Deduction Guides

The lesson focuses on how compiler uses special rules called “Deduction Guides” to work out the template class types.

WE'LL COVER THE FOLLOWING

- Deduction Guides
  - CTAD Limitations

Deduction Guides

The compiler uses special rules called “*Deduction Guides*” to work out the template class types. We have two types of guides:

1. compiler-generated (implicitly generated)
2. user-defined

To understand how the compiler uses the guides, let’s look at an example. Here’s a custom deduction guide[^arrded] for `std::array` :

> **[^arrded]:** simplified version of libstdc++

```
template<typename T, typename... U>
array(T, U...) ->
    array<enable_if_t<(is_same_v<T, U> && ...), T>, 1 + sizeof...(U)>;
```

The syntax looks like a template function with a trailing return type.

The compiler treats such “*imaginary*” function as a candidate for the parameters. If the pattern matches, then the proper type is returned from the deduction.

In our case, when you write:

In our case, when you write:

```
std::array arr {1, 2, 3, 4};
```

Then, assuming `T` and `U...` are of the same type, we can build up an array object of type `std::array<int,4>`.

In most cases, you can rely on the compiler to generate automatic deduction guides. They will be created for each constructor (also copy/move) of the primary class template.

> Please note, that classes that are specialized or partially specialized won't work here.

As mentioned, you might also write your own deduction guides:

A classic example of where you might add your custom deduction guides is a deduction of `std::string` rather than `const char*` :

```
template<typename T>
struct MyType
{
  T str;
};

// custom deduction guide
MyType(const char*) -> MyType <string>;
MyType t{"Hello World"}; // deduces std::string
```

Without the custom deduction, `T` would be deduced as `const char*`. Another example of custom deduction guide comes from the `overload` pattern[^overloadpat]:

> **[^overloadpat]:** Read more about this pattern in the chapter about `std::variant`, section related to `std::visit()`

```
template<class... Ts>
struct overload : Ts... { using Ts::operator()...; };
template<class... Ts>
overload(Ts...) -> overload<Ts...>; // deduction guide
```

The `overload` class inherits from other classes `Ts...` and then exposes their `operator()`. The custom deduction guide is used here to "*transform*" a list of lambdas into the list of classes that we can derive from.

CTAD Limitations

In C++17 template argument deduction for classes has the following limitations:

- it doesn't work with template aggregate types
- the deduction doesn't include inheriting constructors
- it doesn't work with template aliases

Those limitations will be removed in C++20 through the already accepted proposal: P1021.

> **Extra Info:** The CTAD feature was proposed in: P0091R3 and P0433 - Deduction Guides in the Standard Library

> **Please note:** While a compiler might declare full support for Template Argument Deduction for Class Templates, its corresponding STL implementation might still lack custom deduction guides for some STL types. Compiler Support is mentioned at the end of the chapter.

Head over to the next lesson to find out about variadic templates in C++ 17.