# 11.9 — Non-type template parameters

ALEX[1] OCTOBER 20, 2024

In the previous lessons, we discussed how to create function templates that use type template parameters. A type template parameter serves as a placeholder for an actual type that is passed in as a template argument.

While type template parameters are by far the most common type of template parameter used, there is another kind of template parameter worth knowing about: the non-type template parameter.

## Non-type template parameters

A **non-type template parameter** is a template parameter with a fixed type that serves as a placeholder for a constexpr value passed in as a template argument.

A non-type template parameter can be any of the following types:

- An integral type
- An enumeration type
- `std::nullptr_t`
- A floating point type (since C++20)
- A pointer or reference to an object
- A pointer or reference to a function
- A pointer or reference to a member function
- A literal class type (since C++20)

We saw our first example of a non-type template parameter when we discussed `std::bitset` in lesson O.1 -- Bit flags and bit manipulation via std::bitset (https://www.learncpp.com/cpp-tutorial/bit-flags-and-bit-manipulation-via-stdbitset/)[2]:

```cpp
#include <bitset>

int main()
{
    std::bitset<8> bits{ 0b0000'0101 }; // The <8> is a non-type template parameter

    return 0;
}
```

In the case of `std::bitset`, the non-type template parameter is used to tell the `std::bitset` how many bits we want it to store.

## Defining our own non-type template parameters

Here's a simple example of a function that uses an int non-type template parameter:

```
 1  #include <iostream>
 2
 3  template <int N> // declare a non-type template parameter of type int named N
 4  void print()
 5  {
 6      std::cout << N << '\n'; // use value of N here
 7  }
 8
 9  int main()
10  {
11      print<5>(); // 5 is our non-type template argument
12
13      return 0;
14  }
```

This example prints:

```
5
```

On line 3, we have our template parameter declaration. Inside the angled brackets, we're defining a non-type template parameter named `N` that will be a placeholder for a value of type `int`. Inside the `print()` function, we use the value of `N`.

On line 11, we have our call to function `print()`, which uses int value `5` as the non-type template argument. When the compiler sees this call, it will instantiate a function that looks something like this:

```
template <>
void print<5>()
{
    std::cout << 5 << '\n';
}
```

At runtime, when this function is called from `main()`, it prints `5`.

Then the program ends. Pretty simple, right?

Much like `T` is typically used as the name for the first type template parameter, `N` is conventionally used as the name of an int non-type template parameter.

### Best practice

Use `N` as the name of an int non-type template parameter.

## What are non-type template parameters useful for?

As of C++20, function parameters cannot be constexpr. This is true for normal functions, constexpr functions (which makes sense, as they must be able to be run at runtime), and perhaps surprisingly, even consteval functions.

So let's say we have some function like this one:

```
#include <cassert>
#include <cmath> // for std::sqrt
#include <iostream>

double getSqrt(double d)
{
    assert(d >= 0.0 && "getSqrt(): d must be non-negative");

    // The assert above will probably be compiled out in non-debug builds
    if (d >= 0)
        return std::sqrt(d);

    return 0.0;
}

int main()
{
    std::cout << getSqrt(5.0) << '\n';
    std::cout << getSqrt(-5.0) << '\n';

    return 0;
}
```

When run, the call to `getSqrt(-5.0)` will runtime assert out. While this is better than nothing, because `-5.0` is a literal (and implicitly constexpr), it would be better if we could static_assert so that errors such as this one would be caught at compile-time. However, static_assert requires a constant expression, and function parameters can't be constexpr…

However, if we change the function parameter to a non-type template parameter instead, then we can do exactly as we want:

```
#include <cmath> // for std::sqrt
#include <iostream>

template <double D> // requires C++20 for floating point non-type parameters
double getSqrt()
{
    static_assert(D >= 0.0, "getSqrt(): D must be non-negative");

    if constexpr (D >= 0) // ignore the constexpr here for this example
        return std::sqrt(D); // strangely, std::sqrt isn't a constexpr function (until
C++26)

    return 0.0;
}

int main()
{
    std::cout << getSqrt<5.0>() << '\n';
    std::cout << getSqrt<-5.0>() << '\n';

    return 0;
}
```

This version fails to compile. When the compiler encounters `getSqrt<-5.0>()`, it will instantiate and call a function that looks something like this:

```cpp
template <>
double getSqrt<-5.0>()
{
    static_assert(-5.0 >= 0.0, "getSqrt(): D must be non-negative");

    if constexpr (-5.0 >= 0) // ignore the constexpr here for this example
        return std::sqrt(-5.0);

    return 0.0;
}
```

The static_assert condition is false, so the compiler asserts out.

**Key insight**

Non-type template parameters are used primarily when we need to pass constexpr values to functions (or class types) so they can be used in contexts that require a constant expression.

The class type `std::bitset` uses a non-type template parameter to define the number of bits to store because the number of bits must be a constexpr value.

**Author’s note**

Having to use non-type template parameters to circumvent the restriction that function parameters can’t be constexpr isn’t great. There are quite a few different proposals being evaluated to help address situations like this. I expect that we might see a better solution to this in a future C++ language standard.

## Implicit conversions for non-type template arguments Optional

Certain non-type template arguments can be implicitly converted in order to match a non-type template parameter of a different type. For example:

```cpp
#include <iostream>

template <int N> // int non-type template parameter
void print()
{
    std::cout << N << '\n';
}

int main()
{
    print<5>();   // no conversion necessary
    print<'c'>(); // 'c' converted to type int, prints 99

    return 0;
}
```

This prints:

```
5
99
```

In the above example, `'c'` is converted to an `int` in order to match the non-type template parameter of function template `print()`, which then prints the value as an `int`.

In this context, only certain types of constexpr conversions are allowed. The most common types of allowed conversions include:

- Integral promotions (e.g. `char` to `int`)
- Integral conversions (e.g. `char` to `long` or `int` to `char`)
- User-defined conversions (e.g. some program-defined class to `int`)
- Lvalue to rvalue conversions (e.g. some variable `x` to the value of `x`)

Note that this list is less permissive than the type of implicit conversions allowed for list initialization. For example, you can list initialize a variable of type `double` using a `constexpr int`, but a `constexpr int` non-type template argument will not convert to a `double` non-type template parameter.

The full list of allowed conversions can be found here (https://en.cppreference.com/w/cpp/language/constant_expression)[3] under the subsection “Converted constant expression”.

Unlike with normal functions, the algorithm for matching function template calls to function template definitions is not sophisticated, and certain matches are not prioritized over others based on the type of conversion required (or lack thereof). This means that if a function template is overloaded for different kinds of non-type template parameters, it can very easily result in an ambiguous match:

```
#include <iostream>

template <int N> // int non-type template parameter
void print()
{
    std::cout << N << '\n';
}

template <char N> // char non-type template parameter
void print()
{
    std::cout << N << '\n';
}

int main()
{
    print<5>();   // ambiguous match with int N = 5 and char N = 5
    print<'c'>(); // ambiguous match with int N = 99 and char N = 'c'

    return 0;
}
```

Perhaps surprisingly, both of these calls to `print()` result in ambiguous matches.

## Type deduction for non-type template parameters using `auto` C++17

As of C++17, non-type template parameters may use `auto` to have the compiler deduce the non-type template parameter from the template argument:

```cpp
#include <iostream>

template <auto N> // deduce non-type template parameter from template argument
void print()
{
    std::cout << N << '\n';
}

int main()
{
    print<5>();   // N deduced as int `5`
    print<'c'>(); // N deduced as char `c`

    return 0;
}
```

This compiles and produces the expected result:

```
5
c
```

## For advanced readers

You may be wondering why this example doesn't produce an ambiguous match like the example in the prior section. The compiler looks for ambiguous matches first, and then instantiates the function template if no ambiguous matches exist. In this case, there is only one function template, so there is no possible ambiguity.

After instantiating the function template for the above example, the program looks something like this:

```cpp
#include <iostream>

template <auto N>
void print()
{
    std::cout << N << '\n';
}

template <>
void print<5>() // note that this is print<5> and not print<int>
{
    std::cout << 5 << '\n';
}

template <>
void print<'c'>() // note that this is print<`c`> and not print<char>
{
    std::cout << 'c' << '\n';
}

int main()
{
    print<5>();   // calls print<5>
    print<'c'>(); // calls print<'c'>

    return 0;
}
```

## Quiz time

### Question #1

Write a constexpr function template with a non-type template parameter that returns the factorial of the template argument. The following program should fail to compile when it reaches `factorial<-3>()`.

```cpp
// define your factorial() function template here

int main()
{
    static_assert(factorial<0>() == 1);
    static_assert(factorial<3>() == 6);
    static_assert(factorial<5>() == 120);

    factorial<-3>(); // should fail to compile

    return 0;
}
```

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

Next lesson

11.10 Using function templates in multiple files

5

Back to table of contents

6

Previous lesson

11.8 Function templates with multiple template types

7

8

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

160 COMMENTS Newest

Jacob
June 23, 2025 9:11 pm PDT

Just wanted to let you know that I copied the solution you provided for the quiz question and tried running it in Visual Studio (With C++20 of course) and it gave me this exact error message: "static_assert failed: 'N >=0'

Edit: I now realized that the program is supposed to intentionally render that specific error. I thought it was just supposed to run smoothly, and was frustrated that it kept rendering that error. My apologies.

*Last edited 4 days ago by Jacob*

0 Reply

RURU
June 17, 2025 8:31 am PDT

```
#include<iostream>

template <int N>

constexpr int factorial()
{   static_assert (N>=0 ,"WE CANT FIND THE FACTORIAL OF A NEGATIVE FUNCTION" );
    if (N==0)
    {
        return 1;
    }
    return N*factorial<N-1>();
}
```

0 Reply

Cling
Reply to RURU [12] June 20, 2025 4:17 am PDT

Good Intuition, but a small mistake even did I make:
When we are using recursion, even though we've set the condition if(N==0) return 1; This won't suffice since we are doing compile time eval, that means the compiler will have to go through all the lines on our function even if the return statement is hit. That means the statement- return 0*factorial(0-1) is still there, compiled, which results in our static assert to hit. The possible solution can be to just write your code as:-

```
#include <iostream>
template <int N>
constexpr int factorial()
{
    static_assert(N >= 0, "Nawt Possible my fren...");
    if constexpr(N == 0) return 1;
    else return N * factorial<N-1>();
}
// define your factorial() function template here

int main()
{
    static_assert(factorial<0>() == 1);
    static_assert(factorial<3>() == 6);
    static_assert(factorial<5>() == 120);

    // factorial<-3>(); // should fail to compile

    return 0;
}
```

else statement ensures factorial(-1) never compiles...

0 Reply

RURU

Reply to Cling 13 June 20, 2025 6:28 am PDT

Damn didn't thought of it ty bruv

1 Reply

Jacob

May 13, 2025 11:59 am PDT

```cpp
//factorial.h
#pragma once


template<int N>
constexpr int factorial() {

    static_assert(N >= 0, "factorial(): N must be nonnegative");

    if constexpr (N < 2) return 1;
    else return N * factorial<N - 1>();
}


//main.cpp
#include <iostream>
#include "factorial.h"

int main()
{
    static_assert(factorial<0>() == 1);
    static_assert(factorial<3>() == 6);
    static_assert(factorial<5>() == 120);

    //factorial<-3>(); // should fail to compile

    return 0;
}
```

:D

*Last edited 1 month ago by Jacob*

2 Reply

Copernicus
May 13, 2025 2:31 am PDT

Question #1
Took me a couple hours to complete. I get a warning error "Ill-defined for-loop. Loop body not executed.". Which I think means the loop isn't executed when N == 0, which is what I want, so I don't know why I get this warning. Maybe @Alex can explain?

```cpp
#include <iostream>

template <int N>
constexpr int findFactorial()
{
    //Handle negative numbers
    static_assert(N >= 0);

    //Handle 0
    if constexpr (N == 0)
        return 1;

    //Handle rest of the cases
    int total{ N };
    for (int counter{ N - 1 }; counter >= 1; --counter)
    {
        total *= counter;
    }

    return total;
}

int main()
{
    static_assert(findFactorial<0>() == 1);
    static_assert(findFactorial<3>() == 6);
    static_assert(findFactorial<5>() == 120);

    //findFactorial<-3>(); // should fail to compile

    return 0;
}
```

0 Reply

**Tyler S**

Reply to Copernicus 14 May 20, 2025 6:48 am PDT

A bit late to this, but I believe you are correct in your assumption that the loop is unreachable in some instances. This warning may or may not be thrown depending on the version of your compiler. Mine ran this code just fine until I added the code "std::cout << findFactorial<0>() << '\n';". Aside from the compiler's version, it really comes down to your usage of N to initialize counter. If anyone finds something wrong with this explanation, please feel free to correct me!

*Last edited 1 month ago by Tyler S*

0 Reply

**Copernicus**

Reply to Tyler S 15 May 20, 2025 4:11 pm PDT

Hello and thanks for the reply. I too only get the warning when I explicitly call the function with 0.

0 Reply

Alex S

**Alex S**

May 3, 2025 9:20 am PDT

"As of C++17, non-type template parameters may use auto to have the compiler deduce the non-type template parameter from the template argument"

Does this always work? Could you have an example with three calls foo<x>(), foo<y>(), foo<z>() where the third produces an ambiguous match with the first two created instances?

In line with how resolution works for overloaded type template parameters, I anticipate this would probably depend on what the rules were for more and less "restrictive" matches: the confusion would come if there was no match which was more or less restrictive than the other. I'm not sure this is possible for the types you've presented so far, as they all seem to be arranged in an exclusive hierarchy.

0 Reply

Alex S

Alex S

Reply to Alex S 16 May 3, 2025 11:07 am PDT

I misunderstood the explanation. Here's me putting the explanation in my words:

- When a function template is called with a template parameter, the compiler looks up *primary templates* which match the call. It does not look for function instances.
- There is one function template: `template <auto N>...`. If is only one primary template there cannot be an ambiguous match.

If the compiler *did* try to resolve the third call to one of the two existing function instances, then I believe it could have the problem I described. Like this example from the quiz:

```
void print(long x)
{
    std::cout << "long " << x << '\n';
}

void print(double x)
{
    std::cout << "double " << x << '\n';
}

int main()
{
    print(5);

    return 0;
}
```

Where the two existing overloads print(long x) and print(double x) are instead function instances generated by two previous calls.

But function template resolution doesn't work like that.

*Last edited 1 month ago by Alex S*

0 Reply

Bob

March 27, 2025 6:27 am PDT

The quiz asks us to write a constexpr function which hasn’t been covered yet. This is probably due to rearranging of topics.

1 Reply

smart

Reply to Bob 17 March 27, 2025 8:08 am PDT

I don't know when you started reading these lessons, but in 5.6 there is a section called “A brief introduction to constexpr functions”.

0 Reply

smart

March 26, 2025 4:09 pm PDT

At first I thought I could do this question in 15 minutes, but it took me about 4 hours. But it was a pretty enjoyable 4 hours.

```
template <int N>
constexpr int factorial()
{
    static_assert(N >= 0, "No factorial for negatives.");
    int factorial {N * (N - 1)};
    for (int counter {N - 2}; counter > 0; --counter)
    {
        factorial *= counter;
    }
    return factorial ? factorial : 1;
}

int main()
{
    static_assert(factorial<0>() == 1);
    static_assert(factorial<3>() == 6);
    static_assert(factorial<5>() == 120);

    factorial<-3>();

    return 0;
}
```

3 Reply

SonnyP

March 7, 2025 9:21 am PST

C2338 static_assert failed: 'N >= 0'

When I build this, I get an error. What's going on?

2 Reply

Bob

Reply to SonnyP [18] March 27, 2025 6:30 am PDT

The last factorial call is calling a negative 3, thus failing the assert.

1 Reply

Andrea

February 23, 2025 10:14 am PST

Hi Alex, I have a doubt: if I write

```
std::cout << factorial<10>() << '\n';
```

is factorial<10> actually calculated at compile time? and will therefore reduce the run time? and if so, how deep can the computation go in compile time?

0 Reply

Alex Author

Reply to Andrea [19] March 5, 2025 5:28 pm PST

`factorial<10>` may or may not be called at compile-time, depending on whether the optimizer decides to do so. You can force it to evaluate at compile-time by using the result to initialize a constexpr variable, or via of the methods described in https://www.learncpp.com/cpp-tutorial/constexpr-functions-part-3-and-consteval/

I'm not sure how deep recursion can go -- this is likely compiler-specific.

2 Reply

Adam

February 2, 2025 6:43 am PST

Maybe it is just me, but the code examples seem to be missing an important detail. I only see the following

```
template  // deduce non-type template parameter from template argument
void print()
{
    std::cout << N << '\n';
}

int main()
{
    print<5>();   // N deduced as int `5`
    print<'c'>(); // N deduced as char `c`

    return 0;
}
```

From the comments below, it seems that one is missing `<int N>` next to the `template`

0 Reply

Alex Author

Reply to Adam [20] February 3, 2025 1:20 am PST

I see it as having an `<auto N>`. I've received several comments about similar things though but I haven't actually seen it happen myself. Not sure how this could even happen.

1 Reply

## Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/bit-flags-and-bit-manipulation-via-stdbitset/
3. https://en.cppreference.com/w/cpp/language/constant_expression
4. javascript:void(0)
5. https://www.learncpp.com/cpp-tutorial/using-function-templates-in-multiple-files/
6. https://www.learncpp.com/
7. https://www.learncpp.com/cpp-tutorial/function-templates-with-multiple-template-types/
8. https://www.learncpp.com/non-type-template-parameters/
9. https://www.learncpp.com/cpp-tutorial/chapter-21-project/
10. https://www.learncpp.com/cpp-tutorial/narrowing-conversions-list-initialization-and-constexpr-initializers/
11. https://gravatar.com/
12. https://www.learncpp.com/cpp-tutorial/non-type-template-parameters/#comment-610977
13. https://www.learncpp.com/cpp-tutorial/non-type-template-parameters/#comment-611034
14. https://www.learncpp.com/cpp-tutorial/non-type-template-parameters/#comment-609983
15. https://www.learncpp.com/cpp-tutorial/non-type-template-parameters/#comment-610231
16. https://www.learncpp.com/cpp-tutorial/non-type-template-parameters/#comment-609741
17. https://www.learncpp.com/cpp-tutorial/non-type-template-parameters/#comment-608833
18. https://www.learncpp.com/cpp-tutorial/non-type-template-parameters/#comment-608327
19. https://www.learncpp.com/cpp-tutorial/non-type-template-parameters/#comment-607995
20. https://www.learncpp.com/cpp-tutorial/non-type-template-parameters/#comment-607286
21. https://g.ezoic.net/privacy/learncpp.com