

# A Simple Approach to Universal Template Parameters

|                          |                                                                                                                                                                                  |
|--------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Document #:              | P2989R3                                                                                                                                                                          |
| Date:                    | 2026-10-23                                                                                                                                                                       |
| Programming Language C++ |                                                                                                                                                                                  |
| Audience:                | EWG                                                                                                                                                                              |
| Reply-to:                | Corentin Jabot < <a href="mailto:corentin.jabot@gmail.com">corentin.jabot@gmail.com</a> ><br>Gašper Ažman < <a href="mailto:gasper.azman@gmail.com">gasper.azman@gmail.com</a> > |

## Abstract

In [P1985R3 \[3\]](#), we explore universal template parameters (a template parameter that can be a type, non-type, or template name). We proposed allowing the use of universal template parameters in all contexts, which would have added a large amount of complexity to both the specification and implementations for limited benefits. In this paper, we propose a much simpler subset of the feature.

## Revisions

### R3

- Add wording
- Rewrite the section on UTPs in concept definitions
- Add a future work section.

### R2

- Add sections on ABI and injected-class-names

### R1

- Update the implementation experience
- Fix code examples and add compiler explorer links
- Expand the section about deduction of universal template arguments and propose a new solution to deduce the type of value template parameters deduced from a universal template argument.
- Add a CPO example

## R0

- Initial revision

## Difference with [P1985R3](#) [3]

This paper is a follow-up on [P1985R3](#) [3]. The main difference is that we are merely proposing that universal template parameters can be used only as template arguments of other templates. They cannot be used in any other context.

```
template <universal template U>
void f() {
    SomeOtherTemplate<U>(); // ok.
    U foo; // error: U is not used as a template argument.
    U = 0; // error: U is not used as a template argument.
}
```

This change was made to simplify the design as it completely side-step parsing ambiguities, syntax disambiguators, and so on.

Other than this change, the feature and the design are still very much the same, as this feature presents few design options. We present some new design questions not considered in earlier papers and report on early implementation efforts in Clang.

## Examples

We presented motivating examples in [P1985R3](#) [3], but we offer two more here.

### Template CPOs

Applied to a tuple-like `std::get`, can return either the Nth element or the element of a specific type. Indeed, `std::get` has multiple overloads with types and non-types template parameters

```
template<std::size_t I, class... Types >
const auto & get( const tuple<Types...& t ) noexcept;
template<class T, class... Types >
const auto & get( const tuple<Types...& t ) noexcept;

int main() {
    std::tuple t{1, 2.0};
    get<0>(t); // extract the 0th element
    get<double>(t); // extract the element that is a double
}
```

This is great, but it only works for free functions, not CPOs. LEWG wants a CPO alternative to `std::get` as described in [P2769R1](#) [1].

So we can implement `std::get_element<N>(t)`, but not `std::get_element<T>(t)`. The best we could do would be to give that second CPO a different name, maybe `std::get_element_from_type<T>(t)`. Not great, not terrible.

Universal template parameters solve that problem nicely:

```
template <universal template>
inline constexpr int get_element = /*delete*/0;

template <typename T>
inline constexpr __get_element_t::__cpo<T> get_element<T>;

template <std::size_t I>
inline constexpr __get_element_v::__cpo<I> get_element<I>;
```

[\[Full example on Compiler Explorer\]](#).

## Generic rebind

Consider this allocator:

```
template <class T, size_t BlockSize = 42>
class BlockAllocator {
public:
    using value_type = T;
    T* allocate(size_t);
    void deallocate(T*, size_t);
};
```

This allocator cannot be rebound: it provides no rebind nested alias and one of its template arguments is an NTTP. Because of the lack of universal template parameters, `allocator_traits::rebind_alloc` works only with allocators for which all the template parameters are types.

Universal parameters would allow a more generic implementation of `rebind_alloc`:

```
template <typename, typename>
struct rebind_alloc_t;

template <template <class, universal template...> typename Alloc, class T, universal template
... Args, class U>
struct rebind_alloc_t<Alloc<T, Args...>, U> {
    using type = Alloc<U, Args...>;
};

template <class Alloc>
struct allocator_traits {
    template<class T>
    using rebind_alloc = rebind_alloc_t<Alloc, T>::type;
};

using rebound = allocator_traits<BlockAllocator<int>>::rebind_alloc<long>; // now ok
using rebound2 = allocator_traits<BlockAllocator<int, 128>>::rebind_alloc<long>; // ok too
```

[\[Demo on compiler explorer\]](#)

## **ranges::to**

The code :

```
auto v = view | ranges::to<llvm::SmallVector> ();
```

is currently ill-formed because `llvm::SmallVector` has a defaulted NTTP:

```
template <typename T, unsigned N = CalculateSmallVectorDefaultInlinedElements<T>::value>
class SmallVector;
```

This proposal fixes this issue by allowing us to redefine `ranges::to`:

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

Such a change would allow `llvm::SmallVector` and similar classes (e.g., `folly::small_vector` works the same way) to be usable with `ranges::to`.

## **Design**

The design involves as few elements as possible.

- A universal template parameter can be declared in the template-head of a function, class, or variable template.
- The name of a universal template parameter (which is found by unqualified lookup) can only be used as an argument to a template (and nowhere else).
- Universal template parameters can be packs.
- Universal template parameters cannot be defaulted.
- Forwarding a universal template argument to another template combined with partial template specialization is the main mechanism of use of universal template parameters.

## **Partial ordering of universal template parameters**

Anything that can be a template argument can be the argument binding to a universal template parameter, hence the name *universal template parameter*.

Consequently, universal template parameters are less specialized than any other sort of template with which they are compared, which allows specializing an entity based on template parameter kind.

```
template <universal template>
constexpr bool is_variable = false; // #1
template <auto a>
constexpr bool is_variable<a> = true; // #2
```

Here, #2 is more specialized than #1 because a universal template parameter is less specialized than a non-type parameter.

## Deduction of universal template arguments

The guiding principle for this is that whatever the universal template parameter is compared with, that's what it deduces to.

To specify this fully, we have to consider every *kind* of binding.

### typenames

If the corresponding template argument is a type, the corresponding template parameter deduces to that type.

```
template <universal template T>
struct S {};
template <universal template U, template <universal template> typename T>
void f(T<U> x);

using A = S<int>; // T deduces to int
void h1() {
    f(A{}); // calls f<int, S>
}
```

### template names

If the template argument denotes a template name (or concept name), the corresponding universal template parameter is deduced to be a template (or concept) template parameter with the same template head.

```
// S and f from the previous section
using B = S<std::pair>; // T is compared against template <typename T, typename U> struct pair
;
using C = S<std::conditional_t>; // T is a type-alias std::conditional_t
using D = S<std::regular>; // T is a concept-name std::regular
using E = S<std::is_same_v>; // T is a variable-template std::is_same_v

void h2() {
    f(B{}); // calls f<std::pair, S>
    f(C{}); // calls f<std::conditional_t, S>
    f(D{}); // calls f<std::regular, S>
    f(E{}); // calls f<std::is_same_v, S>
}
```

### Expressions

If the template argument is an expression, the corresponding universal template parameter becomes a non-type template parameter. But the question of how the type of these non-type parameters are deduced.

We considered the following options:

- Using `auto` semantics: This would prevent deducing reference types. While the use cases for references in template parameters are somewhat limited, there are use cases, and preventing them would be against the main motivation of universal template parameters, which is to forward arbitrary template arguments. **This is the option we implemented**, but it is also the worst option.
- Using `decltype(auto)` semantics (proposed in R0 of this paper). This works for most cases and is fairly easy to implement.

Consequently, the following would be ill-formed:

```
template <int&> struct A { };
template <universal template U> B : A<U> { };
int x;
B<x> b1;
```

And it makes partial specialization somewhat subtle:

```
static constexpr int a = 0;

template <universal template>
struct S { // #1
    static constexpr int value = 0;
};

template <>
struct S<a> { // #2
    static constexpr int universal = 1;
};

void test() {
    static_assert(S<a>::i == 1); // a is deduced as int
    static_assert(S<(a)>::i == 0); // (a) is deduced as const int& and #2 is
    // specialized for int, so #1 is picked
}
```

- Apply `decltype(auto)` rules but deduce a reference for *id-expressions*. This almost works, except it would create an inconsistency for parameter forwarded to a `decltype(auto)` template parameter (C case):

```
template <auto&> struct A { };
template <auto> struct B { };
template <decltype(auto)> struct C { };

template <template<universal template> class Z, universal template U>
struct Foo : Z<U> { };

int x;
Foo<A, x> b1; // Ok: type of U is int&, instantiate A<int&>
Foo<B, x> b1; // Ok: type of U is int&, instantiate B<int>
Foo<C, x> b1; // Unexpected: type of U is int&, instantiate C<int&>,
// whereas C<x> would be ill-formed.
```

- Defer type deduction of *id-expressions*.

To make `decltype(auto)` consistent, instead of deducing the type of an *id-expression* used as an argument to a universal template parameter, we could defer the deduction to the point at which it is used (as a template argument for a value template parameter):

```
template <int&> struct A { };
template <universal template U> struct B : A<U> { };
int x;
B<x> b1;
```

If we try to deduce the type of `U` from `x`, then `U` is `int` and the program is ill-formed (because `int&` cannot bind to `int`) If instead, `U` is deduced to be a sort of symbolic reference to the variable `x`, and we then perform argument deduction for `A<U>`, which in effect would be identical to performing deduction for `A<x>`, then the program above would be well-formed.

Under this model:

```
template <decltype(auto)> struct A { };
template <auto&> struct B { };
template <template <universal template> class Z, universal template U>
using Alias = Z<U>;

int i;

Alias<A, i> a1;    // ill-formed (non-constant expression used as template argument)
Alias<A, (i)> a2;  // ok A<int&>
Alias<B, i> a3;    // ok B<int&>
```

To achieve this behavior, *id-expressions* bound to a universal template parameter behave as sort of symbolic references to the variable they denote, a bit like structured binding of aggregates. This allows "perfect forwarding" of variables through universal parameters. To match the behavior of `decltype`, this model applies to *id-expressions* and member access expressions.

This last option is the one we propose.

## Injected class names

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

template <typename T>
struct S {
    void g() {
        f<S>(); // #1
    }
};
```

In #1, `S` can either refer to the class template `S`, or the type `S<T>`.

We choose to always treat it as `S<T>`. We considered and rejected these alternatives:

- Always treat it as a template template-argument: Our assumption is that type template arguments are more often used, and that defaulting to a template template-argument would therefore be less useful and defy user expectations. Note however that if `S` was treated as an injected class name, then `S<T>` would spell the type so there is a natural disambiguation syntax. with this option. As such it might be a simpler, more elegant model. However, we are concerned that users would be surprised.
- Invent a new kind of "injected class name" template argument to defer the decision to the template argument is passed to a non-template parameter. While this would work in all cases it introduces additional complexity (both for implementation and teaching) that we did not feel was warranted.
- Make it ill-formed. This would let us delay a decision to a later date.

Ultimately, we feel like treating it as a type is the most useful. To get a template name, it is possible to use a qualified name

```
template <universal template>
void f();
template <typename T>
struct S {
    void g() {
        f<S>(); // S is S<T>
        f<::S>(); // S is S
    }
};
```

In the presence of a class nested in an anonymous class, this will not work. e.g

```
struct {
    template <typename T>
    struct K {
        template <typename>
        using K2 = K;
        void f</*???*/>(); // no way to qualify K
        void f<K2>(); // ok (K2 denotes a class template)
    };
} x;
```

We did consider a disambiguation syntax, however qualifying the name works in the overwhelming majority of cases, and there are workaround (using an alias template) in the fringe "class template nested in an anonymous class" case, so we rejected that approach as it would add additional complexity we consider unnecessary.

## ABI implications

We talked to Itanium ABI people (thanks Richard!), and the current proposal will require new mangling (for universal template parameters and symbolic variables, and maybe injected class

names depending on the direction we choose). These new mangling do not pose particular challenges.

## Concepts

A previous version of this paper claimed we lacked motivation for concepts to support universal template parameters. Playing a bit more with the feature, there seem to be valid use cases. Consider the following example:

```
template <universal template K,  
        class... Types>  
concept VariantKey = requires { std::get<K>(std::declval<std::variant<Types...>>()); };
```

Despite that, we are not proposing universal template parameters in concepts yet. Indeed, invalid substitutions in concept-ids during normalization are IFNDR, and that would make it too easy to write invalid code. Before we can allow UTPs in concept definitions, we should relax or refine that restriction.

```
template<class> concept T = true;  
template<universal template U> concept C = is_type_v<U> && T<U>;  
template<auto V> concept A = C<V>; // IFNDR
```

## Function template

We are not proposing to allow passing an overload set or a function template as a universal template argument. This might be worth future consideration but would come with implementation challenges. Overall this is an orthogonal feature that has less to do with universal template arguments than with the countless “overload set as first class objects” ([P1170R0](#) [7]) and “customization point objects” ([P2547R1](#) [4]) papers.

## Library Support: Universal template parameters with is

We plan to propose a set of library traits to accompany the core language feature.

```
template <universal template T>  
inline constexpr bool is_typename_v = false;  
template <universal template U>  
inline constexpr bool is_nttp_v = false;  
template <universal template U>  
inline constexpr bool is_template_v = false;  
template <universal template U>  
inline constexpr bool is_type_template_v = false;  
template <universal template U>  
inline constexpr bool is_var_template_v = false;  
template <universal template U>  
inline constexpr bool is_concept_v = false;
```

These type traits can be specialized as follows:

```

template <typename T>
inline constexpr bool is_typename_v<T> = true;

template <decltype(auto) V>
inline constexpr bool is_nttp_v<V> = true;

template <template<universal template...> typename U>
constexpr bool is_template_v<U> = true;

template <template<universal template...> auto U>
constexpr bool is_template_v<U> = true;

template <template<universal template...> concept U>
constexpr bool is_template_v<U> = true;

template <template<universal template...> typename U>
inline constexpr bool is_type_template_v<U> = true;

template <template<universal template...> auto U>
inline constexpr bool is_var_template_v<U> = true;

template <template<universal template...> concept U>
inline constexpr bool is_concept_v<U> = true;

```

[\[Demo on compiler explorer\]](#)

The final wording will most likely include support for the non-`_v` version of these traits.

We would be remiss if we did not propose `std::is_specialization_of` ([P2098R1 \[5\]](#)) here:

```

template<template<universal template...> typename, typename>
inline constexpr bool is_specialization_of_v = false;

template<
template<universal template...> typename Primary,
universal template... Args
>
inline constexpr bool is_specialization_of_v<Primary, Primary<Args...>> = true;

```

Unlike [P2098R1 \[5\]](#), which was rejected for not being universal enough (the technology did not exist at the time), this implementation not only supports checking specializations for any class templates, including those having template parameters that are not types, but also specialization of variable templates.

[\[Demo on compiler explorer\]](#)

## In which we mention reflection

One of the things we do not propose in this paper is to force the interpretation of a universal template as a specific kind "in place", in contrast with [P1985R3 \[3\]](#). That is, to use a universal template parameter, you have to pass it to another template, which is then specialized

for templates, NTPP, template names, and so on. Naming a universal template parameter anywhere except as a template argument is ill-formed.

There are a few reasons for this design choice. First, for types and NTPP, writing `as_type` and `as_value`, respectively, as library functions (not proposed), is easy enough.

```
template<universal template>
struct as_type;
template<typename T>
struct as_type<T> { using type = T; };
template <universal template T>
using as_type_t = as_type<T>::type;

template<universal template U>
constexpr auto as_value_v = delete;
template<decltype(auto) V>
constexpr decltype(auto) as_value_v<V> = V;
```

These would allow extracting a type/value from a UTPP:

```
template <universal template U>
constexpr auto test() {
    if constexpr(is_typename_v<U>) {
        using a = as_type<int>::type; // ok
        return a{42};
    }
    else if constexpr(is_ntpp_v<U>) {
        decltype(auto) v = as_value_v<U>;
        return v;
    }
}
static_assert(test<int>() == 42);
static_assert(test<24>() == 24);
```

[\[Demo on Compiler Explorer\]](#)

The second reason is that use cases are limited. We need universal template parameters so we can handle entities of different shapes in generic contexts, by forwarding them to other templates, as illustrated in the examples at the start of this paper.

However, the main reason is that the concern is somewhat orthogonal and not specific to this proposal. Enter reflection, which we already mentioned in [P1985R3](#) [3], and not just because we love to talk about reflection in every paper.

Reflection has a feature called splicing — although terminology changed over the years — by which you can convert a reflection of an entity back into that entity. Because it is possible to reflect on anything (in a theoretical future), then splicing a reflection can produce any kind of entity (we apologize to Core for our liberal use of “entity” throughout this paper).

Quoting from [P1240R2](#) [8]:

```

struct S { struct I { }; };
template<int N> struct X;
auto refl = ^S;
auto tmp1 = ^X;
void f() {
    typename[:refl:] * x; // Okay: declares x to be a pointer to S.
    [:refl:] * x; // Error: attempt to multiply int by x.
    [:refl:]::I i; // Okay: splice as part of a nested-name-specifier.
    typename[:refl:]{}; // Okay: default-constructs an S temporary.
    using T = [:refl:]; // Okay: operand must be a type.
    struct C: [:refl:] {}; // Okay: base classes are types.
    template[:tmp1:]<0>; // Okay: names the specialization.
    [:tmp1:] < 0 > x; // Error: attempt to compare X with 0.
}

```

This set of examples is rather illustrative of what we need to solve in general. Both splices and UTP are dependent expressions and need some form of parsing disambiguator to be usable in arbitrary contexts, like other dependent names (i.e., member of class templates).

So both [P1985R3](#) [3] and reflection had similar needs for disambiguating new interesting names, and both papers try to come up with rules for cleverly avoiding the need for a disambiguator syntax. As shown in [P1985R3](#) [3], if we allowed some form of aliases of universal template parameters and/or some form of general aliasing that would allow more entities to become dependent the need for disambiguation syntaxes would increase. Of course, work is done on member packs and pack aliases, which adds a layer of consideration to these disambiguation syntaxes.

Adding disambiguators and implicit disambiguator rules would have a nontrivial impact on C++ parsers, so progressing one step at a time seems reasonable. Thus we focus solely on allowing universal template parameters. Once we have the basis right, we can expand to allow UTP in more places, if we find a compelling use case for it. Both reflection and the use of UTP outside of template arguments should have a consistent syntax for disambiguators and consistent rules for where we can and cannot omit them.

## Syntax

The set of possible syntaxes is for universal template parameters infinite trying to do an exhaustive search is unlikely productive,

- universal template Foo works (i.e., is not ambiguous). [A search for #define universal](#) finds one instance in a no-longer maintained project.
- template auto which is the syntax used by circle and [P1985R0](#) [2] reused auto in inconsistent ways which we and many others found undesirable.
- template as an isolated keyword is perfectly fine, but some committee members have expressed opposition to that.

- `__any` or `__universal` would also work, but C++ doesn't traditionally embrace underscore-prefixed keywords (unlike C).
- `universal_template` or similar would also work but again we don't often use underscore in keywords (though, `co_yield` and `co_do`).
- `anytmplarg` or any such weird enough keyword to be unlikely to be used as an identifier would also work. (we can't pick something non-weird, to not break existing code).

We are proposing `universal template`. EWG seems to not hate it going from previous discussions. We are, of course, *not* proposing to make `universal` a keyword. In a template parameter declaration, `universal template` would have a special meaning, so the meaning of `universal` would be contextual, like `module`, `final` and `override`.

```
template <
    universal template, // template
    universal foo      // type constraint or NTTP
>
```

The proposed feature allows us to express ideas that can't be expressed otherwise, so it is important, but we expect `universal template` parameters will mostly be used by select generic facilities. Terseness is not a goal here.

The `template auto` syntax is perhaps less cromulent than the two other options illustrated above. In addition to overloading `auto` with a novel meaning (one that has little to do with variables), this syntax makes it harder (for a human) to distinguish variable templates, template variable template parameters, template template parameters with a variable template, universal templates, template template parameters with a universal template, etc.

```
template <
    auto, // variable
    template auto, // universal parameter
    template <auto> auto, // variable template
    template <template auto> auto // variable template with a universal template parameter.
>
struct S;
```

We should clarify that none of the options proposed here pose any challenge for implementation. The only concern is what will feel more intuitive to developers with time, and any reasonable syntax we pick will become familiar with time.

## Status of this proposal and implementation

The design presented in this paper roughly matches the implementation of universal template parameters in Circle (itself inspired by a previous iteration of this proposal), albeit with a different syntax.

We provide a prototype implementation in Clang to demonstrate implementability in a second C++ compiler and discover any interesting design questions we might otherwise miss. Our implementation compiles the examples presented in this paper. However, it remains a

prototype that is not production-ready. The implementation represented a few weeks of work. The main challenge of bringing this proposal to production-ready implementations certainly resides in the testing of all possible combinations of template parameters and arguments.

We aim to agree on general design and syntax in Kona and then follow up in the next few months with wording. We should also progress [P2841R0](#) [6], first since we intend for concepts and variable templates to be valid universal template parameters. There is a better order of operation, especially for ease of specification.

## Future work

The library part of this feature will be explored in a separate paper. A few other bits (pack indexing for universal template parameters, whether we should allow reflecting on universal template parameters, and universal template parameters as concept definition parameters) are also left for future papers.

## Wording



### Template parameters

[temp.param]

The syntax for *template-parameters* is:

```

template-parameter:
    type-parameter
    parameter-declaration
    type-tt-parameter
    variable-tt-parameter
    concept-tt-parameter
    universal-template-parameter

type-parameter:
    type-parameter-key ...opt identifieropt
    type-parameter-key identifieropt = type-id
    type-constraint ...opt identifieropt
    type-constraint identifieropt = type-id

type-parameter-key:
    class
    typename

type-constraint:
    nested-name-specifieropt concept-name
    nested-name-specifieropt concept-name < template-argument-listopt >

type-tt-parameter:
    template-head type-parameter-key ...opt identifieropt
    template-head type-parameter-key identifieropt type-tt-parameter-default
```

*type-tt-parameter-default*:  
 = *nested-name-specifier*<sub>opt</sub> *template-name*  
 = *nested-name-specifier* *template* *template-name*

*variable-tt-parameter*:  
*template-head* *auto* ...<sub>opt</sub> *identifier*<sub>opt</sub>  
*template-head* *auto* *identifier*<sub>opt</sub> = *nested-name-specifier*<sub>opt</sub> *template-name*

*concept-tt-parameter*:  
*template* < *template-parameter-list* > *concept* ...<sub>opt</sub> *identifier*<sub>opt</sub>  
*template* < *template-parameter-list* > *concept* *identifier*<sub>opt</sub> = *nested-name-specifier*<sub>opt</sub> *template-name*

*universal-template-parameter*:  
*universal* *template* ...<sub>opt</sub> *identifier*<sub>opt</sub>

The component names of a *type-constraint* are its *concept-name* and those of its *nested-name-specifier* (if any). [ *Note*: The > token following the *template-parameter-list* of a *type-tt-parameter*, *variable-tt-parameter*, or *concept-tt-parameter* can be the product of replacing a >> token by two consecutive > tokens [temp.names]. — *end note* ]

A template parameter is of one of the following kinds:

- A *type template parameter* is a template parameter introduced by a *type-parameter*.
- A *constant template parameter* is a template parameter introduced by a *parameter-declaration*.
- A *type template template parameter* is a template parameter introduced by a *type-tt-parameter*.
- A *variable template template parameter* is a template parameter introduced by a *variable-tt-parameter*.
- A *concept template parameter* is a template parameter introduced by a *concept-tt-parameter*.
- A *universal template parameter* is a template parameter introduced by a *universal-template-parameter*.

Type template template parameters, variable template template parameters, and concept template parameters are collectively referred to as *template template parameters*.

The *nested-name-specifier* of a *type-constraint*, if any, shall not be dependent.

A concept template parameter shall not have associated constraints [temp.constr.decl].

If a *template-parameter* is a *parameter-declaration* that declares a pack [dcl.fct], or otherwise has an ellipsis prior to its optional *identifier*, then the *template-parameter* declares a template parameter pack [temp.variadic]. A template parameter pack that is a *parameter-declaration* whose type contains one or more unexpanded packs is a pack expansion. Similarly, a template parameter pack that is a template template parameter with a *template-parameter-list* containing one or more unexpanded packs is a pack expansion. A type parameter pack with a *type-constraint* that contains an unexpanded parameter pack is a pack expansion. A template

parameter pack that is a pack expansion shall not expand a template parameter pack declared in the same *template-parameter-list*. [Example:

```

    template <class... Types>                                // Types is a template type parameter
pack
    class Tuple;                                              // but not a pack expansion

    template <class T, int... Dims>                            // Dims is a constant template
parameter pack
    struct multi_array;                                       // but not a pack expansion

    template <class... T>
    struct value_holder {
        template <T... Values> struct apply { };             // Values is a constant template
parameter pack
    };                                                         // and a pack expansion

    template <class... T, T... Values>                          // error: Values expands template
type parameter
    struct static_array;                                       // pack T within the same template
parameter list

```

— end example]

There is no semantic difference between `class` and `typename` in a *type-parameter-key*. `typename` followed by an *unqualified-id* names a template type parameter. `typename` followed by a *qualified-id* denotes the type in a *parameter-declaration*. A *template-parameter* of the form `class identifier` is a *type-parameter*. [Example:

```

class T { /*...*/ };
int i;

template<class T, T i> void f(T t) {
    T t1 = i;          // template parameters T and i
    ::T t2 = ::i;      // global namespace members T and i
}

```

Here, the template `f` has a type template parameter called `T`, rather than an unnamed constant template parameter of class `T`. — end example] The *parameter-declaration* of a *template-parameter* shall not have a *storage-class-specifier*. Types shall not be defined in a template parameter declaration.

The *identifier* in a *template-parameter* denoting a type, [universal template parameter](#), or template is not looked up. An *identifier* that does not follow an ellipsis is defined to be

- a *typedef-name* for a *type-parameter*,
- a *universal-argument-name* for a *universal-template-parameter*,
- a *template-name* for a *variable-tt-parameter*,

- a *template-name* for a *type-tt-parameter*, or
- a *concept-name* for a *concept-tt-parameter*,

in the scope of the template declaration.



## Names of template specializations

[temp.names]

A template specialization [temp.spec] can be referred to by a *template-id*:

*simple-template-id*:

*template-name* < *template-argument-list*<sub>opt</sub> >

*template-id*:

*simple-template-id*

*operator-function-id* < *template-argument-list*<sub>opt</sub> >

*literal-operator-id* < *template-argument-list*<sub>opt</sub> >

*template-name*:

*identifier*

*template-argument-list*:

*template-argument* . . . <sub>opt</sub>

*template-argument-list* , *template-argument* . . . <sub>opt</sub>

*template-argument*:

*template-argument-name*

*universal-argument-name*

*constant-expression*

*type-id*

*braced-init-list*

*template-argument-name*:

*nested-name-specifier*<sub>opt</sub> *identifier*

*nested-name-specifier* *template* *identifier*

*universal-argument-name*:

*identifier*

The component name of a *simple-template-id*, *template-id*, or *template-name* is the first name in it.

[Editor's note: [...]]

The component names of a *template-argument-name* are those of its *nested-name-specifier* (if any) and its *identifier*.

The component name of a *universal-argument-name* is its *identifier*.

[Editor's note: [...]]

## ❖ Template arguments

[temp.arg]

### ❖ General

[temp.arg.general]

The type and form of each *template-argument* specified in a *template-id* or in a *splice-specialization-specifier* shall match the type and form specified for the corresponding parameter declared by the template in its *template-parameter-list*.

[Editor's note: The pre-existing wording seems somewhat vague.]

When the parameter declared by the template is a template parameter pack [temp.variadic], it will correspond to zero or more *template-arguments*. [Example:

```
template<class T> class Array {
    T* v;
    int sz;
public:
    explicit Array(int);
    T& operator[](int);
    T& elem(int i) { return v[i]; }
};

Array<int> v1(20);
typedef std::complex<double> dcomplex; // std::complex is a standard library
template
Array<dcomplex> v2(30);
Array<dcomplex> v3(40);

void bar() {
    v1[3] = 7;
    v2[3] = v3.elem(4) = dcomplex(7,8);
}
```

— end example ]

The template argument list of a *template-head* is a template argument list in which the  $n^{\text{th}}$  template argument has the value of the  $n^{\text{th}}$  template parameter of the *template-head*. If the  $n^{\text{th}}$  template parameter is a template parameter pack [temp.variadic], the  $n^{\text{th}}$  template argument is a pack expansion whose pattern is the name of the template parameter pack.

If a *template-argument* *A* matches the form *template-argument-name*, it is interpreted as such; the *identifier* is looked up and its meaning is determined as follows:

- If lookup finds an injected-class-name [temp.local], then:
  - When *A* is for a type template template parameter, *A* denotes the corresponding class template.
  - Otherwise (if *A* is for a type template parameter or a universal template parameter), it denotes a *type-name*.

- Otherwise, if lookup finds a template, *A* denotes that template.
- Otherwise, if lookup finds a universal template parameter, *A* denotes that universal template parameter.
- Otherwise, if lookup finds a type alias or a type, *A* denotes the underlying type and is interpreted as a *type-id*.
- Otherwise, *A* is interpreted as an expression.

In a *template-argument*, an ambiguity between a *type-id* and an expression is resolved to a *type-id*, regardless of the form of the corresponding *template-parameter*.<sup>1</sup> here is no such ambiguity in a default *template-argument* because the form of the *template-parameter* determines the allowable forms of the *template-argument*. [Example:

```
template<class T> void f();
template<int I> void f();

void g() {
    f<int>();           // int() is a type-id: call the first f()
}
```

— end example]

[Note: Names used in a *template-argument* are subject to access control where they appear. Because a template parameter is not a class member, no access control applies where the template parameter is used. — end note] [Example:

```
template<class T> class X {
    static T t;
};

class Y {
private:
    struct S { /*...*/ };
    X<S> x;           // OK, S is accessible
    // X<Y::S> has a static member of type Y::S
    // OK, even though Y::S is private
};

X<Y::S> y;           // error: S not accessible
```

— end example] For a template argument that is a class type or a class template, the template definition has no special access rights to the members of the template argument. [Example:

```
template <template <class TT> class T> class A {
    typename T<int>::S s;
};
```

---

<sup>1</sup>T

```

template <class U> class B {
    private:
        struct S { /*...*/ };
};

A<B> b;           // error: A has no access to B::S

```

— end example]

When template argument packs or default template arguments are used, a *template-argument* list can be empty. In that case the empty <> brackets shall still be used as the *template-argument-list*. [Example:

```

template<class T = char> class String;
String<>* p;           // OK, String<char>
String* q;            // syntax error
template<class ... Elements> class Tuple;
Tuple<>* t;            // OK, Elements is empty
Tuple* u;             // syntax error

```

— end example]

An explicit destructor call [class.dtor] for an object that has a type that is a class template specialization may explicitly specify the *template-arguments*. [Example:

```

template<class T> struct A {
    ~A();
};
void f(A<int>* p, A<int>* q) {
    p->A<int>::~~A();           // OK, destructor call
    q->A<int>::~~A<int>();      // OK, destructor call
}

```

— end example]

If the use of a template argument gives rise to an ill-formed construct in the instantiation of a template specialization, the program is ill-formed.

When name lookup for the component name of a *template-id* finds an overload set, both non-template functions in the overload set and function templates in the overload set for which the *template-arguments* do not match the *template-parameters* are ignored. [Note: If none of the function templates have matching *template-parameters*, the program is ill-formed. — end note]

When a *simple-template-id* or *splice-specialization-specifier* does not designate a function, a default *template-argument* is implicitly instantiated [temp.inst] when the value of that default argument is needed. [Example:

```
template<typename T, typename U = int> struct S { };
S<bool>* p;           // the type of p is S<bool, int>*
```

The default argument for *U* is instantiated to form the type `S<bool, int>*`. — *end example*]

A *template-argument* followed by an ellipsis is a pack expansion [temp.variadic].

## ◆ Type template arguments

[temp.arg.type]

A *template-argument* for a type template parameter shall be a *type-id* or a *universal-argument-name*.

[Example:

```
template <class T> class X { };
template <class T> void f(T t) { }
struct { } unnamed_obj;

void f() {
    struct A { };
    enum { e1 };
    typedef struct { } B;
    B b;
    X<A> x1;           // OK
    X<A*> x2;          // OK
    X<B> x3;           // OK
    f(e1);             // OK
    f(unnamed_obj);    // OK
    f(b);             // OK
}

template <universal template U>
void g() {
    X<U> x4; // OK
}
```

— *end example*] [Note: A template type argument can be an incomplete type [term.incomplete.type]. — *end note*]

## ◆ Constant template arguments

[temp.arg.nontype]

A template argument *E* for a constant template parameter with declared type *T* shall be such that the invented declaration

```
T x = E ;
```

satisfies the semantic constraints for the definition of a `constexpr` variable with static storage duration [dcl.constexpr].

## ◆ Template template arguments

[temp.arg.template]

A *template-argument* for a template template parameter shall be the name of a template, [or a universal-argument-name](#). For a *type-tt-parameter*, the name shall denote a class template, alias template, or type template template parameter. For a *variable-tt-parameter*, the name shall denote a variable template or variable template template parameter. For a *concept-tt-parameter*, the name shall denote a concept or concept template parameter. Only primary templates are considered when matching the template template argument with the corresponding parameter; partial specializations are not considered even if their parameter lists match that of the template template parameter.

[Editor's note: [...]]

A template *template-argument* A matches a template template parameter P when A and P are compatible and P is at least as specialized as A, ignoring constraints on A if P is unconstrained. If P contains a template parameter pack, then A also matches P if each of A's template parameters matches the corresponding template parameter declared in the *template-head* of P. Two template parameters match if they are of the same kind, for constant template parameters, their types are equivalent [temp.over.link], and for template template parameters, each of their corresponding template parameters matches, recursively. When P's *template-head* contains a *template-parameter* that declares a template parameter pack [temp.variadic], the template parameter pack will match zero or more template parameters or template parameter packs declared in the *template-head* of A with the same type and form as the template parameter pack declared in P (ignoring whether those template parameters are template parameter packs). [A universal template parameter declared in the \*template-head\* of P matches a template parameter of any kind; if it is a template parameter pack, it matches zero or more template parameters or template parameter packs of any kind.](#)

[Example:

```
template<class T> class A { /* ... */ };
template<class T, class U = T> class B { /* ... */ };
template<class ... Types> class C { /* ... */ };
template<auto n> class D { /* ... */ };
template<template<class> class P> class X { /* ... */ };
template<template<class ...> class Q> class Y { /* ... */ };
template<template<int> class R> class Z { /* ... */ };

template<template<universal template ...> class S> class U { /* ... */ };
template<template<class, universal template> class T> class V { /* ... */ };

X<A> xa;           // OK
X<B> xb;           // OK
X<C> xc;           // OK
Y<A> ya;           // OK
Y<B> yb;           // OK
Y<C> yc;           // OK
Z<D> zd;           // OK

U<A> ua;           // OK
```

```
U<D> ud;          // OK
V<B> vb;          // OK
```

— *end example*]

A template template parameter *P* is at least as specialized as a template *template-argument* *A* if, given the following rewrite to two function templates, the function template corresponding to *P* is at least as specialized as the function template corresponding to *A* according to the partial ordering rules for function templates [temp.func.order]. Given an invented class template *X* with the *template-head* of *A* (including default arguments and *requires-clause*, if any):

- Each of the two function templates has the same template parameters and *requires-clause* (if any), respectively, as *P* or *A*.
- Each function template has a single function parameter whose type is a specialization of *X* with template arguments corresponding to the template parameters from the respective function template where, for each *template-parameter* *PP* in the *template-head* of the function template, a corresponding *template-argument* *AA* is formed: as follows:
  - If *PP* declares a template parameter pack, **then** *AA* is the pack expansion *PP*. . . [temp.variadic];
  - Otherwise, if *PP* declares a universal template parameter, *AA* is a *universal-argument-name* naming *PP*.
  - **otherwise** Otherwise, *AA* is an *id-expression* denoting *PP*.

If the rewrite produces an invalid type, then *P* is not at least as specialized as *A*.

## ❖ Universal template arguments [temp.arg.universal]

A *template-argument* *A* for a universal template parameter *P* is interpreted as follows:

- If *A* is a *universal-argument-name*, the template argument for *P* is the universal template parameter named by *A*.
- Otherwise, if *A* is a *type-id*, *A* is a type template argument and the template argument for *P* is the type denoted by *A*.
- Otherwise, if *A* denotes a template [temp.arg.general] other than a function template, *A* is a template template argument and the template argument for *P* is that template.
- Otherwise, if *A* is an unparenthesized *id-expression* that does not name a constant template parameter or an unparenthesized class member access [expr.ref], *A* is a *deferred template argument*, and the template argument for *P* is *A*.
- Otherwise, if *A* is a constant template argument, *P* is considered to be a constant template parameter declared with type `decltype(auto)` [temp.arg.nontype], and the template argument for *P* is the value of *A*.

When template arguments are substituted for template parameters, a *universal-argument-name* naming a universal template parameter, or an *id-expression* naming a constant template

parameter, designates the template argument for that parameter. A *universal-argument-name* that designates a template argument  $X$  is interpreted as

- a *universal-argument-name* naming  $X$ , if  $X$  is a universal template parameter,
- a *type-id* denoting  $X$ , if  $X$  is a type,
- a *template-argument-name* denoting  $X$ , if  $X$  is a template,
- $X$ , if  $X$  is a deferred template argument, and
- an *id-expression* naming an invented constant template parameter of the type of  $X$  and designating  $X$ , if  $X$  is a value.

## ◆ Type equivalence

[temp.type]

Two *template-ids* are the same if

- their *template-names*, *operator-function-ids*, or *literal-operator-ids* refer to the same template, and
- their corresponding type *template-arguments* are the same type, and
- the template parameter values determined by their corresponding constant template arguments [temp.arg.nontype] are template-argument-equivalent (see below), **and**
- their corresponding template *template-arguments* refer to the same template, **and**
- their corresponding deferred template arguments [temp.arg.universal] are equivalent [temp.over.link], and
- their corresponding universal *template-arguments* refer to the same template parameter.

## ◆ Variadic templates

[temp.variadic]

A *pack expansion* consists of a *pattern* and an ellipsis, the instantiation of which produces zero or more instantiations of the pattern in a list (described below). The form of the pattern depends on the context in which the expansion occurs. Pack expansions can occur in the following contexts:

- In a function parameter pack [dcl.fct]; the pattern is the *parameter-declaration* without the ellipsis.
- In a *using-declaration* [namespace.udecl]; the pattern is a *using-declarator*.
- In a *friend-type-declaration* [class.mem.general]; the pattern is a *friend-type-specifier*.
- In a template parameter pack that is a pack expansion [temp.param]:
  - if the template parameter pack is a *parameter-declaration*; the pattern is the *parameter-declaration* without the ellipsis;

- if the template parameter pack is a *type-parameter*; the pattern is the corresponding *type-parameter* without the ellipsis;
- if the template parameter pack is a template template parameter; the pattern is the corresponding *type-tt-parameter*, *variable-tt-parameter*, or *concept-tt-parameter* without the ellipsis.
- if the template parameter pack is a *universal-template-parameter*; the pattern is the corresponding *universal-template-parameter* without the ellipsis;

[Editor's note: [...]]

The instantiation of a pack expansion considers items  $E_1, E_2, \dots, E_N$ , where  $N$  is the number of elements in the pack expansion parameters. Each  $E_i$  is generated by instantiating the pattern and replacing each pack expansion parameter with its  $i^{\text{th}}$  element. Such an element, in the context of the instantiation, is interpreted as follows:

- if the pack is a template parameter pack, the element is
  - a *typedef-name* for a type template parameter pack,
  - an *id-expression* for a constant template parameter pack, **or**
  - a *template-name* for a template template parameter pack, **or**
  - a *universal-argument-name* for a universal template parameter pack

designating the  $i^{\text{th}}$  corresponding type, constant, **or** template, or universal template argument;

- if the pack is a function parameter pack, the element is an *id-expression* designating the  $i^{\text{th}}$  function parameter that resulted from instantiation of the function parameter pack declaration;
- if the pack is an *init-capture* pack, the element is an *id-expression* designating the variable introduced by the  $i^{\text{th}}$  *init-capture* that resulted from instantiation of the *init-capture* pack declaration; otherwise
- if the pack is a structured binding pack, the element is an *id-expression* designating the  $i^{\text{th}}$  structured binding in the pack that resulted from the structured binding declaration.

When  $N$  is zero, the instantiation of a pack expansion does not alter the syntactic interpretation of the enclosing construct, even in cases where omitting the pack expansion entirely would otherwise be ill-formed or would result in an ambiguity in the grammar.

## ◆ Partial ordering of function templates

[temp.func.order]

To produce the transformed template, for each type, constant, type template, variable template, **or** concept, or universal template parameter (including template parameter packs [temp.variadic] thereof) synthesize a unique type, value, class template, variable template, **or** concept, or universal template argument, respectively, and substitute it for each occurrence of that parameter in the function type of the template. The type replacing a placeholder in

the type of the value synthesized for a constant template parameter is a unique synthesized type. [\[Note: A synthesized universal template argument is a valid template argument for a template parameter of any kind. — end note\]](#)

## ◆ **Concept definitions** [temp.concept]

A concept shall not have associated constraints [temp.constr.decl].

[A concept shall not have a universal template parameter.](#)

## ◆ **Dependent template arguments** [temp.dep.temp]

A type *template-argument* is dependent if the type it specifies is dependent.

A constant *template-argument* is dependent if its type is dependent or the constant expression it specifies is value-dependent.

Furthermore, a constant *template-argument* is dependent if the corresponding constant template parameter is of reference or pointer type and the *template-argument* designates or points to a member of the current instantiation or a member of a dependent type.

A template argument is also dependent if it is a pack expansion.

A template template argument is dependent if it names a template parameter or its terminal name is dependent.

[A deferred template argument \[temp.arg.universal\] is dependent if it is type-dependent or value-dependent, or if it names a member of the current instantiation or a member of a dependent type.](#)

[A universal template argument is dependent.](#)

## ◆ **Function template specializations** [temp.fct.spec]

### ◆ **Template argument deduction** [temp.deduct]

### ◆ **Deducing template arguments during partial ordering** [temp.deduct.partial]

If, for a given type, the types are identical after the transformations above and both P and A were reference types (before being replaced with the type referred to above):

- if the type from the argument template was an lvalue reference and the type from the parameter template was not, the parameter type is not considered to be at least as specialized as the argument type; otherwise,
- if the type from the argument template is more cv-qualified than the type from the parameter template (as described above), the parameter type is not considered to be at least as specialized as the argument type.

If, for a given pair of corresponding template parameters of the two templates, one is a universal template parameter, that template parameter is not considered to be at least as specialized as the other.

Function template *F* is *at least as specialized as* function template *G* if, for each pair of types used to determine the ordering, the type from *F* is at least as specialized as the type from *G*, and, for each pair of corresponding template parameters that are not used in those types, the template parameter from *F* is at least as specialized as the template parameter from *G*. *F* is *more specialized than* *G* if *F* is at least as specialized as *G* and *G* is not at least as specialized as *F*. If, after considering the above, function template *F* is at least as specialized as function template *G* and vice-versa, and if *G* has a trailing function parameter pack for which *F* does not have a corresponding parameter, and if *F* does not have a trailing function parameter pack, then *F* is more specialized than *G*.

## ◆ **Deducing template arguments from a type** [temp.deduct.type]

Template arguments can be deduced in several different contexts, but in each case a type that is specified in terms of template parameters (call it *P*) is compared with an actual type (call it *A*), and an attempt is made to find template argument values (a type for a type parameter, a value for a constant template parameter, **or** a template for a template template parameter, or any template argument [temp.arg.universal] for a universal template parameter) that will make *P*, after substitution of the deduced values (call it the deduced *A*), compatible with *A*. [Editor's note: [...]] The non-deduced contexts are:

- [Editor's note: [...]]
- A constant template argument, a deferred template argument, or an array bound in which a subexpression references a template parameter.
- [Editor's note: [...]]

[Editor's note: [...]] A type template argument *T*, a constant template argument *i*, a template template argument *TT* denoting a class template or an alias template, **or** a template template argument *VV* denoting a variable template or a concept, or a universal template argument *U* can be deduced if *P* and *A* have one of the following forms:

```

cvopt T
T*
T&
T&&
Topt [iopt ]
Topt (Topt ) noexcept(iopt )
Topt Topt ::*
TTopt <T>
TTopt <i>
TTopt <TT>
TTopt <VV>
TTopt <U>
TTopt <>

```

where

- $T_{opt}$  represents a type or parameter-type-list that either satisfies these rules recursively, is a non-deduced context in P or A, or is the same non-dependent type in P and A,
- $i_{opt}$  represents an expression that either is an  $i$ , is value-dependent in P or A, or has the same constant value in P and A,
- $TT_{opt}$  represents either a class template or a template template parameter, and
- `noexcept( $i_{opt}$ )` represents an exception specification [except.spec] in which the (possibly-implicit, see [dcl.fct]) *noexcept-specifier's* operand satisfies the rules for an  $i_{opt}$  above.

[Note: If a type matches such a form but contains no Ts, is, or TTs, deduction is not possible. — end note] Similarly,  $\langle X \rangle$  represents template argument lists where at least one argument contains an  $X$ , where  $X$  is one of T,  $i$ , TT, ~~or~~ VV, ~~or~~ U; and  $\langle \rangle$  represents template argument lists where no argument contains a T, an  $i$ , a TT, ~~or~~ a VV, ~~or~~ a U. If P has a form that contains  $\langle T \rangle$ ,  $\langle i \rangle$ ,  $\langle TT \rangle$ , ~~or~~  $\langle VV \rangle$ , ~~or~~  $\langle U \rangle$ , then each argument  $P_i$  of the respective template argument list of P is compared with the corresponding argument  $A_i$  of the corresponding template argument list of A.

[Editor's note: [...]]

If P has a form that contains  $\langle i \rangle$ , deduction fails unless the type of  $i$  is the same as that of the corresponding template parameter  $p$  in the specialization (from A) of the template named by the enclosing *simple-template-id* or *splice-specialization-specifier*; if the declared type of  $i$  contains a placeholder type, the corresponding template argument for the purposes of placeholder type deduction [dcl.type.auto.deduct] is an *id-expression* for  $p$ .

[Editor's note: [...]]

When matching a partial specialization [temp.spec.partial.match], if a template argument  $E$  of the specialization being matched is a deferred template argument [temp.arg.universal] and the corresponding template argument of the partial specialization is  $i$ , then, instead of the rules above for  $\langle i \rangle$ , the template argument deduced for  $i$  is  $E$ ; deduction fails unless  $E$  would be a valid template argument for  $i$  [temp.arg.nontype], and the deduced A is considered compatible with A. [Example:

```
template<universal template> struct K { static constexpr int v = 0; };
template<auto V> struct K<V> { static constexpr int v = 1; };
template<int& R> struct K<R> { static constexpr int v = 2; };
template<long L> void f(K<L>);
constexpr int c = 24;
int x;
static_assert(K<c>::v == 1);    // V is 24
static_assert(K<x>::v == 2);    // R refers to x
static_assert(K<42>::v == 1);
void g() {
    f(K<c>());                  // error: deduction fails
}
```

— end example]

## Feature-test macro

[*Editor's note:* In [tab:cpp.predefined.ft], add `__cpp_universal_template` defined to the date of adoption.]

## Acknowledgments

We would like to thank Bengt Gustafsson, Brian Bi, Joshua Berne, and Pablo Halpern for their valuable feedback on this paper.

Thanks to Barry Revzin for suggesting an alternative to deduce the type of value template parameters.

Thanks to Richard Smith and Brian Bi for the discussions around injected class names.

## References

- [1] Ruslan Arutyunyan and Alexey Kukanov. P2769R1: `get_element` customization point object. <https://wg21.link/p2769r1>, 5 2023.
- [2] Gašper Ažman and Mateusz Pusz. P1985R0: Universal template parameters. <https://wg21.link/p1985r0>, 1 2020.
- [3] Gašper Ažman, Mateusz Pusz, Colin MacLean, Bengt Gustafsson, and Corentin Jabot. P1985R3: Universal template parameters. <https://wg21.link/p1985r3>, 9 2022.
- [4] Lewis Baker, Corentin Jabot, and Gašper Ažman. P2547R1: Language support for customisable functions. <https://wg21.link/p2547r1>, 7 2022.
- [5] Walter E Brown and Bob Steagall. P2098R1: Proposing `std::is_specialization_of`. <https://wg21.link/p2098r1>, 4 2020.
- [6] Corentin Jabot and Gašper Ažman. P2841R0: Concept template parameters. <https://wg21.link/p2841r0>, 5 2023.
- [7] Barry Revzin and Andrew Sutton. P1170R0: Overload sets as function parameters. <https://wg21.link/p1170r0>, 10 2018.
- [8] Daveed Vandevoorde, Wyatt Childers, Andrew Sutton, and Faisal Vali. P1240R2: Scalable reflection. <https://wg21.link/p1240r2>, 1 2022.