Minimal fix for CWG3003 (CTAD from template template parameters)

Document #:	P3683R0
Date:	2025-10-06
Programming Language C++	
Audience:	EWG

Reply-to: Corentin Jabot <corentin.jabot@gmail.com>

Abstract

The standard library depends on CTAD from template template parameters. However, this feature does not exist despite being universally implemented.

History

C++23 added a facility to build a container out of a range, including conversions:

```
std::views::iota(0, 42) | std::ranges::to<std::vector<int>>();
std::views::iota(0, 42) | std::ranges::to<std::vector<double>>()
```

In addition, we allow deducing the element type from the element type of the view.

```
std::views::iota(0, 42) | std::ranges::to<std::vector>();
```

It's not magic, it's CTAD!

Ignoring a lot of details, it boils down to

```
template <template <typename...> typename TT, typename R>
auto to(R&& r) {
    return std::ranges::to<decltype(TT(std::from_range, std::declval<R&&>()))>(r);
}
```

This is a neat trick that I think was first implemented in ranges-v3. It is widely used by users of the C++ standard library and other ranges libraries, and supported by all implementations.

There is just one problem: Nowhere does the standard allow performing CTAD for template template parameters. We standardized a library feature that is simply not C++. And by we, I mean me (P1206R7 [1]).

LWG4381 proposes to remove the offending `ranges::to` overload to fix the fact that the library relies on imaginary features. However, this is not really possible. Regardless of whether the feature is conforming or useful, it is widely used and universally implemented.

The solution is for the standard to reflect existing practice.

Aliases

When aliases template are involved, not all scenarios are supported. It seems important to support the pmr use case, which all implementations do

```
template <template <typename...> typename TT, typename... Args>
auto f(Args&&... args) {
    return TT(args...);
}
template <typename T>
struct alloc;

template <typename T>
struct pmr_alloc;

template <typename T, typename Alloc = alloc<T>>
struct vector {
    vector(T);
};

template <typename T>
using pmr_vector = vector<T, pmr_alloc<T>>;

auto b = f<vector>(0);
auto a = f<pmr_vector>(0);
```

However, no implementations support multiple layers of aliases, as described in CWG3003:

```
template <typename T> struct A { A(T); };

template <typename T, template <typename> class TT = A>
using Alias = TT<T>;

template <typename T>
using Alias2 = Alias<T>;

void h() { Alias2 a(42); } // all reject
void h2() { Alias a(42); } // all reject
```

No implementation supports this example ([Compiler explorer]). In fact, Clang and GCC used to crash on that scenario, which is how this imbroglio came to light in the first place.

Supporting that seems less useful, however, it requires additional wording mechanism (CTAD for alias template is not exactly simple), and ideally, some prototyping. We therefore propose that this remains unsupported and be left as a future evolutionary exercise for an eager reader.

Feature test macro

Because this specifies existing practice, there is no need to update any feature test macros. The change should be a DR to C++23.

Wording

[dcl.type.simple] Simple type specifiers

A *placeholder-type-specifier* is a placeholder for a type to be deduced [dcl.spec.auto]. A *type-specifier* is a placeholder for a deduced class type [dcl.type.class.deduct] if either

- it is of the form `typename`$_{opt}$ *nested-name-specifier*$_{opt}$ *template-name* or
- it is of the form `typename`$_{opt}$ *splice-specifier* and the *splice-specifier* designates a class template or alias template.

The *nested-name-specifier* or *splice-specifier*, if any, shall be non-dependent and the *template-name* or *splice-specifier* shall designate a deducible template. A *deducible template* is either a class template, a type template template parameter, or is an alias template whose *defining-type-id* is of the form

`typename`$_{opt}$ *nested-name-specifier*$_{opt}$ `template`$_{opt}$ *simple-template-id*

where the *nested-name-specifier* (if any) is non-dependent and the *template-name* of the *simple-template-id* names a deducible template that is not a type template template parameter. [*Note:* An injected-class-name is never interpreted as a *template-name* in contexts where class template argument deduction would be performed [temp.local]. — *end note*]

[over.match.class.deduct] Class template argument deduction

When resolving a placeholder for a deduced class type [dcl.type.class.deduct] where the *template-name* or *splice-type-specifier* designates a primary class template `C`, a set of functions and function templates, called the guides of `C`, is formed comprising:

- If `C` is defined, for each constructor of `C`, a function template with the following properties:
  - The template parameters are the template parameters of `C` followed by the template parameters (including default template arguments) of the constructor, if any.
  - The associated constraints[temp.constr.decl] are the conjunction of the associated constraints of `C` and the associated constraints of the constructor, if any. [*Note:* A *constraint-expression* in the *template-head* of `C` is checked for satisfaction before any constraints from the *template-head* or trailing *requires-clause* of the constructor. — *end note*]
  - The *parameter-declaration-clause* is that of the constructor.
  - The return type is the class template specialization designated by `C` and template arguments corresponding to the template parameters of `C`.

When resolving a placeholder for a deduced class type [dcl.type.class.deduct] where

- the *template-name* designates a type template template parameter `TT`,
- the argument corresponding to `TT` designates a primary class template `C`, and
- `TT` and `C` are compatible [temp.arg.template],

then `C` is first substituted into `TT`.

[temp.dep.type] **Dependent types**

A placeholder for a deduced class type [dcl.type.class.deduct] is dependent if

- it has a dependent initializer, ~~or~~
- it refers to a template template parameter, or
- it refers to an alias template that is a member of the current instantiation and whose *defining-type-id* is dependent after class template argument deduction[over.match.class.deduct] and substitution[temp.alias].

Acknowledgnments

Thanks to Jan Schultke, Lénárd Szolnoki, Tomasz Kamiński, and Matheus Izvekov for their feedback on this paper.

References

[1] Corentin Jabot, Eric Niebler, and Casey Carter. P1206R7: Conversions from ranges to containers. https://wg21.link/p1206r7, 1 2022.

[N5008] Thomas Köppe *Working Draft, Standard for Programming Language C++* https://wg21.link/N5008