11.8 — Function templates with multiple template types

ALEX[1] DECEMBER 18, 2024

In lesson 11.6 -- Function templates (https://www.learncpp.com/cpp-tutorial/function-templates/)[2], we wrote a function template to calculate the maximum of two values:

```
#include <iostream>

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

int main()
{
    std::cout << max(1, 2) << '\n';   // will instantiate max(int, int)
    std::cout << max(1.5, 2.5) << '\n'; // will instantiate max(double, double)

    return 0;
}
```

Now consider the following similar program:

```
#include <iostream>

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

int main()
{
    std::cout << max(2, 3.5) << '\n';  // compile error

    return 0;
}
```

You may be surprised to find that this program won't compile. Instead, the compiler will issue a bunch of (probably crazy looking) error messages. On Visual Studio, the author got the following:

```
Project3.cpp(11,18): error C2672: 'max': no matching overloaded function found
Project3.cpp(11,28): error C2782: 'T max(T,T)': template parameter 'T' is ambiguou
Project3.cpp(4): message : see declaration of 'max'
Project3.cpp(11,28): message : could be 'double'
Project3.cpp(11,28): message : or       'int'
Project3.cpp(11,28): error C2784: 'T max(T,T)': could not deduce template argument
Project3.cpp(4): message : see declaration of 'max'
```

In our function call `max(2, 3.5)`, we're passing arguments of two different types: one `int` and one `double`. Because we're making a function call without using angled brackets to specify an actual type, the compiler will first look to see if there is a non-template match for `max(int, double)`. It won't find one.

Next, the compiler will see if it can find a function template match (using template argument deduction, which we covered in lesson 11.7 -- Function template instantiation (https://www.learncpp.com/cpp-tutorial/function-template-instantiation/)[3]). However, this will also fail, for a simple reason: `T` can only represent a single type. There is no type for `T` that would allow the compiler to instantiate function template `max<T>(T, T)` into a function with two different parameter types. Put another way, because both parameters in the function template are of type `T`, they must resolve to the same actual type.

Since both a non-template match and a template match couldn't be found, the function call fails to resolve, and we get a compile error.

You might wonder why the compiler didn't generate function `max<double>(double, double)` and then use numeric conversion to type convert the `int` argument to a `double`. The answer is simple: type conversion is done only when resolving function overloads, not when performing template argument deduction.

This lack of type conversion is intentional for at least two reasons. First, it helps keep things simple: we either find an exact match between the function call arguments and template type parameters, or we don't. Second, it allows us to create function templates for cases where we want to ensure that two or more parameters have the same type (as in the example above).

We'll have to find another solution. Fortunately, we can solve this problem in (at least) three ways.

Use static_cast to convert the arguments to matching types

The first solution is to put the burden on the caller to convert the arguments into matching types. For example:

```cpp
#include <iostream>

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

int main()
{
    std::cout << max(static_cast<double>(2), 3.5) << '\n'; // convert our int to a double so we can call max(double, double)

    return 0;
}
```

Now that both arguments are of type `double`, the compiler will be able to instantiate `max(double, double)` that will satisfy this function call.

However, this solution is awkward and hard to read.

Provide an explicit type template argument

If we had written a non-template `max(double, double)` function, then we would be able to call `max(int, double)` and let the implicit type conversion rules convert our `int` argument into a `double` so the function call could be resolved:

```cpp
#include <iostream>

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

int main()
{
    std::cout << max(2, 3.5) << '\n'; // the int argument will be converted to a double

    return 0;
}
```

However, when the compiler is doing template argument deduction, it won't do any type conversions. Fortunately, we don't have to use template argument deduction if we specify an explicit type template argument to be used instead:

```
#include <iostream>

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

int main()
{
    // we've explicitly specified type double, so the compiler won't use template argument deduction
    std::cout << max<double>(2, 3.5) << '\n';

    return 0;
}
```

In the above example, we call `max<double>(2, 3.5)`. Because we've explicitly specified that `T` should be replaced with `double`, the compiler won't use template argument deduction. Instead, it will just instantiate the function `max<double>(double, double)`, and then type convert any mismatched arguments. Our `int` parameter will be implicitly converted to a `double`.

While this is more readable than using `static_cast`, it would be even nicer if we didn't even have to think about the types when making a function call to `max` at all.

Function templates with multiple template type parameters

The root of our problem is that we've only defined the single template type (`T`) for our function template, and then specified that both parameters must be of this same type.

The best way to solve this problem is to rewrite our function template in such a way that our parameters can resolve to different types. Rather than using one template type parameter `T`, we'll now use two (`T` and `U`):

```
#include <iostream>

template <typename T, typename U> // We're using two template type parameters named T and U
T max(T x, U y) // x can resolve to type T, and y can resolve to type U
{
    return (x < y) ? y : x; // uh oh, we have a narrowing conversion problem here
}

int main()
{
    std::cout << max(2, 3.5) << '\n'; // resolves to max<int, double>

    return 0;
}
```

Because we've defined `x` with template type `T`, and `y` with template type `U`, `x` and `y` can now resolve their types independently. When we call `max(2, 3.5)`, `T` can be an `int` and `U` can be a `double`. The compiler will happily instantiate `max<int, double>(int, double)` for us.

Key insight

Because `T` and `U` are independent template parameters, they resolve their types independent of each other. This means `T` and `U` can resolve to different types, or they can resolve to the same type.

However, this example doesn't work right. If you compile and run the program (with "treat warnings as errors" turned off), it will produce the following result:

```
3
```

What's going on here? How can the max of `2` and `3.5` be `3`?

The conditional operator (?:) requires its (non-condition) operands to be the same common type. The usual arithmetic rules (10.5 -- Arithmetic conversions (https://www.learncpp.com/cpp-tutorial/arithmetic-conversions/)[4]) are used to determine what the common type will be, and the result of the conditional operator will also use this common type. For example, the common type of `int` and `double` is `double`, so when the (non-condition) operands of our conditional operator are an `int` and a `double`, the value produced by the conditional operator will be of type `double`. In this case, that's the value `3.5`, which is correct.

However, the declared return type of our function is `T`. When `T` is an `int` and `U` is a `double`, the return type of the function is `int`. Our value `3.5` is undergoing a narrowing conversion to `int` value `3`, resulting in a loss of data (and possibly a compiler warning).

So how do we solve this? Making the return type a `U` instead doesn't solve the problem, as `max(3.5, 2)` has `U` as an `int` and will exhibit the same issue.

In such cases, return type deduction (via `auto`) can be useful -- we'll let the compiler deduce what the return type should be from the return statement:

```
#include <iostream>

template <typename T, typename U>
auto max(T x, U y) // ask compiler can figure out what the relevant return type is
{
    return (x < y) ? y : x;
}

int main()
{
    std::cout << max(2, 3.5) << '\n';

    return 0;
}
```

This version of `max` now works fine with operands of different types. Just note that a function with an `auto` return type needs to be fully defined before it can be used (a forward declaration won't suffice), since the compiler has to inspect the function implementation to determine the return type.

For advanced readers

If we need a function that can be forward declared, we have to be explicit about the return type. Since our return type needs to be the common type of `T` and `U`, we can use `std::common_type_t` (discussed in lesson 10.5 -- Arithmetic conversions (https://www.learncpp.com/cpp-tutorial/arithmetic-conversions/)[4]) to fetch the common type of `T` and `U` to use as our explicit return type:

```cpp
#include <iostream>
#include <type_traits> // for std::common_type_t

template <typename T, typename U>
auto max(T x, U y) -> std::common_type_t<T, U>; // returns the common type of T
and U

int main()
{
    std::cout << max(2, 3.5) << '\n';

    return 0;
}

template <typename T, typename U>
auto max(T x, U y) -> std::common_type_t<T, U>
{
    return (x < y) ? y : x;
}
```

Abbreviated function templates C++20

C++20 introduces a new use of the `auto` keyword: When the `auto` keyword is used as a parameter type in a normal function, the compiler will automatically convert the function into a function template with each auto parameter becoming an independent template type parameter. This method for creating a function template is called an **abbreviated function template**.

For example:

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

is shorthand in C++20 for the following:

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

which is the same as the `max` function template we wrote above.

In cases where you want each template type parameter to be an independent type, this form is preferred as the removal of the template parameter declaration line makes your code more concise and readable.

There isn't a concise way to use abbreviated function templates when you want more than one auto parameter to be the same type. That is, there isn't an easy abbreviated function template for something like this:

```cpp
template <typename T>
T max(T x, T y) // two parameters of the same type
{
    return (x < y) ? y : x;
}
```

Best practice

Feel free to use abbreviated function templates with a single auto parameter, or where each auto parameter should be an independent type (and your language standard is set to C++20 or newer).

Function templates may be overloaded

Just like functions may be overloaded, function templates may also be overloaded. Such overloads can have a different number of template types and/or a different number or type of function parameters:

```
#include <iostream>

// Add two values with matching types
template <typename T>
auto add(T x, T y)
{
    return x + y;
}

// Add two values with non-matching types
// As of C++20 we could also use auto add(auto x, auto y)
template <typename T, typename U>
auto add(T x, U y)
{
    return x + y;
}

// Add three values with any type
// As of C++20 we could also use auto add(auto x, auto y, auto z)
template <typename T, typename U, typename V>
auto add(T x, U y, V z)
{
    return x + y + z;
}

int main()
{
    std::cout << add(1.2, 3.4) << '\n'; // instantiates and calls add<double>()
    std::cout << add(5.6, 7) << '\n';   // instantiates and calls add<double, int>()
    std::cout << add(8, 9, 10) << '\n'; // instantiates and calls add<int, int, int>()

    return 0;
}
```

One interesting note here is that for the call to `add(1.2, 3.4)`, the compiler will prefer `add<T>(T, T)` over `add<T, U>(T, U)` even though both could possibly match.

The rules for determining which of multiple matching function templates should be preferred are called "partial ordering of function templates". In short, whichever function template is more restrictive/specialized will be preferred. `add<T>(T, T)` is the more restrictive function template in this case (since it only has one template parameter), so it is preferred.

If multiple function templates can match a call and the compiler can't determine which is more restrictive, the compiler will error with an ambiguous match.

Next lesson
11.9 Non-type template parameters

5

Back to table of contents

6

Previous lesson
11.7 Function template instantiation

3

7

124 COMMENTS

Newest

Wide

April 28, 2025 2:08 am PDT

Two questions:

1. Is there a reason why you used the trailing return syntax here?

```cpp
#include <iostream>
#include <type_traits> // for std::common_type_t

template <typename T, typename U>
auto max(T x, U y) -> std::common_type_t<T, U>; // returns the common type of T and U

int main()
{
    std::cout << max(2, 3.5) << '\n';

    return 0;
}

template <typename T, typename U>
auto max(T x, U y) -> std::common_type_t<T, U>
{
    return (x < y) ? y : x;
}
```

I tried running the code on my machine without the trailing return type, `std::common_type_t<T, U> max(T x, U y)`, and it ran just fine without any errors or warnings.

2. Could the above function template be rewritten in the form of an abbreviation?

Last edited 2 months ago by Wide

0 Reply

YFN

Reply to Wide [11] June 8, 2025 9:47 am PDT

1. I think the trailing return type is for improved readability. `std::common_type_t<T, U>` is a bit of a complicated type, and so it's just easier to put it at the end of the line instead of at the beginning.
2. No, because abbreviated templates have a return type of `auto`. And functions using the return type of `auto` must be fully defined before they are used. Here, the function template is used on line 9, but not defined until lines 14 through 18. The forward declaration isn't enough. You could move the definition earlier, but it's just safer to avoid using an `auto` return type if you will be forward-declaring the function. Especially since most of the time, if you are forward-declaring a function, it means the actual definition will be in a different file, and thus only visible to the linker, not the compiler.

Last edited 19 days ago by YFN

0 Reply

Ccongcong

Reply to Wide [11] June 3, 2025 8:38 am PDT

1.using the trailing return syntax so the author can use the forward declaration statement.
2.No!Because Abbreviated function can only be used in function definitions, not declarations.

0 Reply

Na NA

March 18, 2025 6:28 am PDT

```
#include <iostream>
#include <type_traits>

template<typename T, typename U>
std::common_type_t<T,U> max(T x, U y);

int main() {
std::cout<<max(10,20.5);
return 0;
}

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

We can use common_type_t as return type. Isn't it a good choice instead of using auto?

1 Reply

Hovsep Papoyan

March 2, 2025 4:15 pm PST

Nice example, discussed in "C++ Templates: The Complete Guide" !!!

Last edited 3 months ago by Hovsep Papoyan

0 Reply

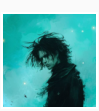
Renaissanceo0

January 20, 2025 8:47 pm PST

"The compiler will happily instantiate `max<int, double>(int, double)` for us"
Should it be `max<int>(int, double)`? Since I suppose that `max<int>(int, double)` calls a function declared as

```
1 | int max(int, double);
```

0 Reply



Bassem

Reply to Renaissanceo0 [12] February 2, 2025 3:36 am PST

There's no difference. the only thing max<int>(int, double) did is explicitly specify the type of the first argument T to int, and because it's already int type , argument type deduction would resolve T to int anyway. max<int , double>(int , double) specified both arguments T and U. max(int , double) would be deduced also to max<int , double>(int ,double)

note that , we match the <int , double > with the order of the template <typename T , typname U>
T >> int
U >> double
etc.

and then the only thing the compiler does is the substitution.

Last edited 4 months ago by Bassem

0 Reply

XenophonSoulis

Reply to Renaissanceo0 [12] January 23, 2025 1:29 am PST

The stuff you put in the <> is exactly the stuff that are in the template declaration, regardless of where they may be used. Since the declaration is <typename T, typename U>, T is replaced by int and U is replaced by double. The following compiles normally:

```
template <typename T, typename U>
char my_template(U u, T t1, T t2, T t3) {
    std::cout << t1 << ' ' << t2 << ' ' << t3 << '\n';
    std::cout << "Oh, also " << u << '\n';
    return 'a';
}

int main() {
    std::cout << "The returned value was " << my_template<int, double>(3.5,
2, 4, 7) << ".\n";
    return 0;
}
```

We used <int, double> because we wanted T as int and U as double, regardless of where and in what order they are used (and neither is used for the return value). The return type doesn't have to match any of them. This would print:

1 2 7
Oh, also 3.5
The returned value was a.

1 Reply

Surab

November 26, 2024 1:28 am PST

Hey! The given issue about conversion of double to int in multiple template parameter actually prints me the answer like it should not the 3 you get there!

```cpp
#include<iostream>
template<typename T,typename U>
T max(T x, U y) {
    return(x < y) ? y : x;
}
int main() {
    std::cout << max(2.2, 1) << std::endl;
    return 0;
}
```

here the compiler returns 2.2 logically on Visual Studio 2022 which has got me confused.

0 Reply

Alex Author

Reply to Surab [13] November 29, 2024 3:51 pm PST

Like Dengchen88 said, in your case, T is double and U is int. Since the function returns a T, which is a double here, it works.

But if you were to flip the order of the arguments, it wouldn't work.

0 Reply

Surab

Reply to Alex [14] December 2, 2024 9:39 am PST

Damn that was so dumb, thanks for the replies!

0 Reply

dengchen88

Reply to Surab [13] November 29, 2024 10:19 am PST

T = double

double int
max(2.2, 1)

0 Reply

Will

October 9, 2024 5:47 am PDT

Dam its so op

1 Reply

David

September 22, 2024 12:28 am PDT

Wouldn't it be better to use auto return again in the last example, instead of T, to avoid narrowing conversion?

`add(20,20.5)` will give 40.

Since the comments also use abbreviated function templates with the auto type.

```
// Add two values with non-matching types
// As of C++20 we could also use auto add(auto x, auto y)
template <typename T, typename U>
T add(T x, U y)
{
    return x + y;
}

// Add three values with any type
// As of C++20 we could also use auto add(auto x, auto y, auto z)
template <typename T, typename U, typename V>
T add(T x, U y, V z)
{
    return x + y + z;
}
```

1 Reply

Alex Author

Reply to David [15] September 23, 2024 11:27 pm PDT

Yes, thank you for the suggestion.

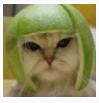
1 Reply

Adrian

September 18, 2024 7:56 pm PDT

Hi Alex. I have a question. During the process of learning, I find a strange thing: In C++, if we only consider coding in one file, there are two kinds of functions that truly bother me. One is functions like constexpr functions, functions returning auto. We cannot first declare that function, then use that function in main() and define the function at the bottom of the file. In my opinion, this is because the compiler scans the file from top to bottom, by the time the compiler sees functions used in main(), it cannot calculate the value(constexpr functions) or deduce the return type(functions returning auto). So the compiler has to see the definitions first or it will report a compile error. But when I try the same process(declare-use-define) for template function (like max<T>), I find it works! But generating functions using template functions needs to see the function body, right? But by the time the compiler sees the use of max<int> in main(), the compiler hasn't seen the definition of template function yet. That's strange! Besides, inline expansion also needs definition of the function. Does that mean we should make all the functions defined at the top of the file or the functions will not be inline expanded by compilers? Maybe the compiler doesn't compile the file from top to bottom?

0 Reply



Alex Author

Reply to Adrian [16] September 23, 2024 5:46 pm PDT

Good questions. The compiler needs to see a full definition prior to use in certain cases, including:

- When the compiler needs to evaluate or expand a function at compile-time.
- When actual types cannot be determined from a forward declaration.

Given this:

1. Constexpr functions need a full definition if they are used in a context that requires a constant expression (as they must be evaluated at compile-time). Otherwise a forward declaration will suffice (and they will be evaluated at runtime).
2. Functions that return auto need a full definition to be used, otherwise the compiler can't infer the return type.
3. Template functions don't need a full definition to be used. They are called at runtime, and the compiler can determine the value of the template types from the forward declaration.
4. Inline functions need a full definition in order for inline expansion to be performed. If a full definition hasn't been seen yet, inline expansion won't happen.

The compiler does compile from top to bottom. Generally you'll want to define your constexpr and inline functions before calling them. One easy way to do that is put their definitions in a header file (since being inline makes them exempt from the part of the ODR that causes multiple definitions in a program to be a violation).

2 Reply

Strain

June 10, 2024 1:52 pm PDT

"There isn't a concise way to use abbreviated function templates when you want more than one auto parameter to be the same type."

I'm thinking of one. Concepts (C++20), when used in the short-hand for parameters, will always be the same. E.g., considering some concept `SomeConcept`:

```
1 void func(SomeConcept a, SomeConcept b);
2 // This is shorthand for:
3 template<SomeConcept T>
4 void func(T a, T b);
```

Contrary to `auto`'s behaviour:

```
1 void func(auto a, auto b);
2 // This is shorthand for:
3 template<typename T, typename U>
4 void func(T a, U b);
```

Ergo, if we defined a concept `Auto<typename T>` that was always `true`, it would be similar to `auto`, except that every `Auto` parameter will have the same type in the signature.

Last edited 1 year ago by Strain

0 Reply

Alex Author

Reply to Strain [17] June 11, 2024 12:44 pm PDT

I wasn't aware of this feature, so thank you for introducing me to it. That said, this doesn't make the parameter types the same, it just ensures they are both constrained in the same way. I suppose if your constraint was narrow enough that it only applied to a single type it might accomplish the same thing, but this seems like a lot of work just to avoid a template parameter declaration.

1 Reply

axel

Reply to Alex [18] July 28, 2024 1:57 am PDT

One can use of decltype to get a concise way of writing abbreviated function templates with two parameters of same type. Namely,

```
1 auto func(auto x, decltype(x) y);
```

However this expression has the drawback that it is not recognized as a partial template specialization of

```
1 auto func(auto x, auto y);
```

if both are present (on clang 14 and gcc 14). In case of specifying it as a partial template specialization one can write

```
1 #include <concepts>
2 auto func(auto x, auto y) requires std::same_as<decltype(x),decltype(y)>
  ;
```

but that is more verbose than explicitly writing template parameters I think

_

`

Last edited 11 months ago by axel

0 Reply

axel

Reply to axel [19] July 31, 2024 1:30 am PDT

I realized that my suggestion is wrong. Actually, with auto func(auto x, decltype(x) y), x and y don't have same type, but similar types. Or more strictly, the type of x has to be convertible to the type of y. Hence, this implementation allows for use misuse do to implicit conversion. Consider for example the call func(2, 2.5). Here, the double(2.5) will be implicitly converted to int(2) likely leading to incorrect behavior. Thus, I retract my suggestion.

1 Reply

emme

April 18, 2024 1:34 pm PDT

In the example

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

Does it mean that the compiler can't make both parameters of the same type and will always make them different types? And if so, does that mean I need to write another template for the functions with the same parameter type?

0 Reply

Alex Author

Reply to emme [20] April 19, 2024 12:59 pm PDT

`T` and `U` can be the same type or different types.

For example, if you call `max(2, 3)`, this will call `max<int, int>`, where both `T` and `U` resolve to type `int`. If you call `max(1, 2.3)`, this will call `max<int, double>`, where `T` resolves to `int` and `U` resolves to `double`.

5 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/function-templates/
3. https://www.learncpp.com/cpp-tutorial/function-template-instantiation/
4. https://www.learncpp.com/cpp-tutorial/arithmetic-conversions/
5. https://www.learncpp.com/cpp-tutorial/non-type-template-parameters/
6. https://www.learncpp.com/
7. https://www.learncpp.com/function-templates-with-multiple-template-types/
8. https://www.learncpp.com/cpp-tutorial/function-overload-resolution-and-ambiguous-matches/
9. https://www.learncpp.com/cpp-tutorial/chapter-10-summary-and-quiz/
10. https://gravatar.com/
11. https://www.learncpp.com/cpp-tutorial/function-templates-with-multiple-template-types/#comment-609620
12. https://www.learncpp.com/cpp-tutorial/function-templates-with-multiple-template-types/#comment-606804
13. https://www.learncpp.com/cpp-tutorial/function-templates-with-multiple-template-types/#comment-604547
14. https://www.learncpp.com/cpp-tutorial/function-templates-with-multiple-template-types/#comment-604673
15. https://www.learncpp.com/cpp-tutorial/function-templates-with-multiple-template-types/#comment-602202

16. https://www.learncpp.com/cpp-tutorial/function-templates-with-multiple-template-types/#comment-602107
17. https://www.learncpp.com/cpp-tutorial/function-templates-with-multiple-template-types/#comment-598250
18. https://www.learncpp.com/cpp-tutorial/function-templates-with-multiple-template-types/#comment-598293
19. https://www.learncpp.com/cpp-tutorial/function-templates-with-multiple-template-types/#comment-600194
20. https://www.learncpp.com/cpp-tutorial/function-templates-with-multiple-template-types/#comment-595942
21. https://g.ezoic.net/privacy/learncpp.com