11.9 — Non-type template parameters

learncpp.com/cpp-tutorial/non-type-template-parameters/

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:

```
#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:

```
#include <iostream>

template <int N> // declare a non-type template parameter of type int named N
void print()
{
    std::cout << N << '\n'; // use value of N here
}

int main()
{
    print<5>(); // 5 is our non-type template argument

    return 0;
}
```

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:

```
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:

```
#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 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:

```
#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:

```
#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>()`.

```
// 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