11.7 — Function template instantiation

👤 **ALEX**[1]   🕐 **AUGUST 21, 2024**

In the previous lesson ([11.6 -- Function templates](https://www.learncpp.com/cpp-tutorial/function-templates/)[2]), we introduced function templates, and converted a normal `max()` function into a `max<T>` function template:

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

In this lesson, we'll focus on how function templates are used.

Using a function template

Function templates are not actually functions -- their code isn't compiled or executed directly. Instead, function templates have one job: to generate functions (that are compiled and executed).

To use our `max<T>` function template, we can make a function call with the following syntax:

```
max<actual_type>(arg1, arg2); // actual_type is some actual type, like int or doub
```

This looks a lot like a normal function call -- the primary difference is the addition of the type in angled brackets (called a **template argument**), which specifies the actual type that will be used in place of template type `T`.

Let's take a look at this in a simple example:

```
#include <iostream>

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

int main()
{
    std::cout << max<int>(1, 2) << '\n'; // instantiates and calls function max<int>
(int, int)

    return 0;
}
```

When the compiler encounters the function call `max<int>(1, 2)`, it will determine that a function definition for `max<int>(int, int)` does not already exist. Consequently, the compiler will implicitly use

our `max<T>` function template to create one.

The process of creating functions (with specific types) from function templates (with template types) is called **function template instantiation** (or **instantiation** for short). When a function is instantiated due to a function call, it's called **implicit instantiation**. A function that is instantiated from a template is technically called a **specialization**, but in common language is often called a **function instance**. The template from which a specialization is produced is called a **primary template**. Function instances are normal functions in all regards.

> ## Nomenclature
>
> The term "specialization" is more often used to refer to explicit specialization, which allows us to explicitly define a specialization (rather than have it implicitly instantiated from the primary template). We cover explicit specialization in lesson [26.3 -- Function template specialization](https://www.learncpp.com/cpp-tutorial/function-template-specialization/)[3].

The process for instantiating a function is simple: the compiler essentially clones the primary template and replaces the template type (`T`) with the actual type we've specified (`int`).

So when we call `max<int>(1, 2)` , the function specialization that gets instantiated looks something like this:

```
1 | template<> // ignore this for now
2 | int max<int>(int x, int y) // the generated function max<int>(int, int)
3 | {
4 |     return (x < y) ? y : x;
5 | }
```

Here's the same example as above, showing what the compiler actually compiles after all instantiations are done:

```
1  | #include <iostream>
2  |
3  | // a declaration for our function template (we don't need the definition any more)
4  | template <typename T>
5  | T max(T x, T y);
6  |
7  | template<>
8  | int max<int>(int x, int y) // the generated function max<int>(int, int)
9  | {
10 |     return (x < y) ? y : x;
11 | }
12 |
13 | int main()
14 | {
15 |     std::cout << max<int>(1, 2) << '\n'; // instantiates and calls function max<int>
16 | (int, int)
17 |
18 |     return 0;
   | }
```

You can compile this yourself and see that it works. A function template is only instantiated the first time a function call is made in each translation unit. Further calls to the function are routed to the already instantiated function.

Conversely, if no function call is made to a function template, the function template won't be instantiated in that translation unit.

Let's do another example:

```cpp
#include <iostream>

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

int main()
{
    std::cout << max<int>(1, 2) << '\n';    // instantiates and calls function max<int>(int, int)
    std::cout << max<int>(4, 3) << '\n';    // calls already instantiated function max<int>(int, int)
    std::cout << max<double>(1, 2) << '\n'; // instantiates and calls function max<double>(double, double)

    return 0;
}
```

This works similarly to the previous example, but our function template will be used to generate two functions this time: one time replacing `T` with `int`, and the other time replacing `T` with `double`. After all instantiations, the program will look something like this:

```cpp
#include <iostream>

// a declaration for our function template (we don't need the definition any more)
template <typename T>
T max(T x, T y);

template<>
int max<int>(int x, int y) // the generated function max<int>(int, int)
{
    return (x < y) ? y : x;
}

template<>
double max<double>(double x, double y) // the generated function max<double>(double, double)
{
    return (x < y) ? y : x;
}

int main()
{
    std::cout << max<int>(1, 2) << '\n';    // instantiates and calls function max<int>(int, int)
    std::cout << max<int>(4, 3) << '\n';    // calls already instantiated function max<int>(int, int)
    std::cout << max<double>(1, 2) << '\n'; // instantiates and calls function max<double>(double, double)

    return 0;
}
```

One additional thing to note here: when we instantiate `max<double>`, the instantiated function has parameters of type `double`. Because we've provided `int` arguments, those arguments will be implicitly

converted to `double`.

Template argument deduction

In most cases, the actual types we want to use for instantiation will match the type of our function parameters. For example:

```
1 | std::cout << max<int>(1, 2) << '\n'; // specifying we want to call max<int>
```

In this function call, we've specified that we want to replace `T` with `int`, but we're also calling the function with `int` arguments.

In cases where the type of the arguments match the actual type we want, we do not need to specify the actual type -- instead, we can use **template argument deduction** to have the compiler deduce the actual type that should be used from the argument types in the function call.

For example, instead of making a function call like this:

```
1 | std::cout << max<int>(1, 2) << '\n'; // specifying we want to call max<int>
```

We can do one of these instead:

```
1 | std::cout << max<>(1, 2) << '\n';
2 | std::cout << max(1, 2) << '\n';
```

In either case, the compiler will see that we haven't provided an actual type, so it will attempt to deduce an actual type from the function arguments that will allow it to generate a `max()` function where all template parameters match the type of the provided arguments. In this example, the compiler will deduce that using function template `max<T>` with actual type `int` allows it to instantiate function `max<int>(int, int)`, so that the type of both function parameters (`int`) matches the type of the provided arguments (`int`).

The difference between the two cases has to do with how the compiler resolves the function call from a set of overloaded functions. In the top case (with the empty angled brackets), the compiler will only consider `max<int>` template function overloads when determining which overloaded function to call. In the bottom case (with no angled brackets), the compiler will consider both `max<int>` template function overloads and `max` non-template function overloads. When the bottom case results in both a template function and a non-template function that are equally viable, the non-template function will be preferred.

> **Key insight**
>
> The normal function call syntax will prefer a non-template function over an equally viable function instantiated from a template.

For example:

```cpp
#include <iostream>

template <typename T>
T max(T x, T y)
{
    std::cout << "called max<int>(int, int)\n";
    return (x < y) ? y : x;
}

int max(int x, int y)
{
    std::cout << "called max(int, int)\n";
    return (x < y) ? y : x;
}

int main()
{
    std::cout << max<int>(1, 2) << '\n'; // calls max<int>(int, int)
    std::cout << max<>(1, 2) << '\n';    // deduces max<int>(int, int) (non-template
functions not considered)
    std::cout << max(1, 2) << '\n';      // calls max(int, int)

    return 0;
}
```

Note how the syntax in the bottom case looks identical to a normal function call! In most cases, this normal function call syntax will be the one we use to call functions instantiated from a function template.

There are a few reasons for this:

- The syntax is more concise.
- It's rare that we'll have both a matching non-template function and a function template.
- If we do have a matching non-template function and a matching function template, we will usually prefer the non-template function to be called.

That last point may be non-obvious. A function template has an implementation that works for multiple types -- but as a result, it must be generic. A non-template function only handles a specific combination of types. It can have an implementation that is more optimized or more specialized for those specific types than the function template version. For example:

```cpp
#include <iostream>

// This function template can handle many types, so its implementation is generic
template <typename T>
void print(T x)
{
    std::cout << x; // print T however it normally prints
}

// This function only needs to consider how to print a bool, so it can specialize how it handles
// printing of a bool
void print(bool x)
{
    std::cout << std::boolalpha << x; // print bool as true or false, not 1 or 0
}

int main()
{
    print<bool>(true); // calls print<bool>(bool) -- prints 1
    std::cout << '\n';

    print<>(true);     // deduces print<bool>(bool) (non-template functions not considered) -- prints 1
    std::cout << '\n';

    print(true);       // calls print(bool) -- prints true
    std::cout << '\n';

    return 0;
}
```

Best practice

Favor the normal function call syntax when making calls to a function instantiated from a function template (unless you need the function template version to be preferred over a matching non-template function).

Function templates with non-template parameters

It's possible to create function templates that have both template parameters and non-template parameters. The type template parameters can be matched to any type, and the non-template parameters work like the parameters of normal functions.

For example:

```
1   // T is a type template parameter
2   // double is a non-template parameter
3   // We don't need to provide names for these parameters since they aren't used
4   template <typename T>
5   int someFcn(T, double)
6   {
7       return 5;
8   }
9
10  int main()
11  {
12      someFcn(1, 3.4); // matches someFcn(int, double)
13      someFcn(1, 3.4f); // matches someFcn(int, double) -- the float is promoted to a
14  double
15      someFcn(1.2, 3.4); // matches someFcn(double, double)
16      someFcn(1.2f, 3.4); // matches someFcn(float, double)
17      someFcn(1.2f, 3.4f); // matches someFcn(float, double) -- the float is promoted to
18  a double
19
        return 0;
    }
```

This function template has a templated first parameter, but the second parameter is fixed with type `double` . Note that the return type can also be any type. In this case, our function will always return an `int` value.

Instantiated functions may not always compile

Consider the following program:

```
1   #include <iostream>
2
3   template <typename T>
4   T addOne(T x)
5   {
6       return x + 1;
7   }
8
9   int main()
10  {
11      std::cout << addOne(1) << '\n';
12      std::cout << addOne(2.3) << '\n';
13
14      return 0;
15  }
```

The compiler will effectively compile and execute this:

```
1   #include <iostream>
2
3   template <typename T>
4   T addOne(T x);
5
6   template<>
7   int addOne<int>(int x)
8   {
9       return x + 1;
10  }
11
12  template<>
13  double addOne<double>(double x)
14  {
15      return x + 1;
16  }
17
18  int main()
19  {
20      std::cout << addOne(1) << '\n';    // calls addOne<int>(int)
21      std::cout << addOne(2.3) << '\n'; // calls addOne<double>(double)
22
23      return 0;
24  }
```

which will produce the result:

```
2
3.3
```

But what if we try something like this?

```
1   #include <iostream>
2   #include <string>
3
4   template <typename T>
5   T addOne(T x)
6   {
7       return x + 1;
8   }
9
10  int main()
11  {
12      std::string hello { "Hello, world!" };
13      std::cout << addOne(hello) << '\n';
14
15      return 0;
16  }
```

When the compiler tries to resolve `addOne(hello)` it won't find a non-template function match for `addOne(std::string)`, but it will find our function template for `addOne(T)`, and determine that it can generate an `addOne(std::string)` function from it. Thus, the compiler will generate and compile this:

```
1    #include <iostream>
2    #include <string>
3
4    template <typename T>
5    T addOne(T x);
6
7    template<>
8    std::string addOne<std::string>(std::string x)
9    {
10       return x + 1;
11   }
12
13   int main()
14   {
15       std::string hello{ "Hello, world!" };
16       std::cout << addOne(hello) << '\n';
17
18       return 0;
19   }
```

However, this will generate a compile error, because `x + 1` doesn't make sense when `x` is a `std::string`. The obvious solution here is simply not to call `addOne()` with an argument of type `std::string`.

Instantiated functions may not always make sense semantically

The compiler will successfully compile an instantiated function template as long as it makes sense syntactically. However, the compiler does not have any way to check that such a function actually makes sense semantically.

For example:

```
1    #include <iostream>
2
3    template <typename T>
4    T addOne(T x)
5    {
6        return x + 1;
7    }
8
9    int main()
10   {
11       std::cout << addOne("Hello, world!") << '\n';
12
13       return 0;
14   }
```

In this example, we're calling `addOne()` on a C-style string literal. What does that actually mean semantically? Who knows!

Perhaps surprisingly, because C++ syntactically allows addition of an integer value to a string literal (we cover this in future lesson [17.9 -- Pointer arithmetic and subscripting](https://www.learncpp.com/cpp-tutorial/pointer-arithmetic-and-subscripting/)[4]), the above example compiles, and produces the following result:

```
ello, world!
```

Warning

For advanced readers

We can tell the compiler that instantiation of function templates with certain arguments should be disallowed. This is done by using function template specialization, which allow us to overload a function template for a specific set of template arguments, along with `= delete`, which tells the compiler that any use of the function should emit a compilation error.

```cpp
#include <iostream>
#include <string>

template <typename T>
T addOne(T x)
{
    return x + 1;
}

// Use function template specialization to tell the compiler that addOne(const char*) should emit a compilation error
// const char* will match a string literal
template <>
const char* addOne(const char* x) = delete;

int main()
{
    std::cout << addOne("Hello, world!") << '\n'; // compile error

    return 0;
}
```

We cover function template specialization in lesson [26.3 -- Function template specialization (https://www.learncpp.com/cpp-tutorial/function-template-specialization/)][3].

Function templates and default arguments for non-template parameters

Just like normal functions, function templates can have default arguments for non-template parameters. Each function instantiated from the template will use the same default argument.

For example:

```cpp
#include <iostream>

template <typename T>
void print(T val, int times=1)
{
    while (times--)
    {
        std::cout << val;
    }
}

int main()
{
    print(5);       // print 5 1 time
    print('a', 3); // print 'a' 3 times

    return 0;
}
```

This prints:

```
5aaa
```

Beware function templates with modifiable static local variables

In lesson 7.11 -- Static local variables (https://www.learncpp.com/cpp-tutorial/static-local-variables/)[5], we discussed static local variables, which are local variables with static duration (they persist for the lifetime of the program).

When a static local variable is used in a function template, each function instantiated from that template will have a separate version of the static local variable. This is rarely a problem if the static local variable is const. But if the static local variable is one that is modified, the results may not be as expected.

For example:

```cpp
#include <iostream>

// Here's a function template with a static local variable that is modified
template <typename T>
void printIDAndValue(T value)
{
    static int id{ 0 };
    std::cout << ++id << ") " << value << '\n';
}

int main()
{
    printIDAndValue(12);
    printIDAndValue(13);

    printIDAndValue(14.5);

    return 0;
}
```

This produces the result:

```
1) 12
2) 13
1) 14.5
```

You may have been expecting the last line to print `3) 14.5`. However, this is what the compiler actually compiles and executes:

```cpp
#include <iostream>

template <typename T>
void printIDAndValue(T value);

template <>
void printIDAndValue<int>(int value)
{
    static int id{ 0 };
    std::cout << ++id << ") " << value << '\n';
}

template <>
void printIDAndValue<double>(double value)
{
    static int id{ 0 };
    std::cout << ++id << ") " << value << '\n';
}

int main()
{
    printIDAndValue(12);   // calls printIDAndValue<int>()
    printIDAndValue(13);   // calls printIDAndValue<int>()

    printIDAndValue(14.5); // calls printIDAndValue<double>()

    return 0;
}
```

Note that `printIDAndValue<int>` and `printIDAndValue<double>` each have their own independent static local variable named `id`, not one that is shared between them.

Generic programming

Because template types can be replaced with any actual type, template types are sometimes called **generic types**. And because templates can be written agnostically of specific types, programming with templates is sometimes called **generic programming**. Whereas C++ typically has a strong focus on types and type checking, in contrast, generic programming lets us focus on the logic of algorithms and design of data structures without having to worry so much about type information.

Conclusion

Once you get used to writing function templates, you'll find they actually don't take much longer to write than functions with actual types. Function templates can significantly reduce code maintenance and errors by minimizing the amount of code that needs to be written and maintained.

Function templates do have a few drawbacks, and we would be remiss not to mention them. First, the compiler will create (and compile) a function for each function call with a unique set of argument types. So while function templates are compact to write, they can expand into a crazy amount of code, which can lead to code bloat and slow compile times. The bigger downside of function templates is that they tend to

produce crazy-looking, borderline unreadable error messages that are much harder to decipher than those of regular functions. These error messages can be quite intimidating, but once you understand what they are trying to tell you, the problems they are pinpointing are often quite straightforward to resolve.

These drawbacks are fairly minor compared with the power and safety that templates bring to your programming toolkit, so use templates liberally anywhere you need type flexibility! A good rule of thumb is to create normal functions at first, and then convert them into function templates if you find you need an overload for different parameter types.

> **Best practice**
>
> Use function templates to write generic code that can work with a wide variety of types whenever you have the need.

6

7

2

8

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

```
Leave a comment...
```

👤 Name*

@ Email* | ⑦ | t

🐞 Find a mistake? Leave a comment above!⑦

👤 Avatars from https://gravatar.com/[11] are connected to your provided email address.

Notify me about replies: 🔔

POST COMMENT

153 COMMENTS

Newest ▼

howard roark

🕐 March 27, 2025 11:05 am PDT

as a reference of advanced area, when i delete const char for std::string ,i dont get any errors but if we are promoting delete for c style string we'll get an error. what is the difference of between that two string models in about my question.

👍 0   ↪ Reply

sserium

🕐 February 17, 2025 4:12 pm PST

Could you include a snippet about the priority of how a call is resolved to a function? I understand that non-template functions are preferred over template functions, but then it seems a non-template exact match is preferred over a template promotion. Does the rule then follow: exact match -> promotion -> conversion with template -> non-template being the ordering at each level? Thanks for the great resource.

EDIT: nvm covered in next lesson

✎ *Last edited 4 months ago by sserium*

👍 0   ↪ Reply

rkh

🕐 February 4, 2025 12:59 pm PST

Is a function call the same as a normal function call, e.g. max(1, 2), that instead of a normal function a function template is used for an emplicit instantiation?

Thanks

👍 0   ↪ Reply

> **Alex** `Author`
>
> 💬 Reply to rkh [12]   🕐 February 7, 2025 7:07 pm PST
>
> Using a normal function call syntax (e.g. `max(1, 2)`), the compiler will consider matching both non-template functions (preferred) and function templates. If the function template is selected as the better match, the appropriate specialization will be instantiated if needed.
>
> 👍 0   ↪ Reply
>
> > **rkh**
> >
> > 💬 Reply to Alex [13]   🕐 February 8, 2025 2:44 am PST
> >
> > Thanks
> >
> > I want to know the exact difference between implicit instantiation and specialization

```
1   #include <iostream>
2
3   template <typename T>
4   T max(T x, T y)
5   {
6       return (x < y) ? y : x;
7   }
8
9   int main()
10  {
11      1) max<int>(1, 2)
12      2) max(1, 2)
13
14      return 0;
15  }
```
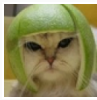
Here, Are the first and second cases examples of specialization and implicit instantiation?

👍 0 　 ➥ Reply

Alex `Author`

💬 Reply to rkh [14] 　 🕐 February 10, 2025 2:50 pm PST

I have a rewrite of this lesson pending to help clarify this. In the meantime:

- The most-generic version of a template (e.g. `T max(T, T)`) is the primary template.
- Anything instantiated from a primary template is calleld a specialization.
- Implicit instantiation is what it is called when the compiler implicitly uses a primary template to produce a specialization (in response to needing a specialization that doesn't exist).

Both 1 and 2 in your program above are just function calls. 1 will implicitly instantiate the specialization `template <> int max(int x, int y)`. 2 will call the specialization (no instantiation happens since the specialization has already been instantiated in this translation unit).

👍 1 　 ➥ Reply

---

rkh

💬 Reply to Alex [15] 　 🕐 February 13, 2025 1:59 pm PST

both of these cases use a generic template defined at the top of the program. since this template is not specialized by a specific type, the functions that are instantiated due to both function calls are implicit instantiation?

👍 0 　 ➥ Reply

Alex `Author`

💬 Reply to rkh [16] 　 🕐 February 13, 2025 9:48 pm PST

Yes, the "generic template" `T max(T x, T y)` is the primary template.

Only one implicit instantiation happens -- when `max<int>(1, 2)` is called. `max(1, 2)` just calls the same function (it is not instantiated again).

rkh

Reply to Alex [17]   February 14, 2025 2:46 am PST

so, can we conclude this point that a specialization `always` happens when a function call is made (such as in cases 1 and 2) since the compiler implicitly uses the generic template to create a specialization? now, the function that is instantiated from this primary template is called specialization.

Alex   Author

Reply to rkh [18]   February 15, 2025 2:15 pm PST

A specialization can result from either an implicit instantiation when a function call is made, or they can be explicitly defined. If a function call is made and the needed specialization already exists in the current translation unit, it is not reinstantiated.

Rajat

December 18, 2024 10:26 pm PST

```cpp
template <typename T>
void print(T x)
{
std::cout<<x;
}
int main(){
double a{5.0};
float b{2.0};
print<double>(a); //instantiates void print(double)
print<>(b); //Does this call the already instantiated double function, or instantiates
a new float function?
}
```

Questions:

1. Since b is float, but can be promoted to double, would it just call the double function?
2. What if it was the case of float and int (same size) instead of double and float (different size), if size is a factor?
3. Would using literals (with suffix) change something?

Alex   Author

Reply to Rajat [19]   December 28, 2024 9:08 pm PST

1. No, it instantiates a float version.

2. Size is not a factor.

3. No.

👍 2   ↪ Reply

KLAP

🕐 October 18, 2024 3:30 pm PDT

I wonder why you don't cover using multiple template parameters like this (I tested and it works):

```cpp
#include <iostream>
#include <string_view>

template <typename T, typename U>
void printIDAndName(T id, U name)
{
    std::cout << "Your name is " << name << ", and your social security is " << id;
}

int main()
{
    std::string_view constexpr name{ "KLAP" };
    auto constexpr id{ 42069 };
    printIDAndName(id, name);

    return 0;
}
```

Or are you expect your students to logically come to conclusion that this is allowed? and people like me need to stop asking for handouts and getting their hand held every single step of the way like babies.

✎ *Last edited 8 months ago by KLAP*

👍 0   ↪ Reply

> **KLAP**
>
> 💬 Reply to **KLAP** [20]   🕐 October 19, 2024 11:40 am PDT
>
> Never mind it's in chapter 11.9. Weird that the next lesson is using template in multiple files and not multiple template types.
>
> 👍 0   ↪ Reply
>
> > **Alex** `Author`
> >
> > 💬 Reply to **KLAP** [21]   🕐 October 20, 2024 3:41 pm PDT
> >
> > Yeah that is weird, isn't it? Rearranged the lessons so we now cover multiple template types and non-type template parameters first.
> >
> > 👍 3   ↪ Reply

qwerty

🕐 September 24, 2024 1:37 am PDT

This doesn't compile on my end

```
template <typename T>
void print(T val, int times=1)
{
while (times--)
{
std::cout << val;
}
}

int main()
{
print(5); // print 5 1 time
print('a', 3); // print 'a' 3 times

return 0;
}
```

returns an error

/usr/bin/ld: /tmp/ccnwmLe8.o: in function `main':

`main.cpp:(.text+0x13): undefined reference to` print(int, int)'

collect2: error: ld returned 1 exit statu

👍 0   ➥ Reply

Alex `Author`

💬 Reply to qwerty [22] 🕐 September 25, 2024 5:14 pm PDT

Compiles fine for me on both GCC and Clang under C++11.

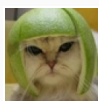👍 0   ➥ Reply

yuhyuh

🕐 August 20, 2024 7:56 am PDT

Hi Alex, it wasn't discussed on this page but I tried default arguements and they seemed to work with templates, is there a reason why it wasn't mentioned, are they unsafe or not best practice?

```
1   #include <iostream>
2
3   template <typename T>
4   T max(T x, int y =5)
5   {
6       return (x < y) ? y : x;
7   }
8
9   int main()
10  {
11      std::cout << max(5.5) << '\n';
12      std::cout << max(4, 3) << '\n';
13
14      return 0;
15  }
```

👍 0   ➤ Reply

Alex `Author`

💬 Reply to yuhyuh [23]   🕐 August 21, 2024 10:04 am PDT

It's fine to use them.

Added a section about this to https://www.learncpp.com/cpp-tutorial/function-template-instantiation/

👍 2   ➤ Reply

Phargelm

🕐 August 11, 2024 6:13 am PDT

My question is regarding this example from the lesson:

```
1   int main()
2   {
3       print<bool>(true); // calls print<bool>(bool) -- prints 1
4       std::cout << '\n';
5
6       print<>(true);      // deduces print<bool>(bool) (non-template functions not
    considered) -- prints 1
7       std::cout << '\n';
8
9       print(true);        // calls print(bool) -- prints true
10      std::cout << '\n';
11
12      return 0;
13  }
```

When we call `print(true)` it calls `print(bool)` but also I see that `print<bool>(bool)` has been called, which means that we already have a function instantiated from the template specifically for the `bool` type. So why calling `print(true);` resolves to the `print(bool)` instead of an ambiguous function call? I understand that template functions have lower priority than non-template functions when we call like `print(true)`, but does it work also for functions that were instantiated from a template already as well? I assume that despite a template function (primary template) having lower priority than the non-template function in this case, the function instance has the same priority as the non-template function.

Alex `Author`
💬 Reply to Phargelm [24] 🕐 August 11, 2024 2:12 pm PDT

You answered your own question: "I understand that template functions have lower priority than non-template functions."

Whether the template has been previously instantiated is irrelevant. Non-template functions take precedence over equally viable template functions.

👍 2 ↳ Reply

april
🕐 July 20, 2024 8:21 am PDT

hey, just letting you know there's an unwanted(?) space in the example for the "Function templates with non-template parameters" section.
it wouldn't actually matter, if i remember correctly? but i thought i'd point it out anyway

```cpp
template <typename T>
int someFcn (T, double) //on this line
{
    return 5;
}
```

✏️ *Last edited 11 months ago by april*

👍 0 ↳ Reply

Alex `Author`
💬 Reply to april [25] 🕐 July 22, 2024 1:22 pm PDT

Removed the extraneous space, but the compiler doesn't care either way.

👍 1 ↳ Reply

april
💬 Reply to april [25] 🕐 July 20, 2024 8:56 pm PDT

oh also, was skipping 11.8 intentional?

👍 0 ↳ Reply

Alex `Author`
💬 Reply to april [26] 🕐 July 22, 2024 1:20 pm PDT

Nope. Readded missing lesson https://www.learncpp.com/cpp-tutorial/using-function-templates-in-multiple-files/

MOEGMA25
🕐 July 6, 2024 8:57 am PDT

```
1  // T is a type template parameter
2  // double is a non-template parameter
3  // We don't need to provide names for these parameters since they aren't used
4  template <typename T>
5  int someFcn (T, double)
6  {
7      return 5;
8  }
9
10 int main()
11 {
12     someFcn(1, 3.4); // matches someFcn(int, double)
13     someFcn(1, 3.4f); // matches someFcn(int, double) -- the float is promoted to a
14 double
15     someFcn(1.2, 3.4); // matches someFcn(double, double)
16     someFcn(1.2f, 3.4); // matches someFcn(float, double)
17     someFcn(1.2f, 3.4f); // matches someFcn(float, double) -- the float is promoted to
18 a double
19
       return 0;
   }
```

I think these comments aren't correct...

👍 0   ↪ Reply

Alex `Author`
💬 Reply to MOEGMA25 [27]   🕐 July 10, 2024 12:29 pm PDT

Why do you think that?

👍 0   ↪ Reply

MOEGMA25
💬 Reply to Alex [28]   🕐 July 12, 2024 2:08 am PDT

My bad, I misunderstood why a 3.4f was said to be a double, but saw that it would be promoted.

👍 0   ↪ 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-specialization/
4. https://www.learncpp.com/cpp-tutorial/pointer-arithmetic-and-subscripting/
5. https://www.learncpp.com/cpp-tutorial/static-local-variables/
6. https://www.learncpp.com/cpp-tutorial/function-templates-with-multiple-template-types/
7. https://www.learncpp.com/
8. https://www.learncpp.com/function-template-instantiation/
9. https://www.learncpp.com/cpp-tutorial/chapter-10-summary-and-quiz/
10. https://www.learncpp.com/cpp-tutorial/introduction-to-c20/
11. https://gravatar.com/
12. https://www.learncpp.com/cpp-tutorial/function-template-instantiation/#comment-607382
13. https://www.learncpp.com/cpp-tutorial/function-template-instantiation/#comment-607491
14. https://www.learncpp.com/cpp-tutorial/function-template-instantiation/#comment-607504
15. https://www.learncpp.com/cpp-tutorial/function-template-instantiation/#comment-607592
16. https://www.learncpp.com/cpp-tutorial/function-template-instantiation/#comment-607708
17. https://www.learncpp.com/cpp-tutorial/function-template-instantiation/#comment-607720
18. https://www.learncpp.com/cpp-tutorial/function-template-instantiation/#comment-607728
19. https://www.learncpp.com/cpp-tutorial/function-template-instantiation/#comment-605436
20. https://www.learncpp.com/cpp-tutorial/function-template-instantiation/#comment-603323
21. https://www.learncpp.com/cpp-tutorial/function-template-instantiation/#comment-603342
22. https://www.learncpp.com/cpp-tutorial/function-template-instantiation/#comment-602292
23. https://www.learncpp.com/cpp-tutorial/function-template-instantiation/#comment-601060
24. https://www.learncpp.com/cpp-tutorial/function-template-instantiation/#comment-600765
25. https://www.learncpp.com/cpp-tutorial/function-template-instantiation/#comment-599880
26. https://www.learncpp.com/cpp-tutorial/function-template-instantiation/#comment-599896
27. https://www.learncpp.com/cpp-tutorial/function-template-instantiation/#comment-599262
28. https://www.learncpp.com/cpp-tutorial/function-template-instantiation/#comment-599404
29. https://g.ezoic.net/privacy/learncpp.com