number of objects in an array:

```
template <class T>
T average(T *array, int length)
{
    T sum = 0;
    for (int count=0; count < length; ++count)
        sum += array[count];

    sum /= length;
    return sum;
}
```

Now let's see it in action:

```
#include <iostream>

template <class T>
T average(T *array, int length)
{
    T sum = 0;
    for (int count=0; count < length; ++count)
        sum += array[count];

    sum /= length;
    return sum;
}

int main()
{
    int array1[] = { 5, 3, 2, 1, 4 };
    std::cout << average(array1, 5) << '\n';

    double array2[] = { 3.12, 3.45, 9.23, 6.34 };
    std::cout << average(array2, 4) << '\n';

    return 0;
}
```

This produces the values:

```
3
5.535
```

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

It is worth noting that because our return type is the same templated type as our array elements, doing an integer average will produce an integer result (dropping any fractional value). This is similar to how doing an integer division will produce an integer result. It's not wrong that we've defined things to work that way, but it may be unexpected, so a good comment to users of the class wouldn't be amiss here.

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

```
#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);
    }
};

template <class T>
T average(T *array, int length)
{
    T sum = 0;
    for (int count=0; count < length; ++count)
        sum += array[count];

    sum /= length;
    return sum;
}

int main()
{
    Cents array3[] = { Cents(5), Cents(10), Cents(15), Cents(14) };
    std::cout << average(array3, 4) << '\n';

    return 0;
}
```

The compiler goes berserk and produces a ton of error messages!

```
c:\consoleapplication1\main.cpp(55): error C2679: binary '<<': no operator found which takes a righ
1>  c:\program files (x86)\microsoft visual studio 14.0\vc\include\ostream(497): note: could be 'st
1>  c:\program files (x86)\microsoft visual studio 14.0\vc\include\ostream(477): note: or       'st
1>  c:\program files (x86)\microsoft visual studio 14.0\vc\include\ostream(457): note: or       'st
1>  c:\program files (x86)\microsoft visual studio 14.0\vc\include\ostream(437): note: or       'st
1>  c:\program files (x86)\microsoft visual studio 14.0\vc\include\ostream(417): note: or       'st
1>  c:\program files (x86)\microsoft visual studio 14.0\vc\include\ostream(396): note: or       'st
1>  c:\program files (x86)\microsoft visual studio 14.0\vc\include\ostream(376): note: or       'st
1>  c:\program files (x86)\microsoft visual studio 14.0\vc\include\ostream(355): note: or       'st
1>  c:\program files (x86)\microsoft visual studio 14.0\vc\include\ostream(335): note: or       'st
1>  c:\program files (x86)\microsoft visual studio 14.0\vc\include\ostream(315): note: or       'st
1>  c:\program files (x86)\microsoft visual studio 14.0\vc\include\ostream(290): note: or       'st
1>  c:\program files (x86)\microsoft visual studio 14.0\vc\include\ostream(270): note: or       'st
1>  c:\program files (x86)\microsoft visual studio 14.0\vc\include\ostream(236): note: or       'st
1>  c:\program files (x86)\microsoft visual studio 14.0\vc\include\ostream(216): note: or       'st
1>  c:\program files (x86)\microsoft visual studio 14.0\vc\include\ostream(209): note: or       'st
1>  c:\program files (x86)\microsoft visual studio 14.0\vc\include\ostream(202): note: or       'st
1>  c:\program files (x86)\microsoft visual studio 14.0\vc\include\ostream(196): note: or       'st
1>  c:\program files (x86)\microsoft visual studio 14.0\vc\include\ostream(694): note: or       'st
1>  c:\program files (x86)\microsoft visual studio 14.0\vc\include\ostream(741): note: or       'st
1>  c:\program files (x86)\microsoft visual studio 14.0\vc\include\ostream(779): note: or       'st
1>  c:\program files (x86)\microsoft visual studio 14.0\vc\include\ostream(826): note: or       'st
1>  c:\program files (x86)\microsoft visual studio 14.0\vc\include\ostream(952): note: or       'st
1>  c:\program files (x86)\microsoft visual studio 14.0\vc\include\ostream(959): note: or       'st
1>  c:\program files (x86)\microsoft visual studio 14.0\vc\include\ostream(966): note: or       'st
1>  c:\program files (x86)\microsoft visual studio 14.0\vc\include\ostream(973): note: or       'st
1>  c:\program files (x86)\microsoft visual studio 14.0\vc\include\ostream(983): note: or       'st
1>          with
```

```
1>          [
1>              T=Cents,
1>              _Ty=Cents
1>          ]
1>  c:\program files (x86)\microsoft visual studio 14.0\vc\include\ostream(1021): note: or       '
1>  c:\consoleapplication1\main.cpp(55): note: while trying to match the argument list '(std::ostr
```

Remember what I said about crazy error messages? We hit the motherload! Despite looking intimidating, these are actually quite straightforward. The first line is telling you that it couldn't find an overloaded operator<< for the Cents class. All of the lines in the middle are all of the different functions it tried to match with but failed. The last error points out the function call that spawned this wall of errors.

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

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

If we compile again, we will get another error:

```
c:test.cpp(14) : error C2676: binary '+=' : 'Cents' does not define this operator or a conversion
```

This error is actually being caused by the function template instance created when we call average(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 <class T>
Cents average(Cents *array, int length)
{
    Cents sum = 0;
    for (int count=0; count < length; ++count)
        sum += array[count];

    sum /= length;
    return sum;
}
```

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

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

In this case, sum is a Cents object, but we have not defined the += 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:

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

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

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

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

```
11 cents
```

If this seems like a lot of work, that's really only because our Cents class was so bare-bones to start. The key point here is actually that we didn't have to modify average() at all to make it work with objects of type Cents (or any other type). We simply had to define the operators used to implement average() for the Cents class, and the compiler took care of the rest!



**13.3 -- Template classes**


**Index**


**13.1 -- Function templates**


## Share this: