template <auto>

Posted on May 23, 2018

by Marius Bancila (https://mariusbancila.ro/blog/author/Admin/)

If you wanted to create templates with non-type template parameters, you had to specify both the type and the value. In C++17, this is no longer the case, as `template <auto>` helps simplify these scenarios.

Let's take as an example the declaration of a constant template.

```C++
template <typename T, T value>
constexpr T numeric_constant = value;

constexpr auto const the_answer = numeric_constant<int, 42>;
```

In C++17, this can be simplified as follows:

```C++
template <auto value>
constexpr auto numeric_constant = value;

constexpr auto const the_answer = numeric_constant<42>;
```

You no longer need to specify the type for non-type template parameters, it is automatically deduced by the compiler. Here is another example:

```cpp
template <auto value>
void foo() { /*... */ }

foo<42>();      // deduces int
foo<false>();   // deduces bool
```

Let's look at an example where `template <auto>` can simplify the code. The standard defines a type called `std::integral_constant` that wraps a static constant of a specified type and is the base class for the C++ type traits. `std::true_type` and `std::false_type` are two of those. These could be implemented as follows:

```cpp
template<class T, T val>
struct integral_constant
{
    static constexpr T value = val;

    using value_type = T;
    using type = integral_constant;

    constexpr operator value_type() const noexcept { return value; }

    [[nodiscard]] constexpr value_type operator()() const noexcept { retu
};

template<bool val>
using bool_constant = integral_constant<bool, val>;

using true_type = bool_constant<true>;
using false_type = bool_constant<false>;
```

With `template <auto>` this code can be written as following:

```cpp
template<auto val>
struct integral_constant
{
    static constexpr auto value = val;
```

```
5
6       using value_type = decltype(val);
7       using type = integral_constant;
8
9       constexpr operator value_type() const noexcept { return value; }
10
11      [[nodiscard]] constexpr value_type operator()() const noexcept { retu
12  };
13
14  using true_type = integral_constant<true>;
15  using false_type = integral_constant<false>;
```

Note: Although this works fine with Clang and GCC, VC++ 15.7 complains about the use of `decltype(val)`.

Note: It may be more cumbersome to deduce some types. For instance, if you need a `short` type, there is no way to specify a `short` literal. You must indicate that with a cast. In order words, if you want a `short` 42 integral constant, you need to declare it like this:

C++

```
1 using short_42 = integral_constant<(short)42>;
```

This feature is also useful with variadic templates for creating compile-time lists of henerogenous values. Here is an example:

C++

```
1 template <auto ... vs>
2 struct heterogenous_list {};
3
4 using ultimate_list = heterogenous_list<42, "the ultimate answer", true>;
```

For more information on this topic, see Declaring non-type template arguments with auto (http://open-std.org/JTC1/SC22/WG21/docs/papers/2016/p0127r1.html).

Print (https://mariusbancila.ro/blog/2018/05/23/template-auto/#print) More

Like this:

Like

Be the first to like this.

C++ (https://mariusbancila.ro/blog/category/it/software/c/)

auto (https://mariusbancila.ro/blog/tag/auto/), C++ (https://mariusbancila.ro/blog/tag/c/), templates (https://mariusbancila.ro/blog/tag/templates/), typetraits (https://mariusbancila.ro/blog/tag/typetraits/)

2 Replies to “template <auto>”

Mihai Todor

July 3, 2018 at 10:58 pm (https://mariusbancila.ro/blog/2018/05/23/template-auto/#comment-420456)

Just curious, is there any benefit to using both `constexpr` and `const` when declaring a variable? I assumed that `constexpr` already implies `const`.

Marius Bancila (https://www.mariusbancila.ro/blog)

July 4, 2018 at 10:16 pm (https://mariusbancila.ro/blog/2018/05/23/template-auto/#comment-420457)

It is indeed redundant. `constexpr` implies `const`.