Class template argument deduction (CTAD) for type template template parameters

Christof Meerwald

NVIDIA

Document Number: P3865R3
Date: 2026-03-26
Reply-To: Christof Meerwald <cmeerw@cmeerw.org>
Audience: EWG, CWG

Abstract

This paper proposes to update the core language wording to allow CTAD (class template argument deduction) for type template template parameters and treat it as a DR, addressing both CWG 3003 and LWG 4381.

Changelog

R2 → R3
: address issue with non-deducible type template template argument

R1 → R2
: EWG approved
: CWG wording review

R0 → R1
: added discussion about constant template parameter conversion
: added wording change for temp.dep.type from P3863R0

Introduction

CWG 3003 "Naming a deducible template for class template argument deduction" has been raised to clarify that with the current core language wording, class template argument deduction does not work for type template template parameters.

However, the library makes use of this feature in [range.utility.conv.to] (adopted in C++23):

```
template<template<class...> class C, input_range R, class... Args>
  constexpr auto to(R&& r, Args&&... args);
```

...

Let *DEDUCE_EXPR* be defined as follows:

— `C(declval<R>(), declval<Args>()...)` if that is a valid expression,

— ...

and all current implementations actually accept it. There is no known fix for the library wording without changes to the core language, see LWG 4381 "std::ranges::to specification using CTAD not supported by core language".

In Re: [isocpp-core] Review of CWG3003 Naming a deducible template for class template argument deduction it was pointed out that it was not the design intent of P0091R3 "Template argument deduction for class templates (Rev. 6)" to disallow CTAD for template template parameters. It should therefore be seen as fixing a wording defect instead of a new feature.

Examples

Simple case:

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

template<template<typename> class X>
void f() {
  X x(1);
}

template void f<C>();
```

It seems reasonable for the type of `x` to be deduced as `C<int>`.

But since P0552R0 "DR: Matching of template template-arguments excludes compatible templates", a template template argument doesn't have to match a template template parameter exactly; the template template parameter just needs to be more specialized than the template template argument. For example:

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

template<template<typename> class X>
void f() {
  X x1{1};
  X x2{1, 2};
}

template void f<C>();
```

Again, allowing the type of `x1` to be deduced as `C<int>` seems reasonable, but what about `x2`? If we just substitute `C` into the template and then perform class template argument deduction, we would get `C<int, int>`, even though there is no way to actually specify those template arguments with `X`. That seems unhelpful, so that should be ill-formed, similar to CTAD for alias templates (which adds a deducible constraint).

How should default template arguments for the template template parameter be handled?

```
template<typename T = int>
struct C {
  C(int);
};

template<template<typename = long> class X>
void f() {
  X x(1);
}

template void f<C>();
```

A simple substitution into the template would result in the type of `x` to be deduced as `C<int>`, but `X<>` would use the default argument specified in the template template parameter (i.e., the type would instead be `C<long>`).

Furthermore, looking at a case where there is a conversion for constant template parameters:

```
template<int>
struct A { };
```

```
template<int I>
struct C {
  C(A<I>);
};

template<template<short> class X>
void f() {
  X x1{A<1>()};
  X x2{A<100000>()};
}

template void f<C>();
```

Once again, by simply substituting `X` with `C`, the type deduced for `x2` would not be representable in terms of `X`.

CWG 3003 shows another example where the alias template refers to a template template parameter.

```
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); }
void h2() { Alias a(42); }
```

This currently crashes most implementations, but as the *defining-type-id* of Alias doesn't denote a deducible template, it should just be ill-formed.

Status Quo

Current implementations accept the above examples (except for the last one) with the "simple substitution" semantics. This means that default template arguments specified on the template template parameters are ignored during deduction and that types can be deduced that can't otherwise be written using the name of the template template parameter. This seems confusing.

Proposed Semantics

This paper proposes to respect the default arguments specified on template template parameters and any restrictions resulting from a template template parameter being more specialized than the template template argument. This is consistent with how CTAD for alias templates works, and the proposal is to use those semantics instead of directly substituting the template template argument.

Placeholder deduction for a type template template parameter should be performed as if the type template template parameter were replaced by an alias template (with the same template parameters) denoting the type template template argument in its *defining-type-id*.

For example, for the pack example above:

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

template<template<typename> class X>
void f() {
  X x1{1};
  X x2{1, 2};
}
```

```
template<typename T>
using XC = C<T>;  // exposition only

template<> void f<C>() {
  XC x1{1};       // OK, deduces C<int>
  XC x2{1, 2};    // error: class template argument deduction fails
}
```

Similarly, for default template arguments:

```
template<typename T = int>
struct C {
  C(int);
};

template<template<typename = long> class X>
void f() {
  X x{1};
}

template<typename T = long>
using XC = C<T>;  // exposition only

template<> void f<C>() {
  XC x{1};        // OK, deduces C<long>
}
```

Using default arguments from the template template parameter instead of deducing an empty pack:

```
template<typename ... Ts>
struct Y {
  Y();
  Y(Ts ...);
};

template<template<typename T = char> class X>
void f() {
  X x0{};
}

template<typename T = char> using XY = Y<T>;

template<> void f<Y>() {
  XY x0{}; // OK, deduces Y<char>
}
```

The following steps are performed during class template argument deduction:

— Implicit deduction guides:
`template<typename ... Ts> Y() -> Y<Ts ...>`
`template<typename ... Ts> Y(Ts ...) -> Y<Ts ...>`

— Deduce `Y<Ts ...>` from `Y<T>` and substitute into the deduction guides

— Form `f'` for each guide:
`template<typename T = char> auto f_p() -> Y<T>`
`template<typename T = char> auto f_p(T) -> Y<T>`

— Perform template argument deduction (from an empty argument list): → `Y<char>`

— Perform the deducible check

And finally, when a conversion is required for a constant template parameter, we would get:

```
template<int>
```

```
struct A { };

template<int I>
struct C {
  C(A<I>);
};

template<template<short> class X>
void f() {
  X x1{A<1>()};
  X x2{A<100000>()};
}

template<short S>
using XC = C<S>;  // exposition only

template<> void f<C>() {
  XC x1{A<1>()};
  XC x2{A<100000>()};
}
```

Both cases would currently be ill-formed due to the current rules for CTAD for alias templates, i.e., they would both fail the deducible check (at least with the direction for CWG 2467 "CTAD for alias templates and the deducible check"). But this looks like something that should be addressed by a future revision of P3579R0 "Fix matching of non-type template parameters when matching template template parameters"

Implementation Experience

None yet for the exact semantics specified in this paper, but all implementations already support the simple case.

Wording (relative to N5032)

Change [dcl.type.simple] paragraph 3

> 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 A 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 other than a type template template parameter of A .
>
> [Note 1: ... — end note]

Insert a new paragraph before [over.match.class.deduct] paragraph 3

When resolving a placeholder for a deduced class type where the *template-name* designates a type template template parameter P, the type template template argument for P shall be a deducible template. Let A be an alias template whose template parameter list is that of P and whose *defining-type-id* is a *simple-template-id* whose *template-name* designates the type template template argument and whose *template-argument-list* is the template argument list of P. A is then used instead of the original *template-name* to resolve the placeholder.

3 When resolving a placeholder for a deduced class type ([dcl.type.simple]) where the *template-name* or *splice-type-specifier* designates an alias template A, ...

Add a new paragraph at the end of [over.match.class.deduct]

[Example:

```
template<typename ... Ts>
struct Y {
  Y();
  Y(Ts ...);
};
template<template<typename T = char> class X>
void f() {
  X x;        // OK, deduces Y<char>
  X x0{};     // OK, deduces Y<char>
  X x1{1};    // OK, deduces Y<int>
  X x2{1, 2}; // error: cannot deduce X<T> from Y<int, int>
}
template void f<Y>();
```

— end example]

Add a new bullet in [temp.dep.type] paragraph 8

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

— it has a dependent initializer, or

— it refers to a type 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]).

Acknowledgements

Corentin Jabot has a very similar paper P3863R0 "Minimal fix for CWG3003 (CTAD from template template parameters)". I also would like to thank James Touton for hallway discussions during breaks and Matheus Izvekov for asking about the interaction with constant template parameters.

References

- Thomas Köppe: N5032 Working Draft, Standard for Programming Language C++
- Hubert Tong: CWG 3003 Naming a deducible template for class template argument deduction
- Jens Maurer: LWG 4381 std::ranges::to specification using CTAD not supported by core language
- James Touton, Hubert Tong: P0522R0 DR: Matching of template template-arguments excludes compatible templates
- Mike Spertus, Faisal Vali, Richard Smith: P0091R3 Template argument deduction for class templates (Rev. 6)

- Corentin Jabot: P3863R0 Minimal fix for CWG3003 (CTAD from template template parameters)
- Matheus Izvekov: P3579R0 Fix matching of non-type template parameters when matching template template parameters
- paper issue 2443 CWG3003 Naming a deducible template for class template argument deduction