

11.6 — Function templates

by [ALEX¹](#)

JANUARY 27, 2025

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

```
1 | int max(int x, int y)
2 | {
3 |     return (x < y) ? y : x;
4 |     // Note: we use < instead of > because std::max uses <
5 | }
```

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`:

```
1 | double max(double x, double y)
2 | {
3 |     return (x < y) ? y : x;
4 | }
```

COPY

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 definition 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 defined, but that will be provided later (when the template is used).

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 to create the rest 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 actual 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 function templates, 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. The initial function template that is used to generate other functions is called the **primary template**, and the functions generated from the primary template are called **instantiated functions**.

When we create a primary function template, we use **placeholder types** (technically called **type template parameters**, informally called **template types**) for any parameter types, return types, or types used in the function body that we want to be specified later, by the user of the template.

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 focus on those first. We'll also discuss non-type template parameters, which are seeing increased usage in modern C++.

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 `max()` function template

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

```
1 | int max(int x, int y)
2 | {
3 |     return (x < y) ? y : x;
4 | }
```

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 for `max()`, we're going to do two things. First, we're going to replace any actual types that we want to be specified later 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, where all occurrences of actual type `int` have been replaced with type template parameter `T`:

```
1 | T max(T x, T y) // won't compile because we haven't defined T
2 | {
3 |     return (x < y) ? y : x;
4 | }
```

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 template, and that `T` is a type template parameter that is a placeholder for any type. Both of these are done using a **template parameter declaration**, which defines any template parameters that will be subsequently used. **The scope of a template parameter declaration is strictly limited to the function template (or class template) that follows.** Therefore, each function template or class template needs its own template parameter declaration.

```
1 | template <typename T> // this is the template parameter declaration defining T as a
   | type template parameter
2 | T max(T x, T y) // this is the function template definition for max<T>
3 | {
4 |     return (x < y) ? y : x;
5 | }
```

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` (preferred) 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/) (<https://www.learncpp.com/cpp-tutorial/function-templates-with-multiple-template-types/>)².

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 accept arguments of different types.

In the next lesson, we'll look at how we use our `max` function template to generate one or more `max()` functions with parameters of different types and actually call those functions.

Naming template parameters

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

```

1 | template <typename T>
2 | T max(T x, T y)
3 | {
4 |     return (x < y) ? y : x;
5 | }

```

We don't need to give `T` a complex name, because it's obviously just a placeholder type for the values being compared, and `T` can be any type that can be compared (such as `int`, `double`, or `char`, but not `nullptr`).

Our function templates will generally use this naming convention.

If a type template parameter has a non-obvious usage or specific requirements that must be met, 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:

```

1 | template< class T, class Compare >
2 | 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` (whatever that is).

When a function template is instantiated, the compiler replaces the template parameters with the template arguments and then compiles the resulting instantiated function. Whether the function compiles depends on how the object(s) of each type are used within the function. Therefore, the requirements for a given template parameter are essentially implicitly defined.

Because it can be hard to infer requirements from how objects of the type are used, this is one of those areas where it is useful to consult technical documentation, which should explicitly state the requirements. For example, if we want to know what the requirements for a `Compare` are, we can look up the documentation for `std::max` (e.g. see <https://en.cppreference.com/w/cpp/algorithm/max> (<https://en.cppreference.com/w/cpp/algorithm/max>)³) and it should be listed there.

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 and represent "any reasonable type".

If the type template parameter has a non-obvious usage or specific requirements that must be met, then a more descriptive name is warranted (e.g. `Allocator` or `TAllocator`).

Quiz time

Question #1

Describe why construction blueprints are a type of template.

[Show Solution](#) (javascript:void(0))⁴

 **[Next lesson](#)**
11.7 [Function template instantiation](#)

5

 **[Back to table of contents](#)**

6

 **[Previous lesson](#)**
11.5 [Default arguments](#)

7

8




- B
- U
- URL
- INLINE CODE
- C++ CODE BLOCK
- HELP!


Leave a comment...

 Name*

@ Email*  

 Find a mistake? Leave a comment above!?

 Avatars from <https://gravatar.com/>¹¹ are connected to your provided email address.

Notify me about replies: 

POST COMMENT

245 COMMENTS

Newest ▼



Copernicus
May 7, 2025 9:27 pm PDT

Question #1 because construction blueprints are created once and can reused as many times as need to construct said item/building. Construction blueprints only detail how make something not what is

needed.

👍 1 ➡ Reply



smart

🕒 March 8, 2025 7:11 am PST

which is better?

```
1 | template <typename T>
2 | T max(T x, T y) {
3 |     return (x < y) ? y : x;
4 | }
```

or

```
1 | auto max(auto x, auto y)
2 | {
3 |     return (x < y) ? y : x;
4 | }
```

👍 0 ➡ Reply



YFN

➡ Reply to [smart](#)¹² 🕒 June 5, 2025 1:20 pm PDT

The first one.

You can't use type deduction for function parameter types. The end of lesson 10.9 says that the `auto` version will fail to compile prior to C++20, and that the `auto` keyword means something else in that context in C++20 or later.

👍 0 ➡ Reply



RSH

➡ Reply to [smart](#)¹² 🕒 May 6, 2025 2:16 am PDT

So o which was better from ur answers

👍 0 ➡ Reply



smart

➡ Reply to [smart](#)¹² 🕒 March 25, 2025 1:06 pm PDT

I found the answer in 11.8

👍 0 ➡ Reply

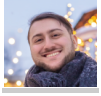


smart

➡ Reply to [smart](#)¹² 🕒 March 8, 2025 9:31 am PST

 Last edited 3 months ago by smart

 0  Reply



Tchelet Levi

 February 2, 2025 10:26 am PST

C++ Templates seem very similar to Generics from languages such as C#, Java, Typescript and others. Is it wrong to treat them as such?

I'm wondering this because at first glance it seems that they try to solve the same problem through very similar means, but maybe I'm missing how these ideas are incompatible with each other (if they are incompatible that is).

As always, thank you for your hard work on this amazing website! :)

 0  Reply



Alex

Author

 Reply to [Tchelet Levi](#) ¹³  February 4, 2025 8:53 pm PST

Yup, they're similar conceptually.

 3  Reply



rkh

 February 1, 2025 9:30 am PST

in the middle of the lesson, I was confused by this sentence `Second, we're going to tell the compiler that this is a template.`

Is that a template or a function template?

Are there differences between function templates and templates?

template :

"""

T max(T x, T y)

return (x < y) ? y : x;

}

"""

function template:

"""

template <typename T>

T max(T x, T y)

{

return (x < y) ? y : x;

}

Is this correct?

 Last edited 4 months ago by rkh

 0  Reply



Alex Author

 Reply to [rkh](#)¹⁴  February 2, 2025 10:21 pm PST

There are different kinds of templates: function templates, class template, type alias template, variable templates, and concept templates. ALL of these start with a template parameter declaration (exception: the abbreviated function template syntax). The type of template is then determined by what follows the template parameter declaration.

Put another way:

```
1 | template <typename T> // This is a template (of some kind) with type template  
  | parameter T  
2 | T max(T x, T y)      // And that "some kind" is a function template  
3 | ...
```

 1  Reply



rkh

 February 1, 2025 7:03 am PST

Hi Alex,

1. My question may seem strange, but I want to know what is different about using a Template rather than

auto to generate a variable or definition!

both of them require the compiler to detect the type and also able to work with different types

2. Do the arguments passed to a template function call (for a function with prototype `T add (T a, T b)`)

have to be the same? For example, if we call `add(2.5, 4)`, will the compiler complain?

3. Are not there any matching functions for the call to `add(auto x, auto y)`?

Thanks.

 Last edited 4 months ago by rkh

 0  Reply



BiscuitRE-L

 Reply to [rkh](#)¹⁵  February 20, 2025 8:41 pm PST

This small code that i wrote to understand how templates work with namespaces... might help you,
funTemp.h

```

1  #ifndef FUN_TEMPLATE
2  #define FUN_TEMPLATE
3  #pragma once
4
5  #include <iostream>
6  //template <typename T> you cant have it declared in global namespace and
   think it will be available across namespaces
7
8  //when using forward declaration on header files, fo fun_templates you have
   to define it in the header!!
9
10 namespace Templates {
11     namespace GetValue {
12         template <typename T>
13         T userInput(T x) {
14             T value;
15             std::cout << "Enter value #" << x << ": ";
16             std::cin >> value;
17             return value;
18         }
19     }
20     namespace CalcSum {
21         template <typename T>
22         T addValue(T x, T y) {
23             T sum;
24             sum = x + y;
25             //std::cout << "Result: " << sum << std::endl; //compiler will
   not default to a void, it is MUST to have a return type. T is based on return
   type
26             return sum;
27         }
28     }
29 }
30 #endif // !FUN_TEMPLATE

```

funTemp.cpp

```

1  #include <iostream>
2  #include "funTemp.h"
3
4  int main() {
5      auto num1 = Templates::GetValue::userInput<double>(1); //you HAVE to
   specify using <type> otherwise it will decide from the argument value 1 which
   will be int
6      auto num2 = Templates::GetValue::userInput<double>(2);
7
8      auto sum = Templates::CalcSum::addValue<double>(num1, num2);
9      std::cout << "Results: " << sum << std::endl;
10 }

```

👍 0

➡ Reply



BiscuitRE-L

🗨 Reply to [BiscuitRE-L](#) ¹⁶ 🕒 February 20, 2025 8:43 pm PST

feel free to correct me if im wrong.. i also learnt this just now <3

👍 0

➡ Reply



Alex

Author

Reply to [rkh](#)¹⁵ February 2, 2025 9:37 pm PST

`auto` as a function parameter is discussed in lesson <https://www.learncpp.com/cpp-tutorial/function-templates-with-multiple-template-types/>

1

Reply



PooperShooter

December 28, 2024 12:01 pm PST

```
1 #include <iostream>
2
3 template<class T>
4 const T& max(const T& a, const T& b)
5 {
6     return (a > b) ? a : b;
7 }
8
9 int main()
10 {
11     std::cout << max(1, 2);
12
13     return 0;
14 }
```

boom shakalaka

0

Reply



PooperShooter

December 28, 2024 11:51 am PST

I swear the "we'll cover these in a future lesson" is catching up to me and im starting to go clinically insane

3

Reply



BiscuitRE-L

Reply to [PooperShooter](#)¹⁷ February 20, 2025 8:49 pm PST

LMAOO same here waiting till i get pointers

1

Reply



yassin

November 21, 2024 7:04 am PST

I have a question , what could be the advantage of templates over just using auto type deduction in the context of the function max?

1 Reply



Alex Author

Reply to [yassin](#) ¹⁸ November 25, 2024 3:43 pm PST

I don't understand what you mean by "using auto type deduction in the context of the function max?"
Can you clarify with an example?

0 Reply



Yassin

Reply to [Alex](#) ¹⁹ November 25, 2024 8:00 pm PST

Auto max(auto X, auto y)

1 Reply



Alex Author

Reply to [Yassin](#) ²⁰ November 29, 2024 3:30 pm PST

A few advantages of templates over abbreviated function templates:

- Templates work prior to C++20.
- Templates allow you to enforce having multiple arguments with the same type.
- Templates work with other related template features, such as concepts.

3 Reply



Matt

August 10, 2024 9:48 pm PDT

Your last section "for advanced readers" has me stumped -- why do we not care what type a and b are (the variables of type T) but the variable of type Compare must meet requirements? How does it know that Compare is not just another templated variable? Is the Compare class defined elsewhere that is just not included in the snippet, or have I missed a fundamental understanding of this topic?

Presumably this would work just fine:

```
1 | template <typename Compare>
2 | Compare max(Compare x, Compare y)
3 | {
4 |     return (x < y) ? y : x;
5 | }
```

Last edited 10 months ago by Matt

2 Reply

**Alex**

Author

Reply to [Matt](#)²¹ ⌚ August 11, 2024 12:01 pm PDT

Because that's how `std::max` is designed. :) When we see template parameters with single-letter names (e.g. `T`, `U`, `V`), those typically represent "any value type". When we see named template parameters, those typically have additional requirements.

Remember, when a function template is instantiated, the compiler replaces the the template parameters with the template arguments and then compiles the function. It doesn't know that `T` means "any type" and `Compare` means something else. It just does the substitutions, and then the function either compiles or it doesn't. `Compare` isn't defined anywhere else in the code -- the requirements are enforced solely by how `Compare` is used within the function. Therefore, in code, the requirements are implicit.

To explicitly see the requirements, we have to consult documentation. e.g. <https://en.cppreference.com/w/cpp/algorithm/max>, which tells us the requirements that the argument we pass in for `Compare` must adhere to.

Your snippet would work just fine, though the naming of the template parameter is misleading.

👍 4

➡ Reply

**Viktor M**

⌚ July 31, 2024 10:14 pm PDT

When the compiler generates object code, as you've stated before, it redefines every overloaded function with a unique name, so in fact it's not one function, but many.

How does it work for templates? Does the compiler go through the code and generates every possible template function that was/could be called?

👍 1

➡ Reply

**Alex**

Author

Reply to [Viktor M](#)²² ⌚ August 1, 2024 6:23 pm PDT

We answer this question next lesson.

👍 1

➡ Reply

Links

1. <https://www.learncpp.com/author/Alex/>
2. <https://www.learncpp.com/cpp-tutorial/function-templates-with-multiple-template-types/>
3. <https://en.cppreference.com/w/cpp/algorithm/max>

4. [javascript:void\(0\)](#)
5. <https://www.learncpp.com/cpp-tutorial/function-template-instantiation/>
6. <https://www.learncpp.com/>
7. <https://www.learncpp.com/cpp-tutorial/default-arguments/>
8. <https://www.learncpp.com/function-templates/>
9. <https://www.learncpp.com/cpp-tutorial/random-file-io/>
10. <https://www.learncpp.com/cpp-tutorial/overloading-operators-and-function-templates/>
11. <https://gravatar.com/>
12. <https://www.learncpp.com/cpp-tutorial/function-templates/#comment-608360>
13. <https://www.learncpp.com/cpp-tutorial/function-templates/#comment-607293>
14. <https://www.learncpp.com/cpp-tutorial/function-templates/#comment-607232>
15. <https://www.learncpp.com/cpp-tutorial/function-templates/#comment-607228>
16. <https://www.learncpp.com/cpp-tutorial/function-templates/#comment-607937>
17. <https://www.learncpp.com/cpp-tutorial/function-templates/#comment-605820>
18. <https://www.learncpp.com/cpp-tutorial/function-templates/#comment-604380>
19. <https://www.learncpp.com/cpp-tutorial/function-templates/#comment-604521>
20. <https://www.learncpp.com/cpp-tutorial/function-templates/#comment-604542>
21. <https://www.learncpp.com/cpp-tutorial/function-templates/#comment-600750>
22. <https://www.learncpp.com/cpp-tutorial/function-templates/#comment-600362>
23. <https://g.ezoic.net/privacy/learncpp.com>