Why isn’t C++ using my default parameter to deduce a template type?

devblogs.microsoft.com/oldnewthing/20240325-00

March 25, 2024

Raymond Chen

Consider the following function:

```
template<typename T>
void Increment(T value = 1);
```

If you call it with no arguments, C++ cannot deduce the template type parameter:

```
// clang
error: no matching function for call to 'Increment'
couldn't infer template argument 'T'

// gcc
error: no matching function for call to 'Increment()'
couldn't deduce template parameter 'T'

// msvc
error C2672: 'Increment': no matching overload function found
could not deduce template argument for 'T'
```

“What do you mean, you can’t deduce the template argument? It’s right there: The default parameter is `1`, so `T` should be `int`!”

The catch is that C++ does not consider default parameters when performing template deduction. It performs deduction only with explicitly-provided parameters.

A very common pattern is for the default value of a function parameter to be dependent upon a template type parameter.

```
template<typename T>
void SetSize(size_t N, T value = T{});
```

The intention here is that if you call `SetSize` and don’t provide a value, we will use a default-constructed value. You see this pattern, for example, in `std::vector::resize()`.

If default parameters were used by type inference, then you'd have a circular dependency:[1] To figure out what `T` is, you need to know the default parameter, but to know the default parameter, you need to know what `T` is.[2]

But all is not lost. If you want to provide a default type for `T`, you can provided it as part of the template type parameter list:

```
template<typename T = int>
void Increment(T value = 1);
```

If you call `Increment()` with no parameters, the compiler cannot deduce `T` from the actual parameters (since there are none), so it uses the default type in the template type parameter list, and you get `int`. Once that's decided, the default parameter is then implicitly converted from `1`.

[1] I wouldn't be surprised if there turns out to be a circular dependency with overload resolution, but I am too tired to try to work out the details.

[2] You might say, "Sure, that general case doesn't work, but certainly you can make it work in this case!" There is a trade-off when you add rules to handle special cases. The good part is that more cases work. The bad part is that you now have more rules. Often, it's better to have a simple set of easily-understood and easily-remembered rules, even if they don't cover all possible cases.[3]

[3] One might argue that the C++ committee had already abandoned the principle of "fewer rules that are easier to remember" a long time ago. For example, C++20 made `typename` optional in a half dozen special cases.