11.6 — Function templates

ALEX DECEMBER 28, 2023

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) ? y : x;
    // Note: we use < instead of > because std::max uses <
}
```

While the caller can pass different values into the function, the type of the parameters is fixed, so the caller can only pass in `int` values. That means this function really only works well for integers (and types that can be promoted to `int`).

So what happens later when you want to find the max of two `double` values? Because C++ requires us to specify the type of all function parameters, the solution is to create a new overloaded version of `max` with parameters of type `double`:

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

Note that the code for the implementation of the double version of `max` is exactly the same as for the int version of `max`! In fact, this implementation works for many different types: including `int`, `double`, `long`, `long double`, and even new types that you've created yourself (which we'll cover how to do in future lessons).

Having to create overloaded functions with the same implementation for each set of parameter types we want to support is a maintenance headache, a recipe for errors, and a clear violation of the DRY (don't repeat yourself) principle. There's a less-obvious challenge here as well: a programmer who wishes to use the `max` function may wish to call it with an argument type that the author of the `max` did not anticipate (and thus did not write an overloaded function for).

What we are really missing is some way to write a single version of `max` that can work with arguments of any type (even types that may not have been anticipated when the code for `max` was written). Normal functions are simply not up to the task here. Fortunately, C++ supports another feature that was designed specifically to solve this kind of problem.

Welcome to the world of C++ templates.

Introduction to C++ templates

In C++, the template system was designed to simplify the process of creating functions (or classes) that are able to work with different data types.

Instead of manually creating a bunch of mostly-identical functions or classes (one for each set of different types), we instead create a single template. Just like a normal definition, a **template** describes what a function or class looks like. Unlike a normal definition (where all types must be specified), in a template we can use one or more placeholder types. A placeholder type represents some type that is not known at the time the template is written, but that will be provided later.

Once a template is defined, the compiler can use the template to generate as many overloaded functions (or classes) as needed, each using different actual types!

The end result is the same -- we end up with a bunch of mostly-identical functions or classes (one for each set of different types). But we only have to create and maintain a single template, and the compiler does all the hard work for us.

Key insight

The compiler can use a single template to generate a family of related functions or classes, each using a different set of types.

As an aside...

Because the concept behind templates can be hard to describe in words, let's try an analogy.

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 a thin piece of material (such as a piece of cardboard or plastic) with a shape cut out of it (e.g. a happy face). By placing the stencil on top of another object, then spraying paint through the hole, you can very quickly replicate the cut-out shape. The stencil itself only needs to be created once, and then it can be reused as many times as desired, to create the cut out shape in as many different colors as you like. Even better, the color of a shape made with the stencil doesn't have to be determined until the stencil is actually used.

A template is essentially a stencil for creating functions or classes. We create the template (our stencil) once, and then we can use it as many times as needed, to stencil out a function or class for a specific set of actual types. Those actual types don't need to be determined until the template is actually used.

Because the actual types aren't determined until the template is used in a program (not when the template is written), the author of the template doesn't have to try to anticipate all of the actual types that might be used. This means template code can be used with types that didn't even exist when the template was written! We'll see how this comes in handy later, when we start exploring the C++ standard library, which is absolutely full of template code!

Key insight

Templates can work with types that didn't even exist when the template was written. This helps make template code both flexible and future proof!

In the rest of this lesson, we'll introduce and explore how to create templates for functions, and describe how they work in more detail. We'll save discussion of class templates until after we've covered what classes are.

Function templates

A **function template** is a function-like definition that is used to generate one or more overloaded functions, each with a different set of actual types. This is what will allow us to create functions that can work with many different types.

When we create our function template, we use placeholder types (also called **type template parameters**, or informally **template types**) for

any parameter types, return types, or types used in the function body that we want to be specified later.

For advanced readers

C++ supports 3 different kinds of template parameters:

- Type template parameters (where the template parameter represents a type).
- Non-type template parameters (where the template parameter represents a constexpr value).
- Template template parameters (where the template parameter represents a template).

Type template parameters are by far the most common, so we'll be focused on those. We'll cover non-type template parameters in the chapter on arrays.

Function templates are something that is best taught by example, so let's convert our normal `max(int, int)` function from the example above into a function template. It's surprisingly easy, and we'll explain what's happening along the way.

Creating a templated max function

Here's the int version of `max` again:

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

Note that we use type `int` three times in this function: once for parameter `x`, once for parameter `y`, and once for the return type of the function.

To create a function template, we're going to do two things. First, we're going to replace our specific types with type template parameters. In this case, because we have only one type that needs replacing (`int`), we only need one type template parameter (which we'll call `T`):

Here's our new function that uses a single template type:

```
T max(T x, T y) // won't compile because we haven't defined T
{
    return (x < y) ? y : x;
}
```

This is a good start -- however, it won't compile because the compiler doesn't know what `T` is! And this is still a normal function, not a function template.

Second, we're going to tell the compiler that this is a function template, and that `T` is a type template parameter that is a placeholder for any type. This is done using what is called a **template parameter declaration**. The scope of a template parameter declaration is limited to the function template (or class template) that follows. Therefore, each function template (or class) needs its own template parameter declaration.

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

In our template parameter declaration, we start with the keyword `template`, which tells the compiler that we're creating a template. Next, we specify all of the template parameters that our template will use inside angled brackets (`<>`). For each type template parameter, we use the keyword `typename` or `class`, followed by the name of the type template parameter (e.g. `T`).

Related content

We discuss how to create function templates with multiple template types in lesson 11.8 -- Function templates with multiple template types (https://www.learncpp.com/cpp-tutorial/function-templates-with-multiple-template-types/).

As an aside...

There is no difference between the `typename` and `class` keywords in this context. You will often see people use the `class` keyword since it was introduced into the language earlier. However, we prefer the newer `typename` keyword, because it makes it clearer that the type template parameter can be replaced by any type (such as a fundamental type), not just class types.

Believe it or not, we're done! We have created a template version of our `max` function that can now accept arguments of different types.

Because this function template has one template type named `T`, we'll refer to it as `max<T>`. In the next lesson, we'll look at how we use our `max<T>` function template to generate one or more `max()` functions with parameters of different types.

Naming template parameters

Much like we often use a single letter for variable names used in trivial situations (e.g. `x`), it's a common convention to use a single capital letter (starting with `T`) when the meaning of that type is trivial or obvious. For example, in our `max` function template:

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

We don't need to give `T` a complex name, because it's obviously just a placeholder type for the values being compared.

Our function templates will generally use this naming convention.

If a type template parameter has a non-obvious usage or meaning, then a more descriptive name is warranted. There are two common conventions for such names:

- Starting with a capital letter (e.g. `Allocator`). The standard library uses this naming convention.
- Prefixed with a `T`, then starting with a capital letter (e.g. `TAllocator`). This makes it easier to see that the type is a type template parameter.

Which you choose is a matter of personal preference.

For advanced readers

As an example, the standard library has an overload of `std::max` that is declared like this:

```
template< class T, class Compare >
const T& max( const T& a, const T& b, Compare comp ); // ignore the & for now, we'll cover these in a future lesson
```

Because `a` and `b` are of type `T`, we know that we don't care what type `a` and `b` are -- they can be any type. Because `comp` has type `Compare`, we know that `comp` must be a type that meets the requirements for a `Compare`. We can consult technical documentation to determine what those particular requirements are.

Best practice

Use a single capital letter starting with `T` (e.g. `T`, `U`, `V`, etc...) to name type template parameters that are used in trivial or obvious ways.

If the type template parameter has a non-obvious usage or meaning, then a more descriptive name is warranted. (e.g. `Allocator` or `TAllocator`).

Next lesson
11.7 Function template instantiation

Back to table of contents

Previous lesson
11.5 Default arguments