Concept and variable-template template-parameters

Document #: P2841R0
Date: 2023-05-18
Programming Language C++
Audience: EWG
Reply-to: Corentin Jabot <corentin.jabot@gmail.com>
Gašper Ažman <gasper.azman@gmail.com>

Abstract

C++ allows passing templates as template parameters. It does not, however, support these template template-parameters to be non-types, i.e., either variables or concepts, which represents a hole in the template facilities and is the topic of this paper.

Motivation

Template template-parameters allow for higher-order templates and greater composability. They can be used, for example, to parametrize a function that operates on any container of any type or to write CRTP-based interfaces.

C++23 limits template template-parameters to be class templates or alias templates. A variable template (C++14) or a concept (concepts are themselves templates) cannot be passed as a template argument in C++23.

The motivation for passing a concept as a template argument is very much the same as our reason for supporting class templates as template arguments: to allow higher-level constructs.

Being able to define a concept adaptor, for instance, would be very nice:

```
template <typename T, template <typename> concept C>
concept decays_to = C<decay_t<T>>;
```

Being able to use it with any concept constraint would also be helpful:

```
template <decays_to<copyable> T>
auto f(T&& x);
```

Other such constructs might, for example, include the following.

- `range_of`
  Many algorithms can operate on a sequence of integer or string-like types and while it is possible to express `range<T> && SomeConcept<ranges::range_reference_t<R>>`, some codebases do that enough that they might want to have a shorter way to express that idea, one that would let them use the abbreviated syntax in more cases.

- `tuple_of`
  This follows the same idea, but expressing this idea in the require clause of each function or class that might need it would be an exercise in frustration and a maintenance nightmare. We explore a `tuple_of` concept later in this paper. Representing vectors as tuples-like things of numbers is common in the scientific community, and these scientific libraries have no ideal way to express these constraints.

- Avoiding duplication.
  In his blog post on this very topic, Barry Revzin observed that `std::ranges` defines a handful of concepts that are very similar to one another except they use different concepts internally. Concept template-parameters can reduce a lot of duplication. Compare the definitions in the Standard and the implementation with our proposal.

To quote Barry's aforementioned blog post

> I'd rather write a one-line definition per metaconcept, not a one-line definition per metaconcept instantiation.

So part of the motivation for concept template-parameters is the same motivation we have for functions, templates, and classes: We want to be able to reuse code and to make it less repetitive and error-prone.

We also demonstrated how this feature can be leveraged to provide better diagnostics when a concept is not satisfied [Compiler Explorer].

There is community interest in these features.

- Is it possible to pass a concept as a template parameter?
- Concept to assert an argument is another concept, with whatever parameters
- Passing a concept to a function
- How to pass a variable template as template argument
- Can a variable template be passed as a template template argument?

Unfortunately, this is one of those features that truly shows its power on large examples that don't tend to fit into papers.

Variable-template template-parameters

Variable-template template-parameters (previously proposed in P2008R0 [5]) are arguably less useful. They can be emulated with a template class with a static public `value` data member. Most standard type traits are defined as a type and have an equivalent `_v` variable:

```
template <typename T, typename U>
constexpr bool is_same_v = is_same<T, U>::value;
```

But this is not compile-time efficient: A class has to be instantiated, which is usually slow on most implementations. Then we still need to instantiate the variable. This performance issue is explored in more detail in P1715R1 [2].

Wrapping variables in class templates also adds complexity for users: The main reason we expose both a variable template and a class template for every boolean trait is because the language does not support variable-template template-parameters. (Note that we are aware of some codebases using traits as tags for dispatch but this far from the common case.)

For instance, counting elements that satisfy a specific predicate could be done as

```
template <template <typename> auto p, typename... Ts>
std::size_t count_if_v = (... + p<Ts>);
```

We could do the same thing with a type, but it incurs a class template instantiation for each element:

```
template <template <typename> typename p, typename... Ts>
std::size_t count_if_v = (... + p<Ts>::value);
```

It will always be more work for the compiler to instantiate a whole class together with its body (not just its declaration) to allow access to the inner value member than just instantiating a variable template, no matter how much we try to optimize this pattern. P1715R1 [2] makes the same case.

The authors have use cases that don't fit in the paper (typical for the most *interesting* use cases) where type-based vs variable-based metaprogramming means the difference of 300s compile-times per unit vs. more than an hour (currently by textually duplicating definitions that could have been genericized if variable template template-parameters were available).

Previous works

Variable-template template-parameters were proposed in P2008R0 [5] and were part of the original design for variable templates N3615 [6]. Concept template-parameters have been described by Barry Revzin (back when Concept names were uppercase) in his blog here and here. We mentioned them in P2632R0 [4] and P1985R3 [1].

Universal template-parameters

The fact that variable-template template-parameters and concept template-parameters appear in the same papers is not accidental. For a universal template-parameter to be universal, we need to make sure it covers the set of entities we could possibly want to ever use as template-parameters. There is, therefore, an important order of operations. If we were to add universal template-parameters before concept template-parameters and variable-template template-parameters, we would be in a situation where either

- we can't ever add concept template-parameters and variable-template template-parameters
- "universal template-parameters would not be truly universal"
- we would feel forced to come up with some kind of "more universal template-parameter" syntax

None of these outcomes seems desirable; therefore, the best course of action is to ensure that we support as best we can the full set of entities we might ever want to support as template-parameters, before adding support for universal template-parameters.

Design

Syntax

We propose the following syntax for the declaration of a template head accepting a concept as parameter:

```
template<
    template <template-parameter-list> concept C
>
```

We propose the following syntax for the declaration of a template head accepting a variable template as parameter:

```
template<
    template <template-parameter-list> auto C
>
```

This forms a natural, somewhat intuitive extension of the existing syntax for template extension:

```
template<
    typename T,
    auto V,
    template <template-parameter-list> typename TT,
    template <template-parameter-list> auto VT,
    template <template-parameter-list> concept C,
>
```

Default Arguments

Like type template template parameters, concepts, and variable template template parameters can have a default argument that is a concept name or the name of a variable template respectively. Packs can't be defaulted. (That's a separate paper!)

Usage

Within the definition of a templated entity, a concept template-parameter can be used anywhere a concept name can be used, including as a type constraint, in the requires clause, and so forth.

For example, the following should be valid:

```
template <template <typename T> concept C>
struct S {
```

```
    void f(C auto);

};
```

Concept template-parameters and subsumption

Consider:

```
template <typename T>
requires view<T> && input_range<T>
void f(); // #1

template <typename T>
requires view<T> && contiguous_range<T>
void f(); // #2
```

We expect #2 to be more specialized than #1 because `contiguous_range` subsumes `input_range`.

Now, consider:

```
template <typename T>
requires all_of<T, view, input_range>
void f(); // #1

template <typename T>
requires all_of<T, view, contiguous_range>
void f(); // #2
```

[Run this example on Compiler Explorer]

This example ought to be isomorphic to the previous one, and #2 should still be more specialized than #1. To do that, we need to be able to substitute concept template arguments in constraint expressions when normalizing constraints.

When establishing subsumption, we have historically not substituted template arguments, instead establishing a mapping of template parameters to arguments for each constraint and comparing those mappings.

But to establish subsumption rules for concept template-parameters, we need to depart from that somewhat.

Concepts have the particularity of never being explicitly specialized, deduced, dependent, or even instantiated. Substituting a concept template argument is only a matter of replacing the corresponding template parameter with the list of constraints of the substituted concept, recursively.

As such, subsumption for concept template-parameters does not violate the guiding principle of subsumption.

```
template<typename T, template <typename...> concept C>
```

```
concept range_of = std::ranges::range<T> && C<std::remove_cvref_t<std::ranges::
    range_reference_t<T>>>; // #1

template<typename T>
concept range_of_integrals = std::ranges::range<T> && std::integral<std::remove_cvref_t<std::
    ranges::range_reference_t<T>>>; // #2
```

Note that this transformation does not change any other behavior of normalization, i.e., concept template-parameters that appear within other atomic constraints are not substituted, and arguments that are not concept names are not substituted either.

Subsumption and fold expressions over `&&` and `||`

Before we handle the more complicated scenario where the fold expression's pattern is a pack of concepts, we must fix an old concept-ts issue that fell through the cracks; see ordering of constraints involving fold expressions:

```
template <class T> concept bool A = std::is_move_constructible_v<T>;
template <class T> concept bool B = std::is_copy_constructible_v<T>;
template <class T> concept bool C = A<T> && B<T>;

template <class... T>
requires (A<T> && ...)
void g(T...);

template <class... T>
requires (C<T> && ...)
void g(T...);
```

We want to apply the subsumption rule to the normalized form of the requires clause (and its arguments). As of C++23, the above `g` is ambiguous.

While this is technically a breaking change, it only makes more programs valid. (Where there was simple ambiguity in C++23, we now have a refined partial ordering of that pair.)

This is useful when dealing with algebraic-type classes. Consider a concept constraining a (simplified) environment implementation via a type-indexed `std::tuple`. (In real code, the environment is a type-tag indexed map.)

```
template <typename X, typename... T>
concept environment_of = (... && requires (X& x) { { get<T>(x) } -> std::same_as<T&>; } );

auto f(sender auto&& s, environment_of<std::stop_token> auto env); // uses std::allocator
auto f(sender auto&& s, environment_of<std::stop_token, std::pmr::allocator> auto env); //
    uses given allocator
```

Without the subsumption fixes to fold expressions, the above two overloads conflict, even though they should by rights be partially ordered.

Fold expressions involving concept template-parameters

When the fold expression is over a pack of *concepts*, we must be a bit more careful.

```
template <
    typename T,
    template <typename...> concept... C>
concept all_of = (C<T> && ...);
```

Once substituted, the sequence of binary `&&` or `||` is normalized, `all_of`, `any_of`, and so on can then be implemented in a way that supports subsumption.

One very important case where this facility is absolutely essential is constraining tuples (and other algebraic data-types) by dimension:

```
template <typename X, template <typename> concept... C>
concept product_type_of = (... && C<std::tuple_element_t<C...[?], X>>);
//   index-of-current-element, not proposed, but needed  ~~~~~~~~
```

P2632R0 [4] discusses alternatives to the awful index-of-current-element syntax above.

ADL

Similar to variables, variable templates and concepts are not associated entities when performing argument-dependent lookup. This is consistent with previous work (for example N3595 [3] and P0934R0 [7]) and the general consensus toward ADL.

Deduction and partial ordering

Variable and concept template-parameters should be deducible from a template argument of a class template, used in the argument list of a function, including a function that is synthesized for the purpose of partial ordering; i.e., the intent is that the following is well-formed:

```
template <template <class> auto V, template <class> concept C>
struct A {}; // A takes a variable template template argument

template <template <class T> auto V, template <class> concept C>
void foo(A<V, C>);  // can accept any specialization of A; V and C are deduced

template <class T>
auto Var = 0;

template <class T>
concept Concept = true;

void test() {
    foo(A<Var, Concept>{});
}
```

[Run this example on Compiler Explorer]

Equivalence of atomic constraints

One interesting concept to consider is `tuple_of`, which would e.g., allow constraining a function on a *tuple-like* of integrals, a frequent use case in scientific computation.

In the absence of member and alias packs, a `tuple_like` concept could look like

```
template <typename T, int N>
constexpr bool __tuple_check_elements = [] {
    if constexpr (N == 0)
      return true;
    else if constexpr(requires (T t) {
        typename std::tuple_element_t<N-1, T>;
        { std::get<N-1>(t) };
    })
      return  __tuple_check_elements<T, N-1>;
    return false;
}();

template <typename T>
concept tuple_like = requires {
    typename std::tuple_size<T>::type;
} &&  __tuple_check_elements<T, std::tuple_size_v<T>>;
```

Here, we use a constexpr variable template to check the constraint on individual elements. We can trivially adapt this code to take a concept argument:

```
template <typename T, template <typename> concept C>
concept decays_to = C<std::decay_t<T>>;

template <typename T, int N, template <typename> concept C>
constexpr bool __tuple_check_elements = [] {
    if constexpr (N == 0)
      return true;
    else if constexpr(requires (T t) {
        typename std::tuple_element_t<N-1, T>;
        { std::get<N-1>(t) } -> decays_to<C>;
    })
      return  __tuple_check_elements<T, N-1, C>;
    return false;
}();

template <typename T, template <typename> concept C>
concept tuple_of = requires {
    typename std::tuple_size<T>::type;
} &&  __tuple_check_elements<T, std::tuple_size_v<T>, C>;
```

And this works fine, but `__tuple_check_elements` is an atomic constraint, so we cannot establish a subsumption relationship for this concept.

With a sufficient number of pack features, we could probably write a concept that checks all elements with a single constraint, i.e.,

```cpp
template <typename T, typename E, int N, template <typename> concept C>
concept __tuple_of_element = requires (T t) {
    typename std::tuple_element_t<N, T>;
    { std::get<N>(t) } -> decays_to<C>;
} &&  C<std::tuple_element_t<0, T>>;

template <typename T, template <typename> concept C>
concept tuple_of = requires {
    typename std::tuple_size<T>::type;
} &&  (__tuple_of_element<T, T::[:], current_expansion_index_magic(), C> && ...);
```

But in addition to relying on imaginary features, this is pretty inefficient since ordering complexity would be proportional to the square of the number of tuple elements.

Fortunately, while checking satisfaction does require looking at every element, we can look at just one element to establish subsumption in this particular case.

We can rewrite our concept as

```cpp
template <typename T, int N, template <typename> concept C>
concept __tuple_of_element = requires (T t) {
    typename std::tuple_element_t<N, T>;
    { std::get<N>(t) } -> decays_to<C>;
} &&  C<std::tuple_element_t<0, T>>;

template <typename T, int N, template <typename> concept C>
constexpr bool __check_tuple_elements = [] {
    if constexpr (N == 1)
        return true;
    else if constexpr(__tuple_of_element<T, N-1, C>)
        return  __check_tuple_elements<T, N-1, C>;
    return false;
}();

template <typename T, template <typename> concept C>
concept tuple_of = requires {
    typename std::tuple_size<T>::type;
} &&  (std::tuple_size_v<T>  == 0 || (
         // Check the first element with a concept to establish subsumption
        __tuple_of_element<T, 0, C> &&
        // Check constraint satisfaction for subsequent elements
        __check_tuple_elements<T, std::tuple_size_v<T>, C>
));
```

[Run this example on Compiler Explorer]

For this to work, the concept template-parameter C needs to be substituted in the concept `__tuple_of_element` but not in the atomic constraint `__check_tuple_elements<T, std::tuple_size_v<T>, C>`.

Atomic constraints also need to ignore concept template-parameters for the purpose of comparing their template arguments when establishing atomic constraint equivalence during subsumption.

Partially applied concepts

The first argument of a concept is the entity for which we are checking satisfaction. Concepts can take additional arguments parameterizing the concept.

For example, `invocable<T, int>` checks whether an object of type `T` can be called with an `int` argument.

In specific contexts, a `type-constraint` can appear in a template head, or before `auto` to constrain — or assert — a type. In that case, the compiler injects the type to which the concept is applied, rather than this type being specified by the user.

For example, the following two declarations are equivalent:

```
template<invocable<int> T>
void f();

template<typename T>
void f() requires invocable<T, int>;
```

This is helpful because a great many concepts are binary or n-ary. In the `concepts` header, half of the concepts are not unary.

When passing concepts as template arguments, this becomes quickly apparent. Let us consider a `range_of` concept constraining the range's value type with a concept template-parameter:

```
template <typename R, template <typename> concept C>
concept range_of = std::ranges::range<R> && C<std::ranges::range_value_t<R>>;
```

We can then have an algorithm taking a range of `integral`:

```
auto median(range_of<std::integral> auto&&);
```

But maybe we want to be a little more specific. Can we accept only a range of ints? Given that our `range_of` expects a concept (it should really accept a universal template parameter!), maybe we could use `same_as`. But `same_as` requires arguments. Can we make something like that work?

```
auto f(range_of<same_as<int>> auto&&);
```

We can rewrite our `range_of` concept to take an extra argument pack forwarded to the concept parameter:

```
template <typename R, template <typename...> concept C,  typename... Args>
concept range_of = std::ranges::range<R> && C<std::ranges::range_value_t<R>, Args...>;
void f(range_of<std::same_as, int> auto &&);
```

This is not the clearest interface. And crucially, it doesn't compose. For example, a range of `regular` and also convertible to int type is not expressible.

One workaround would be to lift the concept and its argument into a type:

```
template <template <typename...> concept C, typename... Args>
struct packed_concept {
    template <typename T>
    static constexpr bool apply  = C<T, Args...>;
};

template <typename R, typename... PackedConcept>
concept range_of = std::ranges::range<R>
                   && (PackedConcept::template apply<std::ranges::range_value_t<R>> && ...);
void f(range_of<packed_concept<std::convertible_to, int>,
                packed_concept<std::regular>> auto &&);
```

This works. It is also not particularly usable or readable, and all hope of subsumption is lost. Concepts cannot be class members for a very good reason: It would fundamentally break subsumption. So `apply` is a `bool` variable, an atomic constraint.

Can we still support making something similar in the language? Could we make something like

```
void f(range_of<concept convertible_to<int>> auto)
```

simply work? Yes!

[Run this example on Compiler Explorer]

We can indeed create a new kind of template argument — or a new kind of template-id — that carries additional arguments that are injected when the corresponding concept template-parameter is specialized.

```
// Given this declaration,
template <typename T, template <typename> concept C>
constexpr bool b = C<T>;

// this specialization
b<double, concept std::convertible_to<int>>;

// is rewritten like that by injecting the arguments passed to the concept argument.
template<>
constexpr bool b<double, concept convertible_to<int>> = convertible_to<double, int>;
```

Notice that in this example, `C` is a concept template-parameter that accepts exactly one argument, even though `convertible_to` needs two. The other arguments will be filled in automatically.

In effect, it's like we created a new concept

```
template <typename T>
concept convertible_to_int = convertible_to<T, int>;
```

but we did it inline, directly in the template argument. This feature (which we call partial application) expresses intent very clearly and works with subsumption.

Syntax of partially applied concepts

Partially applied concepts can appear as template arguments matching a concept template-parameter (and nowhere else). You might have noticed the `concept` keyword prefixing template arguments in previous examples and wonder why `range_of<concept convertible_to<int>>` rather than simply `range_of<convertible_to<int>>`?

Of course, there is an undeniable parallel with the syntax of concept declaration, which is nice, but the keyword is there to avoid ambiguity.

If the concept accepts a variable number of arguments (because it has defaulted or variadic parameters), whether `ConceptName<Args>` is a partial application or a complete specialization (which is then a boolean expression) can be ambiguous .

This has not been an issue with *type-constraint*s because they only ever appear in a context where the first parameter is always injected.

However, the ambiguity between a partial concept application and a boolean expression can appear in a couple of cases.

1. In the presence of universal template parameters: Should a given argument matching a UTP be considered a bool or a concept template-parameter?
2. In an overload set, the concept could be matched to both a bool and a concept, as in this example courtesy of Barry Revzin:

   ```
   template <bool B> void f();
   template <template <typename> concept C> void f();
   f<invocable<int>>(); // boolean or partially applied concept?
   ```

We explored various solutions to this ambiguity, including always treating something that *could* be a concept as a concept and forcing an explicit cast to bool to force a boolean expression or making the concept keyword optional when the concept has a fixed number of arguments.

Both approaches suffer the same issue: They could change the meaning of code when the concept is modified by adding defaulted or variadic parameters. It could also break pre-C++26 code. So they do not appear to be robust solutions.

We also refrained from any solution that would make the nature of template arguments somehow deduced from the corresponding parameter during overload resolution, for this would be madness (and it would not really solve the question for universal template parameters).

Ultimately, the `concept` keyword is a great way to show intent and mirrors concept declaration. The following syntax would be valid:

```
some_template_name <
    std::regular,  /// can pass the name directly
    concept regular<>, // no reason this should not work
    invocable<int>, // that is a bool - it will get diagnosed
```

```
                    // if no overload or specialization expect a bool
    concept invocable<int> // ok
>;
```

Generalized partially applied template template-parameters?

A question that arises when considering this partial application of concepts is whether it generalizes to other kinds of template-parameters.

Concepts are unique in that the first parameter has a special meaning, and as such, that you can pass the first parameter at a different time than all other parameters does make a lot of sense. After all, this is why `type-constraints` exist.

To generalize partial application to other template template-parameters, we would need to be able to provide — or not — arbitrary parameters; we could imagine, for example, being able to write some kind of code where some but not all template arguments are provided.

```
Foo<std::map<?, ? , my_comparator>>;
```

However, this is more complicated than what we need for concepts, and whether it would be useful is debatable. How arguments and parameters would be matched is less clear. Besides, unlike concepts, aliases and variable templates can be created at class scope.

Therefore the motivation for partially applied concepts does not necessarily generalize terribly well. We did consider whether this sort of placeholder syntax would be a good fit for concepts in isolation, but we think `concept` *`type-constraint`* is more consistent, less novel, and therefore more teachable than a placeholder syntax.

Packs of partial concepts?

Arguably one could imagine a scenario in which a pack of partial concepts could be constructed:

```
template <typename T, template <typename> concept... Concepts>
concept AllOf  = (Concepts<T> && ...);

template <typename T, template <typename...> concept... Concepts>
concept Foo = AllOf<T, concept Concepts<double>...>;
```

But it seems difficult to imagine a scenario where this would be useful; i.e., when do different concepts accept the same arguments (besides the first one)? Supporting that does not seem like a good time investment.

Status of this proposal and further work

Our main priority should be to make progress on some form of universal template parameters. We will have an implementation of universal template parameters for the Standards Committee meeting in Kona. Before that, we need to ensure concepts and variable-template

template-parameters are supported features so that universal template-parameters support the gamut of entities that could reasonably be used as template-parameters.

As part of that, subsumption for concept template-parameters, as proposed in this paper, as well as subsumption of fold expressions should be considered an integral part of the design since adding them later might be somewhat annoying. (It could affect existing code, in theory.)

However, partial concept application does not affect template-parameters; it's a new kind of template argument or template name. So this can — and probably should — be pursued as a separate effort to avoid blocking progress on universal template-parameters.

Things to be careful about

Concepts have carefully designed limitations aimed to make subsumption possible and reasonably efficient. Care has to be taken not to change that.

In particular, specialization of concepts is not allowed nor is declaring concepts in classes or other non-global (potentially dependent) contexts.

Concept template-parameters need to be substituted when evaluating constraints, but other arguments do not. Efficient memoization is still possible by caching concept template arguments (and only concept template arguments) along the concept.

Concept template-parameters do not allow a concept to refer to itself, i.e., recursion. Universal template-parameters may allow recursion.

```
template <typename T, __any C, typename...Args>
concept Y = C<C, Args...>;

template <typename T, template <typename...> Concept , typename...Args>
concept Foo = Concept<T, Args...>;

Y<int, Foo>;
```

This will need careful consideration, but we have options, such as preventing the same concept name from appearing multiple times or having an implementation-defined limit for how many concepts can be replaced during subsumption.

Implementation

The paper as proposed has been implemented in a fork of Clang and is available on compiler-explorer. The implementation revealed no particular challenge. In particular, we confirmed that the proposed changes do not prevent memoization for subsumption and satisfiability, i.e., a concept and the set of its concept parameters are what needs to be cached.

Partial concept application was also implemented, modeled by a new kind of template argument. This is still a work in progress.

Wording

� Preamble [basic.pre]

Every name is introduced by a *declaration*, which is a

- *name-declaration*, *block-declaration*, or *member-declaration* [dcl.pre,class.mem],
- *init-declarator* [dcl.decl],
- *identifier* in a structured binding declaration [dcl.struct.bind],
- *init-capture* [expr.prim.lambda.capture],
- *condition* with a *declarator* [stmt.pre],
- *member-declarator* [class.mem],
- *using-declarator* [namespace.udecl],
- *parameter-declaration* [dcl.fct],
- *type-parameter* [temp.param],
- *template-template-parameter* [temp.param],
- *elaborated-type-specifier* that introduces a name [dcl.type.elab],
- *class-specifier* [class.pre],
- *enum-specifier* or *enumerator-definition* [dcl.enum],
- *exception-declaration* [except.pre], or
- implicit declaration of an injected-class-name [class.pre].

� Argument-dependent name lookup [basic.lookup.argdep]

For each argument type T in the function call, there is a set of zero or more *associated entities* to be considered. The set of entities is determined entirely by the types of the function arguments (and any ~~template~~ *type-template-parameter* template arguments). Any *typedef-name* s and *using-declaration*s used to specify the types do not contribute to this set. The set of entities is determined in the following way:

- If T is a fundamental type, its associated set of entities is empty.
- If T is a class type (including unions), its associated entities are: the class itself; the class of which it is a member, if any; and its direct and indirect base classes. Furthermore, if T is a class template specialization, its associated entities also include: the entities associated with the types of the template arguments provided for template type parameters; the templates used as template template arguments; and the classes of which any member templates used as template template arguments are members. [*Note:*

Non-type template arguments do not contribute to the set of associated entities. — *end note*]

- If `T` is an enumeration type, its associated entities are `T` and, if it is a class member, the member's class.
- If `T` is a pointer to `U` or an array of `U`, its associated entities are those associated with `U`.
- If `T` is a function type, its associated entities are those associated with the function parameter types and those associated with the return type.
- If `T` is a pointer to a member function of a class `X`, its associated entities are those associated with the function parameter types and return type, together with those associated with `X`.
- If `T` is a pointer to a data member of class `X`, its associated entities are those associated with the member type together with those associated with `X`.

In addition, if the argument is an overload set or the address of such a set, its associated entities are the union of those associated with each of the members of the set, i.e., the entities associated with its parameter types and return type. Additionally, if the aforementioned overload set is named with a *template-id*, its associated entities also include its template *template-argument*s denoting a class template or an alias template and those associated with its type *template-argument* s.

The *associated namespaces* for a call are the innermost enclosing non-inline namespaces for its associated entities as well as every element of the inline namespace set [namespace.def] of those namespaces. Argument-dependent lookup finds all declarations of functions and function templates that

- are found by a search of any associated namespace, or
- are declared as a friend [class.friend] of any class with a reachable definition in the set of associated entities, or
- are exported, are attached to a named module `M` [module.interface], do not appear in the translation unit containing the point of the lookup, and have the same innermost enclosing non-inline namespace scope as a declaration of an associated entity attached to `M` [basic.link].

If the lookup is for a dependent name [temp.dep,temp.dep.candidate], the above lookup is also performed from each point in the instantiation context [module.context] of the lookup, additionally ignoring any declaration that appears in another translation unit, is attached to the global module, and is either discarded [module.global.frag] or has internal linkage.

� The `typedef` specifier [dcl.typedef]

A *simple-template-id* is only a *typedef-name* if its *template-name* names an alias template or ~~a template *template-parameter*~~ a *type-template-parameter* template-parameter. [*Note:* A *simple-template-id* that names a class template specialization is a *class-name* [class.name]. If a *typedef-name* is used to identify the subject of an *elaborated-type-specifier* [dcl.type.elab], a class definition [class], a constructor declaration [class.ctor], or a destructor declaration [class.dtor], the program is ill-formed. — *end note*]

� 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*$_{opt}$ >

template-id:
  simple-template-id
  operator-function-id < *template-argument-list*$_{opt}$ >
  literal-operator-id < *template-argument-list*$_{opt}$ >

template-name:
  identifier

template-argument-list:
  template-argument ...$_{opt}$
  template-argument-list , *template-argument* ...$_{opt}$

template-argument:
  constant-expression
  type-id
  id-expression
  concept-name

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

A *concept-id* is a *simple-template-id* where the *template-name* is a *concept-name* or names a *concept-parameter* . A concept-id is a prvalue of type `bool`, and does not name a template specialization. A concept-id evaluates to `true` if the concept's normalized *constraint-expression* [temp.constr.decl] is satisfied [temp.constr.constr] by the specified template arguments and `false` otherwise. [*Note:* Since a *constraint-expression* is an unevaluated operand, a concept-id appearing in a *constraint-expression* is not evaluated except as necessary to determine whether the normalized constraints are satisfied. — *end note*] [*Example:*

```
template<typename T> concept C = true;
static_assert(C<int>);      // OK
```

— *end example*]

� Template parameters [temp.param]

The syntax for *template-parameter*s is:

template-parameter:
  type-parameter
  parameter-declaration
  template-template-parameter

type-parameter:
  type-parameter-key ...$_{opt}$ *identifier*$_{opt}$
  type-parameter-key identifier$_{opt}$ = *type-id*
  type-constraint ...$_{opt}$ *identifier*$_{opt}$
  type-constraint identifier$_{opt}$ = *type-id*
  ~~*template-head type-parameter-key* ...$_{opt}$ *identifier*$_{opt}$~~
  ~~*template-head type-parameter-key identifier*$_{opt}$ = *id-expression*~~

type-parameter-key:
  `class`
  `typename`

type-constraint:
  nested-name-specifier$_{opt}$ *concept-name*
  nested-name-specifier$_{opt}$ *concept-name* < *template-argument-list*$_{opt}$ >

template-template-parameter:
  type-template-parameter
  variable-template-parameter
  concept-parameter

type-template-parameter:
  template-head type-parameter-key ...$_{opt}$ *identifier*$_{opt}$
  template-head type-parameter-key identifier$_{opt}$ = *id-expression*

variable-template-parameter:
  template-head `auto` ...$_{opt}$ *identifier*$_{opt}$
  template-head `auto` *identifier*$_{opt}$ = *id-expression*

concept-parameter:
  template < *template-parameter-list* > `concept` ...$_{opt}$ *identifier*$_{opt}$
  template < *template-parameter-list* > `concept` *identifier*$_{opt}$ = *id-expression*

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-parameter* can be the product of replacing a >> token by two consecutive > tokens [temp.names]. — *end note*]

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 non-type [*Footnote:* ~~Since template *template-parameters* and template *template-arguments* are treated as types for descriptive purposes, t~~ The terms *non-type parameter* and *non-type argument* are used to refer to non-type, non-template parameters and arguments. — *end note*] *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-parameter* called `T`, rather than an unnamed non-type *template-parameter* of class `T`. — *end example*] A storage class shall not be specified in a *template-parameter* declaration. Types shall not be defined in a *template-parameter* declaration.

The *identifier* in a *type-parameter* is not looked up. A *type-parameter* whose *identifier* does not follow an ellipsis defines its *identifier* to be a *typedef-name* ~~(if declared without `template`) or *template-name* (if declared with `template`)~~ in the scope of the template declaration.

The *identifier* in a *template-template-parameter* is not looked up. A *template-template-parameter* whose *identifier* does not follow an ellipsis defines its *identifier* to be a *template-name*.

[*Note:* A template argument can be a class template or alias template. For example,

```
template<class T> class myarray { /*...*/ };

template<class K, class V, template<class T> class C = myarray>
class Map {
    C<K> key;
    C<V> value;
};
```

— *end note*]

[Editor's note: Modify [temp.param]/p16 As follow]

If a *template-parameter* is ~~a *type-parameter* with an ellipsis prior to its optional *identifier* or is a *parameter-declaration* that declares a pack [dcl.fct]~~

- a *type-parameter* with an ellipsis prior to its optional *identifier*,
- a *parameter-declaration* that declares a pack [dcl.fct], or
- a *template-template-parameter* with an ellipsis prior to its optional *identifier*,

then the *template-parameter* is 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 *type-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*.

� Template template arguments [temp.arg.template]

� General [temp.arg.general]

There are three forms of *template-argument*, corresponding to the three forms of *template-parameter*: type, non-type and template. A template template argument can name a type or alias template, a variable template or a concept.

The type and form of each *template-argument* specified in a *template-id* shall match the type and form specified for the corresponding parameter declared by the template in its *template-*

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

~~A *template-argument* for a template *template-parameter* shall be the name of a class template or an alias template, expressed as *id-expression*.~~

A *template-argument* for a template *template-parameter* shall be an *id-expression* denoting

- the name of a class template or an alias template for a *type-template-parameter*,
- the name of a variable template for a *variable-template-parameter*, and
- the name of a concept for a *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.

Any partial specializations [temp.spec.partial] associated with the primary template are considered when a specialization based on the template *template-parameter* is instantiated. If a specialization is not reachable from the point of instantiation, and it would have been selected had it been reachable, the program is ill-formed, no diagnostic required. [*Example:*

```
template<class T> class A {     // primary template
    int x;
};
template<class T> class A<T*> { // partial specialization
    long x;
};
template<template<class U> class V> class C {
    V<int>  y;
    V<int*> z;
```

```
    };
    C<A> c;              // V<int> within C<A> uses the primary template, so c.y.x has
type int
    // V<int*> within C<A> uses the partial specialization, so c.z.x has type long
```

— *end example*]

Two template parameters are of the same kind if:

- they are both *type-parameter*,
- they are both non-type parameter,
- they are both *type-template-parameter*,
- they are both *variable-template-parameter*, or
- they are both *concept-parameter*.

A template template parameter *P* and a template argument *A* are compatible if

- *A* denotes the name of a class template or alias template and *P* is a *type-template-parameter*,
- *A* denotes the name of a variable template and *P* is a *variable-template-parameter*, or
- *A* denotes the name of a concept and *P* is a *concept-parameter*.

A *template-argument* matches a template *template-parameter* `P` when `A` and `P` are compatible and `P` is at least as specialized as the *template-argument* `A`. In this comparison, if `P` is unconstrained, the constraints on `A` are not considered. If `P` contains a template parameter pack, then `A` also matches `P` if each of `A`'s template parameters matches the corresponding template parameter in the *template-head* of `P`. Two template parameters match if they are of the same kind ~~(type, non-type, template)~~, for non-type *template-parameter*s, their types are equivalent [temp.over.link], and for template *template-parameter*s, each of their corresponding *template-parameter*s matches, recursively. When `P`'s *template-head* contains a template parameter pack [temp.variadic], the template parameter pack will match zero or more template parameters or template parameter packs in the *template-head* of `A` with the same type and form as the template parameter pack in `P` (ignoring whether those template parameters are template parameter packs).

[*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 { /*...*/ };

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
```

— *end example*] [*Example:*

```
template <class T> struct eval;

template <template <class, class...> class TT, class T1, class... Rest>
struct eval<TT<T1, Rest...>> { };

template <class T1> struct A;
template <class T1, class T2> struct B;
template <int N> struct C;
template <class T1, int N> struct D;
template <class T1, class T2, int N = 17> struct E;

eval<A<int>> eA;                // OK, matches partial specialization of eval
eval<B<int, float>> eB;         // OK, matches partial specialization of eval
eval<C<17>> eC;                 // error: C does not match TT in partial specialization
eval<D<int, 17>> eD;            // error: D does not match TT in partial specialization
eval<E<int, float>> eE;         // error: E does not match TT in partial specialization
```

— *end example*] [*Example:*

```
template<typename T> concept C = requires (T t) { t.f(); };
template<typename T> concept D = C<T> && requires (T t) { t.g(); };

template<template<C> class P> struct S { };

template<C> struct X { };
template<D> struct Y { };
template<typename T> struct Z { };

S<X> s1;            // OK, X and P have equivalent constraints
S<Y> s2;            // error: P is not at least as specialized as Y
S<Z> s3;            // OK, P is at least as specialized as Z
```

— *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. If `PP` declares a template parameter pack, then `AA` is the pack expansion `PP...` [temp.variadic]; otherwise, `AA` is the *id-expression* `PP`.

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

� Partial ordering by constraints [temp.constr.order]

A constraint P *subsumes* a constraint Q if and only if, for every disjunctive clause P_i in the disjunctive normal form [*Footnote:* A constraint is in disjunctive normal form when it is a disjunction of clauses where each clause is a conjunction of atomic constraints. For atomic constraints A, B, and C, the disjunctive normal form of the constraint $A \wedge (B \vee C)$ is $(A \wedge B) \vee (A \wedge C)$. Its disjunctive clauses are $(A \wedge B)$ and $(A \wedge C)$. — *end note*] of P, P_i subsumes every conjunctive clause Q_j in the conjunctive normal form [*Footnote:* A constraint is in conjunctive normal form when it is a conjunction of clauses where each clause is a disjunction of atomic constraints. For atomic constraints A, B, and C, the constraint $A \wedge (B \vee C)$ is in conjunctive normal form. Its conjunctive clauses are A and $(B \vee C)$. — *end note*] of Q, where

- a disjunctive clause P_i subsumes a conjunctive clause Q_j if and only if there exists an atomic constraint P_{ia} in P_i for which there exists an atomic constraint Q_{jb} in Q_j such that P_{ia} subsumes Q_{jb}, and
- an atomic constraint A subsumes another atomic constraint B if and only if A and B are identical using the rules described in **??**.

  [Editor's note: Add wording for fold expressions]

[*Example:* Let A and B be atomic constraints [temp.constr.atomic]. The constraint $A \wedge B$ subsumes A, but A does not subsume $A \wedge B$. The constraint A subsumes $A \vee B$, but $A \vee B$ does not subsume A. Also note that every constraint subsumes itself. — *end example*]

[*Note:* The subsumption relation defines a partial ordering on constraints. This partial ordering is used to determine

- the best viable candidate of non-template functions [over.match.best],
- the address of a non-template function [over.over],
- the matching of template template arguments [temp.arg.template],
- the partial ordering of class template specializations [temp.spec.partial.order], and
- the partial ordering of function templates [temp.func.order].

— *end note*]

� Constraint normalization [temp.constr.normal]

The *normal form* of an *expression* `E` is a constraint [temp.constr.constr] that is defined as follows:

- The normal form of an expression `( E )` is the normal form of `E`.
- The normal form of an expression `E1 || E2` is the disjunction [temp.constr.op] of the normal forms of `E1` and `E2`.
- The normal form of an expression `E1 && E2` is the conjunction of the normal forms of `E1` and `E2`.
- The normal form of a concept-id `C<`A_1`,` A_2`, ...,` A_n`>` is the normal form of the *constraint-expression* of `C`, after ~~substituting A_1, A_2, ..., A_n~~
  - substituting each use of A_i's corresponding template parameter in the *constraint-expression* of `C` if A_i denotes a *concept-name*
  - substituting each A_i that is not a *concept-name* for `C`'s respective template parameters in the parameter mappings in each atomic constraint. If any such substitution results in an invalid type or expression, the program is ill-formed; no diagnostic is required.

  [*Example:*

  ```
  template<typename T> concept A = T::value || true;
  template<typename U> concept B = A<U*>;
  template<typename V> concept C = B<V&>;
  ```

  Normalization of `B`'s *constraint-expression* is valid and results in `T::value` (with the mapping `T` $\mapsto$ `U*`) $\vee$ `true` (with an empty mapping), despite the expression `T::value` being ill-formed for a pointer type `T`. Normalization of `C`'s *constraint-expression* results in the program being ill-formed, because it would form the invalid type `V&*` in the parameter mapping. — *end example*]

- The normal form of a *fold-expression* [expr.prim.fold] `F` whose *fold-operator* `op` is either `&&` or `||` and whose pattern is a `concept-name` `C` is the normal form of the expanded expression F′ produced by the expansion of `C`:
  - $(((\mathrm{E}_1\ \mathit{op}\ \mathrm{E}_2)\ \mathit{op}\ \cdots)\ \mathit{op}\ \mathrm{E}_N\ )$ for a unary left fold,
  - $(\ \mathrm{E}_1\ \mathit{op}\ (\cdots\ \mathit{op}\ (\mathrm{E}_{N-1}\ \mathit{op}\ \mathrm{E}_N))\ )$ for a unary right fold,
  - $((((\mathrm{E}\ \mathit{op}\ \mathrm{E}_1)\ \mathit{op}\ \mathrm{E}_2)\ \mathit{op}\ \cdots)\ \mathit{op}\ \mathrm{E}_N\ )$ for a binary left fold, and
  - $(\ \mathrm{E}_1\ \mathit{op}\ (\cdots\ \mathit{op}\ (\mathrm{E}_{N-1}\ \mathit{op}\ (\mathrm{E}_N\ \mathit{op}\ \mathrm{E})))\ )$ for a binary right fold.
- The normal form of any other expression `E` is the atomic constraint whose expression is `E` and whose parameter mapping is the identity mapping.

The process of obtaining the normal form of a *constraint-expression* is called *normalization*. [*Note:* Normalization of *constraint-expression*s is performed when determining the associated constraints [temp.constr.constr] of a declaration and when evaluating the value of an *id-expression* that names a concept specialization [expr.prim.id]. — *end note*]

[*Example:*

```
template<typename T> concept C1 = sizeof(T) == 1;
template<typename T> concept C2 = C1<T> && 1 == 2;
template<typename T> concept C3 = requires { typename T::type; };
template<typename T> concept C4 = requires (T x) { ++x; };

template<C2 U> void f1(U);      // #1
template<C3 U> void f2(U);      // #2
template<C4 U> void f3(U);      // #3
```

The associated constraints of #1 are `sizeof(T) == 1` (with mapping $\mathtt{T} \mapsto \mathtt{U}$) $\wedge$ `1 == 2`.
The associated constraints of #2 are `requires { typename T::type; }` (with mapping $\mathtt{T} \mapsto \mathtt{U}$).
The associated constraints of #3 are `requires (T x) { ++x; }` (with mapping $\mathtt{T} \mapsto \mathtt{U}$). — *end example*]

� Variadic templates [temp.variadic]

A *template parameter pack* is a template parameter that accepts zero or more template arguments. [*Example:*

```
template<class ... Types> struct Tuple { };

Tuple<> t0;                     // Types contains no arguments
Tuple<int> t1;                  // Types contains one argument: int
Tuple<int, float> t2;           // Types contains two arguments: int and float
Tuple<0> error;                 // error: 0 is not a type
```

— *end example*]

A *function parameter pack* is a function parameter that accepts zero or more function arguments. [*Example:*

```
template<class ... Types> void f(Types ... args);

f();                            // args contains no arguments
f(1);                           // args contains one argument: int
f(2, 1.0);                      // args contains two arguments: int and double
```

— *end example*]

An *init-capture pack* is a lambda capture that introduces an *init-capture* for each of the elements in the pack expansion of its *initializer*. [*Example:*

```
template <typename... Args>
void foo(Args... args) {
```

```
    [...xs=args]{
        bar(xs...);             // xs is an init-capture pack
    };
}

foo();                          // xs contains zero init-captures
foo(1);                         // xs contains one init-capture
```

— end example]

A *pack* is a template parameter pack, a function parameter pack, or an *init-capture* pack. The number of elements of a template parameter pack or a function parameter pack is the number of arguments provided for the parameter pack. The number of elements of an *init-capture* pack is the number of elements in the pack expansion of its *initializer*.

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 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 *template-template-parameter* without the ellipsis.
- In an *initializer-list* [dcl.init]; the pattern is an *initializer-clause*.
- In a *base-specifier-list* [class.derived]; the pattern is a *base-specifier*.
- In a *mem-initializer-list* [class.base.init] for a *mem-initializer* whose *mem-initializer-id* denotes a base class; the pattern is the *mem-initializer*.
- In a *template-argument-list* [temp.arg]; the pattern is a *template-argument*.
- In an *attribute-list* [dcl.attr.grammar]; the pattern is an *attribute*.
- In an *alignment-specifier* [dcl.align]; the pattern is the *alignment-specifier* without the ellipsis.
- In a *capture-list* [expr.prim.lambda.capture]; the pattern is the *capture* without the ellipsis.
- In a `sizeof...` expression [expr.sizeof]; the pattern is an *identifier*.

- In a *fold-expression* [expr.prim.fold]; the pattern is the *cast-expression* that contains an unexpanded pack.

[*Example:*

```
template<class ... Types> void f(Types ... rest);
template<class ... Types> void g(Types ... rest) {
    f(&rest ...);     // ``&rest ...'' is a pack expansion; ``&rest'' is its pattern
}
```

— *end example*]

� Type equivalence [temp.type]

Two *template-id*s are the same if

- their *template-name*s, *operator-function-id*s, or *literal-operator-id*s refer to the same template, and
- their corresponding type *template-argument*s are the same type, and
- their corresponding non-type *template-argument*s are template-argument-equivalent (see below) after conversion to the type of the *template-parameter*, and
- their corresponding template *template-argument*s refer to the same template.

Two *template-id*s that are the same refer to the same class, function, concept, or variable.

� Partial ordering of function templates [temp.func.order]

If multiple function templates share a name, the use of that name can be ambiguous because template argument deduction [temp.deduct] may identify a specialization for more than one function template. *Partial ordering* of overloaded function template declarations is used in the following contexts to select the function template to which a function template specialization refers:

- during overload resolution for a call to a function template specialization [over.match.best];
- when the address of a function template specialization is taken;
- when a placement operator delete that is a function template specialization is selected to match a placement operator new [basic.stc.dynamic.deallocation,expr.new];
- when a friend function declaration [temp.friend], an explicit instantiation [temp.explicit] or an explicit specialization [temp.expl.spec] refers to a function template specialization.

Partial ordering selects which of two function templates is more specialized than the other by transforming each template in turn (see next paragraph) and performing template argument deduction using the function type. The deduction process determines whether one of the templates is more specialized than the other. If so, the more specialized template is the one chosen by the partial ordering process. If both deductions succeed, the partial ordering selects the more constrained template (if one exists) as determined below.

To produce the transformed template, for each type, non-type, or template template parameter (including template parameter packs [temp.variadic] thereof) synthesize a unique type, value, or class template respectively and substitute it for each occurrence of that parameter in the function type of the template.

[Editor's note: Do we need to change anything here?]

[*Note:* The type replacing the placeholder in the type of the value synthesized for a non-type template parameter is also a unique synthesized type. — *end note*] Each function template M that is a member function is considered to have a new first parameter of type $X(M)$, described below, inserted in its function parameter list. If exactly one of the function templates was considered by overload resolution via a rewritten candidate [over.match.oper] with a reversed order of parameters, then the order of the function parameters in its transformed template is reversed. For a function template M with cv-qualifiers *cv* that is a member of a class A:

� Deducing template arguments from a type [temp.deduct.type]

[Editor's note: Modify [temp.deduct.type]/p8 As follow]

A template type argument `T`, a template template argument denoting a class template or an alias template `TT`, or a template non-type argument `i` can be deduced if `P` and `A` have one of the following forms:

cv_{opt} T
T*
T&
T&&
$\mathtt{T}_{opt}$ [$\mathtt{i}_{opt}$]
$\mathtt{T}_{opt}$ ($\mathtt{T}_{opt}$) noexcept($\mathtt{i}_{opt}$)
$\mathtt{T}_{opt}$ $\mathtt{T}_{opt}$ ::*
$\mathtt{TT}_{opt}$ <T>
$\mathtt{TT}_{opt}$ <i>
$\mathtt{TT}_{opt}$ <TT>
$\mathtt{TT}_{opt}$ <>

Acknowledgments

Many people contributed valuable discussions and feedbacks to this paper, notably Nina Dinka Ranns, Alisdair Meredith, Joshua Berne, Pablo Halpern, Lewis Baker, Bengt Gustafsson, Hannes Hauswedell and Barry Revzin.

We also want to thank Lori Hughes for helping editing this paper and Bloomberg for sponsoring this work.

References

[1] Gašper Ažman, Mateusz Pusz, Colin MacLean, Bengt Gustafsonn, and Corentin Jabot. P1985R3: Universal template parameters. https://wg21.link/p1985r3, 9 2022.

[2] Jorg Brown. P1715R1: Loosen restrictions on "_t" typedefs and "_v" values. https://wg21.link/p1715r1, 2 2023.

[3] Peter Gottschling. N3595: Simplifying argument-dependent lookup rules. https://wg21.link/n3595, 3 2013.

[4] Corentin Jabot, Pablo Halpern, John Lakos, Alisdair Meredith, Joshua Berne, and Gašper Ažman. P2632R0: A plan for better template meta programming facilities in c++26. https://wg21.link/p2632r0, 10 2022.

[5] Mateusz Pusz. P2008R0: Enable variable template template parameters. https://wg21.link/p2008r0, 1 2020.

[6] Gabriel Dos Reis. N3615: Constexpr variable templates. https://wg21.link/n3615, 3 2013.

[7] Herb Sutter. P0934R0: A modest proposal: Fixing adl. https://wg21.link/p0934r0, 2 2018.