13.1 — Function templates

BY ALEX ON APRIL 18TH, 2008 | LAST MODIFIED BY ALEX ON DECEMBER 6TH, 2016

The need for function templates

In previous chapters, you've learned how to write functions and classes that help make programs easier to write, safer, and more maintainable. While functions and classes are powerful and flexible tools for effective programming, in certain cases they can also be somewhat limiting because of C++'s requirement that you specify the type of all parameters.

For example, let's say you wanted to write a function to calculate the maximum of two numbers. You might do so like this:

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

This function would work great -- for integers. What happens later when you realize your max() function needs to work with doubles? Traditionally, the answer would be to overload the max() function and create a new version that works with doubles:

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

Note that the code for the implementation of the double version of maximum() is exactly the same as for the int version of max()! In fact, this implementation would work for all sorts of different types: chars, ints, doubles, and if you've overloaded the > operator, even classes! However, because C++ requires you to make your variables specific types, you're stuck writing one function for each type you wish to use.

Having to specify different "flavors" of the same function where the only thing that changes is the type of the parameters can become a maintenance headache and time-waster, and it also violates the general programming guideline that duplicate code should be minimized as much as possible. Wouldn't it be nice if we could write one version of max() that was able to work with parameters of ANY type?

Welcome to the world of templates.

What is a function template?

If you were to look up the word "template" in the dictionary, you'd find a definition that was similar to the following: "a template is a model that serves as a pattern for creating similar objects". One type of template that is very easy to understand is that of a stencil. A stencil is an object (e.g. a piece of cardboard) with a shape cut out of it (eg. the letter J). By placing the stencil on top of another object, then spraying paint through the hole, you can very quickly produce stenciled patterns in many different colors! Note that you only need to create a given stencil once -- you can then use it as many times as you like, to create stenciled patterns in whatever color(s) you like. Even better, you don't have to decide the color of the stenciled pattern you want to create until you decide to actually use the stencil.

In C++, **function templates** are functions that serve as a pattern for creating other similar functions. The basic idea behind function templates is to create a function without having to specify the exact type(s) of some or all of the variables. Instead, we define the function using placeholder types, called **template type parameters**. Once we have created a function using these placeholder types, we have effectively created a "function stencil".

When you call a template function, the compiler "stencils" out a copy of the template, replacing the placeholder types with the actual variable types from the parameters in your function call! Using this methodology, the compiler can create multiple "flavors" of a function from one template! We'll take a look at this process in more detail in the next lesson.

Creating function templates in C++

At this point, you're probably wondering how to actually create function templates in C++. It turns out, it's not all that difficult.

Let's take a look at the int version of max() again:

```
int max(int x, int y)
```

```
{
    return (x > y) ? x : y;
}
```

Note that there are 3 places where specific types are used: parameters x, y, and the return value all specify that they must be integers. To create a function template, we're going to replace these specific types with placeholder types. In this case, because we have only one type that needs replacing (int), we only need one template type parameter.

You can name your placeholder types almost anything you want, so long as it's not a reserved word. However, in C++, it's customary to name your template types the letter T (short for "Type").

Here's our new function with a placeholder type:

```cpp
T max(T x, T y)
{
    return (x > y) ? x : y;
}
```

This is a good start -- however, it won't compile because the compiler doesn't know what "T" is!

In order to make this work, we need to tell the compiler two things: First, that this is a template definition, and second, that T is a placeholder type. We can do both of those things in one line, using what is called a **template parameter declaration**:

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

Believe it or not, that's all we need. This will compile!

Now, let's take a slightly closer look at the template parameter declaration. We start with the keyword *template* -- this tells the compiler that what follows is going to be a list of template parameters. We place all of our parameters inside angled brackets (<>). To create a template type parameter, use either the keyword *typename* or *class*. There is no difference between the two keywords in this context, so which you use is up to you. Note that if you use the class keyword, the type passed in does not actually have to be a class (it can be a fundamental variable, pointer, or anything else that matches). Then you name your type (usually "T").

If the template function uses multiple template type parameter, they can be separated by commas:

```cpp
template <typename T1, typename T2>
// template function here
```

For classes using more than one type, it's common to see them named "T1" and "T2", or other single capital letter names, such as "S".

One final note: Because the function argument passed in for type T could be a class type, and it's generally not a good idea to pass classes by value, it would be better to make the parameters and return types of our templated function const references:

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

Using function templates

Using a function template is extremely straightforward -- you can use it just like any other function. Here's a full program using our template function:

```cpp
#include <iostream>

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

int main()
```

```
{
    int i = max(3, 7); // returns 7
    std::cout << i << '\n';

    double d = max(6.34, 18.523); // returns 18.523
    std::cout << d << '\n';

    char ch = max('a', '6'); // returns 'a'
    std::cout << ch << '\n';

    return 0;
}
```

This will print:

```
7
18.523
a
```

Note that all three of these calls to max() have parameters of different types! Because we've called the function with 3 different types, the compiler will use the template definition to create 3 different versions of this function: one with int parameters, one with double parameters, and one with char parameters.

Summary

As you can see, template functions can save a lot of time, because you only need to write one function, and it will work with many different types. Once you get used to writing function templates, you'll find they actually don't take any longer to write than functions with actual types. Template functions reduce code maintenance, because duplicate code is reduced significantly. And finally, template functions can be safer, because there is no need to copy functions and change types by hand whenever you need the function to work with a new type!

Template functions do have a few drawbacks, and we would be remiss not to mention them. First, some older compilers do not have very good template support. However, this downside is no longer as much of a problem as it used to be. Second, template functions often produce crazy-looking error messages that are much harder to decipher than those of regular functions (we'll see an example of this in the next lesson). Third, template functions can increase your compile time and code size, as a single template might be "realized" and recompiled in many files (there are ways to work around this one).

However, these drawbacks are fairly minor compared with the power and flexibility templates bring to your programming toolkit!

Note: The standard library already comes with a templated max() function (in the algorithm header), so you don't have to write your own (unless you want to). If you do write your own, note the potential for naming conflicts if you use the statement "using namespace std;", as the compiler will be unable to tell whether you want your version of max() or std::max().

In the rest of this chapter, we'll continue to explore the topic of templates.

13.2 -- Function template instances

Index

12.x -- Chapter 12 comprehensive quiz

Share this:

Facebook Twitter Google Pinterest