Concept and variable-template template-parameters

Document #: P2841R3
Date: 2024-05-22
Programming Language C++
Audience: EWG
Reply-to: Corentin Jabot <corentin.jabot@gmail.com>
Gašper Ažman <gasper.azman@gmail.com>
James Touton <bekenn@gmail.com>

Abstract

C++ allows passing templates as template parameters. However, they are forced to be typenames (either type alias templates or class templates). Variable templates or concepts are not supported. This is a hole in the template facilities and is the topic of this paper.

We introduce a way to pass concepts and variable templates as template parameters.

Example:

```
template<
   template <typename T> concept C,
   template <typename T> auto C
>
struct S{};

template <typename T>
concept Concept = true;

template <typename T>
constexpr auto Var = 42;

S<Concept, Var> s;
```

Note: this paper is a subset of the larger P1985R3 [1] (Universal Template Parameters); the authors felt this topic is subtle enough to warrant its own paper.

Revisions

R3

- Wording improvements

R2

- Alter the design so that partial ordering remains independent of template arguments, following guidance given in Kona.
- Add a section on the deduction of template parameters from the arguments of a variable template/concept specialization

R1

- Add examples, motivation
- Wording improvement

R0

- Initial revision

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.

While there are workarounds - for example by wrapping a variable in a struct with a `value` member that can then be passed as a type template template parameter, these workarounds all suffer the same limitations:

- They have terrible ergonomics
- They have a noticeable impact on performance - instantiating types is expensive
- They do not allow to take advantage of nice concept properties such as terse syntax and subsumption.

All of these limitations of available patterns are additional motivations for this proposal.

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:

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

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

- `range_of<Concept>`
  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<Concept>`
  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 as for having 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 [6]) are useful by themselves. 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 in addition to generating the value for the constant, which is strictly more work than just producing the constant. For a 'bool' constant, for instance, the difference is substantial; on Apple clang15, it's about 60% (so, less than half the time). The memory footprint is more difficult to gauge, but it seems around a 40% difference.

This performance issue is also explored in more detail in P1715R1 [2].

In other microbenchmarks, Gašper has observed a minimum of 30% speedup by not instantiating class bodies, and a 50% memory usage reduction for programs with heavy traits usage, specifically when implementing P2300-like classes.

Also, if one has multiple metaprogramming libraries, relying on idioms like `::value` is fundamentally less composable than a value just being a value. Similarly, if you have a concept in your codebase, you shouldn't have to wrap it into a static constexpr `::value` member of a type to pass it to a metafunction.

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 that 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 is 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>
constexpr 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>
constexpr 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 [**?**] makes the same case.

Additional examples

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).

Terse syntax, overloading, and reusing existing concepts

The following example, simplified from production code shows multiple interesting properties of concept template parameters. `with_values_t` takes a function and a predicate, and calls the function with all the arguments satisfying this predicate.

Here we demonstrate the function with `either` and `maybe`, but in reality, this is used with receiver types - which are also monadic. The call operator applies `f` to all engaged arguments. But all the arguments must be of the same shape (all optionals, all expected), etc.

To do that, we here use the abbreviated function template syntax with `type-constraints`, which is only possible with concept template parameters.

```
template <typename T>
struct maybe;
template <typename L, typename R>
struct either;

template <typename T>
concept a_maybe = /*...*/;
template <typename T>
concept an_either = /*...*/;

template <template <typename> concept C>
struct _with_values_t {
  static constexpr auto operator()(auto&& f, C auto&& v, C auto&& ... vs) -> decltype(auto) {
    if (is_active<C>(e)) {       // does the active type in the variant satisfy C
        return _with_values_t{}(bind_front(f, *v), FWD(vs)...);
    } else {
        return _with_values_t{}(f, FWD(vs)...);
    }
  }
};

// have to enforce it's the same monad or it doesn't make any sense
inline constexpr struct with_values_t : _with_values_t<a_maybe>, _with_values_t<an_either> {
  using _with_values_t<a_maybe>::operator();
  using _with_values_t<an_either>::operator();
} with_values {};
```

It would be technically possible to use a type instead here

```
template <typename T>
struct an_either_t {
    static constexpr bool value = an_either<T>;
};

struct _with_values_t {
    template <typename First, typename... Tail>
    requires (an_either_t<First>::value && (an_either_t<Tail>::value && ...))
    static constexpr auto operator()(auto&& f, C auto&& e, C auto&& ... es) -> decltype(auto);

};
```

But again:

- This is much less ergonomic as it forces users to wrap their concepts in types which is not intuitive (ie we have found that difficult to teach).
- The necessity of introducing new names for the same predicate - just exposed as a type, concept, or variable - adds unnecessary complexity to APIs
- Composability only works by convention.
- Creating types has a significant performance impact on compile times
- Diagnostic messages are slightly worse than they could be because of the added layers of wrapping and because compilers will decompose concepts in diagnostic messages.

When life gives you Lambdas

To work around the lack of concept parameters, users have started to use generic lambdas

```
template <typename T, auto ConceptWrapperLambda>
concept decays_to = requires {
    ConceptWrapperLambda.template operator()<std::decay_t<T>>();
};
template <class T>
requires decays_to<T, ([]<std::copyable>(){})>
auto f(T&& x) {}
```

Here the concepts we want to parametrize on are passed as a constrained generic lambda - which we then try to call when checking our higher-level concepts. This allows not to have to create a new type for each concept, so it might be slightly easier to use, although the reader will agree that it particularly arcane. In addition to the usability concerns, lambdas are never a solution to compile times performance.

All the existing work arounds suffer similar performance and usability concerns, and of course none support subsumption. Yet, many such workarounds have been developed and a number of them have been deployed in production. Daisy Hollman provided an entire collection of such workarounds.

Previous work

Variable-template template-parameters were proposed in P2008R0 [6] and were part of the original design for variable templates N3615 [7]. 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 [5] 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 want to 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 a 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
>
```

Note that because variable templates and their type can be arbitrarily specialized, `auto` here acts only as a syntactic marker and cannot be replaced by a *type-id*.

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.

For example, a `range_of_integrals` defined as follow:

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

Can be mechanically lifted:

```
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>>>;
template<typename T>
concept range_of_integrals = range_of<T, std::integral>;
```

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.

Fold expressions involving concept template-parameters

Our proposed design allow for subsumption in the the presence of fold expressions whose pattern is a concept. (For the non-concept case, see P2963R0 [4])

```
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 [5] 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 [8]) and the general consensus toward ADL.

Deduction of concept and template parameters

Variable and concept template-parameters should be deducible from a template argument of a class template, used in the argument list of a function.

```
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]

Partial ordering of function templates involving concept template parameters

Let us introduce three concepts that refine eachother `A`, `B`, and `C`, as well as a class template `S` that carries a concept `X` and a type `T`.

If we then define an overload set of two functions where one deduces the concept, we get into an interesting situation where if concept parameters are allowed participate in partial ordering, the choice of template arguments of `S` can change the subsumption order.

```
template <template <typename T> concept X, typename T>
struct S {};
template <typename T>
concept A = true;
template <typename T>
concept B = true && A<T>;
template <typename T>
concept C = true && B<T>;

template <template <typename T>  concept X, typename T>
int answer(S<X, T>) requires B<T> { return 42; }
template <template <typename T> concept X, typename T>
int answer(S<X, T>) requires X<T> { return 43; }

answer(S<A, int>{});
answer(S<C, int>{});
answer(S<B, int>{});
```

In a previous version of this proposal, we proposed that the concept template argument (`A`, `B`, `C` respectively) would be substituted in each viable `answer overload` before determining partial ordering.

However, historically, it was always possible to determine the partial ordering of two function templates before substitution, and independently of any template argument. This has notably allowed compilers to cache partial orderings of function templates, and even though the compiler isn't confused, one might legitimately be concerned that the users might be. On the face of it, it seems valuable for a C++ programmer to be able to partially order function templates in their head, and this feature seems to allow a corner-case where that is impossible before substitution.

It was always the position of the authors that use cases where concepts are deduced from functions arguments were contrived but we did not want to outright limit the set of places were a concept template parameter could be used, and it took us a while to find a reasonable way to resolve these opposite design goals.

Ultimately we found a solution that preserves all the uses cases this feature was designed for, while not making partial ordering dependent on arguments.

The rule we are proposing is:

Given a template declaration `D` with a concept parameter `C`, if `C` appears in the associated constraints of `D`, then `D` is never at least as constrained as another constrained declaration. In the example above, the 3 calls to `answer` are, with is rule, ambiguous.

This rule makes any overload that references a concept template parameter in its requires clause unorderable solely based on subsumption.

We think this has nice properties:

- It’s fairly straightforward to teach
- It’s easy to produce a good diagnosis for.
- It leaves the design space open.

Consider this slightly different example:

```
template <template <typename> concept C>
concept A = C<int>;
template <template <typename> concept C>
concept B = true && A<C>;

template <template <typename T> concept X>
void f() {}; // #1
template <template <typename T> concept X>
void f() requires A<X> {} // #2
template <template <typename T> concept X>
void f() requires B<X> {} // #3

template <typename T>
concept Foo = true;

f<Foo>(); // #4 (ambiguous between 2 and 3)
```

Here, #2 and #3 are more specialized than #1 (because they are constrained and #1 is not).

With the rule proposed above, neither #2 or #3 are as least as constrained as each other (as they refer to a concept templaste parameter `X`). As such #2 is not more specialized than #3 and #3 is not more specialized than #2, and the call #4 is ambiguous.

We could conceive an alternative design instead, such that we would **consider dependent `concept-id` (ie dependent on a concept template parameter of the function template) to be atomic constraints** (option 2).

With that alternative design, for the example above the associated constraints of #2 would be, after normalization `C<int>` (where `C<int>` is an atomic constraint and `C` refers to some invented template argument), and the associated constraints of #3 would be, after normalization `true && C<int>` (where `C<int>` is the same expression as #2’s).

In that model, #3 subsumes #2 and the call is not ambiguous. The key observation is that, the nature of `C` does not affect subsumption whether it would be substituted or not.

Not looking at template arguments (and considering dependent `concept-id`) can lead to situations where overloads are ambiguous, when they would not be if the concept argument was written verbatim and not passed via a parameter.

```
template <typename T>
concept Foo = true;

template <template <typename> concept C>
concept B = true && C<int>;
```

```cpp
template <template <typename T> concept X>
void f() requires Foo<int>{}; // #1

template <template <typename T> concept X>
void f() requires B<X> {} // #2

f<Foo>(); // ambiguous between #1 and #2
```

The opposite is not possible.

There is a compromise between these two options. We could consider that a concept that appears (either as concept-id, or as concept template argument of another concept-id) in a subexpression of `&&` or `||` makes that subexpression atomic (and that subexpression only).

```cpp
template <template <typename> concept X>
concept AlwaysTrue = true; // X is not used
template <typename T>
concept A = true;
template <typename T, template <typename> concept C>
void f(T) requires
A<T>
|| C<int>  // atomic (depends on C)
|| AlwaysTrue<T, C> {}  // atomic (depends on C, even if C is never used by AlwaysTrue)
```

This would be less restrictive than Option 1, and less precise than Option 2, but easier to implement. Lets refer to this option as 1B.

In no case do we expand concept template arguments when considering subsumption; the question is merely about how much subsumption depth we want to preserve, that is, how much rope for resolving ambiguity do we want to give users.

Ultimately, while whe have a slight preference for option 1.

In the previous revision of this paper, EWG was asked to choose between these options:

- Option 1: Don't try to determine a more constrained overload at all in the presence of a referenced concept template parameter.
- Option 1B: Before normalization (ie at the top level), if a concept template parameter is referenced in the subexpression of of a logical && or ||, consider that subexpression atomic
- Option 2: After normalization of non-dependent concept-id, consider concept-id refering to a concept template parameter to be atomic constraints. (Option 1 and 1B can be evolved into option 2 later, the opposite would be a breaking change.)

EWG choose the first option.

Deduction of template parameters from the argument list of a variable template argument

This is not proposed.

Consider:

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

template<
template <typename...> auto v,
typename... Args
>
inline constexpr bool is_specialization_of_v<v, v<Args...>> = true; // #2

template <typename T>
constexpr int i = 42;

static_assert(is_specialization_of_v<i, i<int>>); // #3
```

[Compiler Explorer]

Should we be able to deduce `Args` from `int`? Some existing implementations will eagerly substitute `i<int>` by its value (here, `42`), such that there is subsequently nothing left to deduce `Args` against.

While it would be possible to make that work, the implementation effort is non-negligible and the benefits limited, as we could only deduce the arguments of entities that are valid template arguments - which sounds obvious but that means that the above example can only work on a subset of variables (constexpr variables template specialization of structural types).

We would also need to decide whether `is_specialization_of_v<i, i<int>>` behaves differently from `is_specialization_of_v<i, (i<int>)>` and how that generalizes to arbitrary subexpressions involving variable template specializations.

So, for now, arguments of variable template template parameters are not deduced. instead, we should make #2 ill-formed, so that we have the opportunity to extend that at a later time if we find sufficient motivation for it.

There are existing cases where we make non-deductible partial specializations ill-formed (see [temp.spec.partial.match]), however in the general case we don't seem to (for example here is an example with a non-deducible pack)

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.,

```
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

```
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.

Status of this proposal and further work

Our main priority should be to make progress on some form of universal template parameters.

This paper has been implemented in an experimental version of clang, available on godbolt.

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 challenging, although it should not affect existing valid code.

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.

Wording

1. Add a grammar production for qualified *concept-names*

� Concept definitions [temp.concept]

concept-definition:
    `concept` *concept-name attribute-specifier-seq*$_{opt}$ = *constraint-expression* ;

qualified-concept-name:
    nested-name-specifier$_{opt}$ *concept-name*

concept-name:
    identifier

[*Editor's note:* In [temp.param], use the new production]

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

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

2. Unifying existing terminology

The introduction of new non-type, non non-type template parameters might lead to confusion given the state of the current terminology. Beside actually specifying the behavior of concept parameters and variable template parameters, we recommend renaming non-type template parameters and distinguishing type template parameters from type-parameters that are not types.

If it is not a type, what is it?

A non-type template parameter is neither an expression nor an object. It might be a variable but a non-type template argument is not. ”value template parameter” might work, but we talk in a few places of the ”value of a template parameter”, and talking about the ”value of a value template-parameter” might be a new source of confusion. Ultimately, we propose ”constant template parameter”.

This gives us:

Grammar	Prose
type-parameter	type template parameter
parameter-declaration	constant template parameter
type-template-parameter	type template template parameter
variable-template-parameter	variable template template parameter
concept-template-parameter	concept template parameter

3. Wording for variable-template and concept template parameters

� 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],
- *type-template-parameter* [temp.param],
- *variable-template-parameter* [temp.param],
- *concept-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]

When the *postfix-expression* in a function call[expr.call] is an *unqualified-id*, and unqualified lookup[basic.lookup.unqual] for the name in the *unqualified-id* does not find any

- declaration of a class member, or
- function declaration inhabiting a block scope, or
- declaration not of a function or function template

then lookup for the name also includes the result of *argument-dependent lookup* in a set of associated namespaces that depends on the types of the arguments (and for type template template arguments, the namespace of the template argument), as specified below.

[...]

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* 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 type template template arguments; and the classes of which any member templates used as type template template arguments are members. [*Note:* ~~Non-type~~ constant template arguments, variable template template arguments and concept 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 type template *template-argument*s 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.

� User-defined literals [lex.ext]

If *L* is a *user-defined-string-literal*, let *str* be the literal without its *ud-suffix* and let *len* be the number of code units in *str* (i.e., its length excluding the terminating null character). If *S* contains a literal operator template with a ~~non-type~~ constant template parameter for which *str* is a well-formed *template-argument*, the literal *L* is treated as a call of the form

```
operator ""X<str>()
```

Otherwise, the literal *L* is treated as a call of the form

```
operator ""X(str, len)
```

� Constant expressions [expr.const]

A *converted constant expression* of type `T` is an expression, implicitly converted to type `T`, where the converted expression is a constant expression and the implicit conversion sequence contains only

- user-defined conversions,
- [...]

- function pointer conversions[conv.fctptr],

and where the reference binding (if any) binds directly. [*Note:* Such expressions can be used in `new` expressions[expr.new], as case expressions[stmt.switch], as enumerator initializers if the underlying type is fixed[dcl.enum], as array bounds[dcl.array], and as ~~non-type~~ constant template arguments[temp.arg]. — *end note*]

� 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* . [*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*]

� Decltype specifiers [dcl.type.decltype]

For an expression E, the type denoted by `decltype(`E`)` is defined as follows:

- if E is an unparenthesized *id-expression* naming a structured binding[dcl.struct.bind], `decltype(`E`)` is the referenced type as given in the specification of the structured binding declaration;
- otherwise, if E is an unparenthesized *id-expression* naming a ~~non-type~~ constant *template-parameter* [temp.param], `decltype(`E`)` is the type of the *template-parameter* after performing any necessary type deduction[dcl.spec.auto,dcl.type.class.deduct];
- otherwise, if E is an unparenthesized *id-expression* or an unparenthesized class member access[expr.ref], `decltype(`E`)` is the type of the entity named by E. If there is no such entity, the program is ill-formed;
- otherwise, if E is an xvalue, `decltype(`E`)` is `T&&`, where `T` is the type of E;
- otherwise, if E is an lvalue, `decltype(`E`)` is `T&`, where `T` is the type of E;
- otherwise, `decltype(`E`)` is the type of E.

� Placeholder type deduction [dcl.type.auto.deduct]

Placeholder type deduction is the process by which a type containing a placeholder type is replaced by a deduced type.

A type `T` containing a placeholder type, and a corresponding *initializer-clause* E, are determined as follows:

- For a non-discarded `return` statement that occurs in a function declared with a return type that contains a placeholder type, `T` is the declared return type.
- [...]

- For a ~~non-type~~ constant template parameter declared with a type that contains a placeholder type, T is the declared type of the ~~non-type~~ constant template parameter and E is the corresponding template argument.

� Structured binding declarations [dcl.struct.bind]

Otherwise, if the *qualified-id* std::tuple_size<E> names a complete class type with a member named value, the expression std::tuple_size<E>::value shall be a well-formed integral constant expression and the number of elements in the *attributed-identifier-list* shall be equal to the value of that expression. Let i be an index prvalue of type std::size_t corresponding to v$_i$. If a search for the name get in the scope of E[class.member.lookup] finds at least one declaration that is a function template whose first template parameter is a ~~non-type~~ constant template parameter, the initializer is *e*.get<i>(). Otherwise, the initializer is get<i>(*e*), where get undergoes argument-dependent lookup[basic.lookup.argdep].

� Address of an overload set [over.over]

An *id-expression* whose terminal name refers to an overload set S and that appears without arguments is resolved to a function, a pointer to function, or a pointer to member function for a specific function that is chosen from a set of functions selected from S determined based on the target type required in the context (if any), as described below. The target can be

- an object or reference being initialized [dcl.init,dcl.init.ref,dcl.init.list],
- [...]
- a ~~non-type~~ constant *template-parameter* [temp.arg.nontype].

The *id-expression* can be preceded by the & operator.

� User-defined literals [over.literal]

A *numeric literal operator template* is a literal operator template whose *template-parameter-list* has a single *template-parameter* that is a ~~non-type~~ constant template parameter pack[temp.variadic] with element type char. A *string literal operator template* is a literal operator template whose *template-parameter-list* comprises a single ~~non-type~~ constant *template-parameter* of class type. The declaration of a literal operator template shall have an empty *parameter-declaration-clause* and shall declare either a numeric literal operator template or a string literal operator template.

� Template parameters [temp.param]

The syntax for *template-parameter*s is:

template-parameter:
  type-parameter
  parameter-declaration
  type-template-parameter
  variable-template-parameter
  concept-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:
  qualified-concept-name
  qualified-concept-name < *template-argument-list*$_{opt}$ >

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*$_{opt}$ = *qualified-concept-name*

A *template template parameter* ia a *type-template-parameter*, a *variable-template-parameter*, or a *concept-parameter*. A *constant template parameter* is a template parameter introduced by a *parameter-declaration*.

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*~~ template template 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 *parameter-declaration*~~ constant template parameter.

[*Editor's note:* Remove the footnote]

[*Footnote:* ~~Since template *template-parameters* and template *template-arguments* are treated as types for descriptive purposes, t~~he terms *non-type parameter* and *non-type argument* are used to refer to non-type, non-template parameters and arguments. — *end note*]

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~~ constant *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. `P` whose *identifier* does not follow an ellipsis defines its *identifier* to be a *concept-name* `T` if `P` is a *concept-parameter* or a *template-name* `T` otherwise. `T` is declared in the scope of the template declaration.

[*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*]

[...]

A ~~non-type~~ constant *template-parameter* shall have one of the following (possibly cv-qualified) types:

- a structural type (see below),
- a type that contains a placeholder type[dcl.spec.auto], or
- a placeholder for a deduced class type[dcl.type.class.deduct].

The top-level *cv-qualifier*s on the *template-parameter* are ignored when determining its type.

A *structural type* is one of the following:

- a scalar type, or
- an lvalue reference type, or
- a literal class type with the following properties:
  - all base classes and non-static data members are public and non-mutable and

- the types of all bases classes and non-static data members are structural types or (possibly multidimensional) array thereof.

An *id-expression* naming a ~~non-type~~ constant *template-parameter* of class type T denotes a static storage duration object of type const T, known as a *template parameter object*, which is template-argument-equivalent[temp.type] to the corresponding template argument after it has been converted to the type of the *template-parameter* [temp.arg.nontype]. No two template parameter objects are template-argument-equivalent. [*Note:* If an *id-expression* names a ~~non-type~~ non-reference constant *template-parameter*, then it is a prvalue if it has non-class type. Otherwise, if it is of class type T, it is an lvalue and has type const T [expr.prim.id.unqual]. — *end note*]

[*Example:*

```
using X = int;
struct A {};
template<const X& x, int i, A a> void f() {
    i++;                          // error: change of template-parameter value

    &x;                           // OK
    &i;                           // error: address of non-reference
template-parameter
    &a;                           // OK
    int& ri = i;                  // error: attempt to bind non-const reference to
temporary
    const int& cri = i;           // OK, const reference binds to temporary
    const A& ra = a;              // OK, const reference binds to a template parameter
object
}
```

— *end example*]

[*Note:* A ~~non-type~~ constant *template-parameter* cannot be declared to have type *cv* void. [*Example:*

```
template<void v> class X;       // error
template<void* pv> class Y;     // OK
```

— *end example*] — *end note*]

A ~~non-type~~ constant *template-parameter* of type "array of T" or of function type T is adjusted to be of type "pointer to T". [*Example:*

```
template<int* a>   struct R {  };
template<int b[5]> struct S {  };
int p;
R<&p> w;                       // OK
S<&p> x;                       // OK due to parameter adjustment
int v[5];
R<v> y;                        // OK due to implicit argument conversion
S<v> z;                        // OK due to both adjustment and conversion
```

— *end example*]

A ~~non-type~~ constant template parameter declared with a type that contains a placeholder type with a *type-constraint* introduces the immediately-declared constraint of the *type-constraint* for the invented type corresponding to the placeholder [dcl.fct].

A *default template argument* is a template argument [temp.arg] specified after = in a *template-parameter*. A default template argument may be specified for any kind of *template-parameter* (type, ~~non-type~~ constant, template) that is not a template parameter pack[temp.variadic]. A default template argument may be specified in a template declaration. A default template argument shall not be specified in the *template-parameter-list*s of the definition of a member of a class template that appears outside of the member's class. A default template argument shall not be specified in a friend class template declaration. If a friend function template declaration D specifies a default template argument, that declaration shall be a definition and there shall be no other declaration of the function template which is reachable from D or from which D is reachable.

[...]

When parsing a default template argument for a ~~non-type~~ constant *template-parameter*, the first non-nested > is taken as the end of the *template-parameter-list* rather than a greater-than operator. [*Example:*

```
template<int i = 3 > 4 >        // syntax error
class X {  };

template<int i = (3 > 4) >      // OK
class Y {  };
```

— *end example*]

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], 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*. [*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 non-type 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 non-type 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*]

[...]

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*~~ 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*.

[...]

� 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
  qualified-concept-name

[…]

A *concept-id* is a *simple-template-id* where the *template-name* is a *concept-name*. 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 arguments [temp.arg]

� General [temp.arg.general]

There are three forms of *template-argument*, corresponding to the three forms of *template-parameter*: type, ~~non-type~~ constant 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*]

[...]

� Template type arguments [temp.arg.type]

A *template-argument* for a ~~*template-parameter* which is a type~~ *type-parameter* shall be a *type-id*.

[*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
}
```

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

[...]

� Template ~~non-type~~ constant arguments [temp.arg.nontype]

If the type `T` of a *template-parameter* [temp.param] contains a placeholder type [dcl.spec.auto] or a placeholder for a deduced class type [dcl.type.class.deduct], the type of the parameter is the type deduced for the variable `x` in the invented declaration

`T x =` E `;`

where E is the template argument provided for the parameter. [*Note:* E is a *template-argument* or (for a default template argument) an *initializer-clause*. — *end note*] If a deduced parameter type is not permitted for a *template-parameter* declaration [temp.param], the program is ill-formed.

The value of a ~~non-type~~ constant *template-parameter* P of (possibly deduced) type `T` is determined from its template argument A as follows. If `T` is not a class type and A is not a *braced-init-list*, A shall be a converted constant expression [expr.const] of type `T`; the value of P is A (as converted).

Otherwise, a temporary variable

```
constexpr T v = A;
```

is introduced. The lifetime of `v` ends immediately after initializing it and any template parameter object (see below). For each such variable, the *id-expression* `v` is termed a *candidate initializer*.

If `T` is a class type, a template parameter object[temp.param] exists that is constructed so as to be template-argument-equivalent to `v`; P denotes that template parameter object. P is copy-initialized from an unspecified candidate initializer that is template-argument-equivalent to `v`. If, for the initialization from any candidate initializer,

- the initialization would be ill-formed, or
- the full-expression of an invented *init-declarator* for the initialization would not be a constant expression when interpreted as a *constant-expression* [expr.const], or
- the initialization would cause P to not be template-argument-equivalent[temp.type] to `v`,

the program is ill-formed.

Otherwise, the value of P is that of v.

For a ~~non-type~~ constant *template-parameter* of reference or pointer type, or for each non-static data member of reference or pointer type in a ~~non-type~~ constant *template-parameter* of class type or subobject thereof, the reference or pointer value shall not refer to or be the address of (respectively):

- a temporary object[class.temporary],
- a string literal object[lex.string],
- the result of a `typeid` expression[expr.typeid],
- a predefined `__func__` variable[dcl.fct.def.general], or
- a subobject[intro.object] of one of the above.

� Template template arguments [temp.arg.template]

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

For a *type-template-parameter*, the name shall denote a class template or alias template. For a *variable-template-parameter*, the name shall denote a variable template. For a *concept-parameter*, the name shall denote a concept.

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*s,
- they are both constant template parameters,
- they are both *type-template-parameter*s,
- they are both *variable-template-parameter*s, or
- they are both *concept-parameter*s.

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

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

[*Editor's note:* See CWG2398]

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~~ constant *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).

[...]

� 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
- the template parameter values determined by their corresponding ~~non-type~~ constant template arguments[temp.arg.nontype] are template-argument-equivalent (see below), 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 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 **??**.

[*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*]

The associated constraints `C` of a declaration `D` are *subsumption eligible* unless `D` is a template declaration and `C` is dependent on a *concept-parameter* of `D`.

A declaration `D1` is *at least as constrained* as a declaration `D2` if

- `D1` and `D2` are both constrained declarations and `D1`'s associated constraints are subsumption eligible and subsume those of `D2`; or
- `D2` has no associated constraints.

A declaration `D1` is *more constrained* than another declaration `D2` when `D1` is at least as constrained as `D2`, and `D2` is not at least as constrained as `D1`. [*Example:*

```
template<typename T> concept C1 = requires(T t) { --t; };
template<typename T> concept C2 = C1<T> && requires(T t) { *t; };

template<C1 T> void f(T);       // #1
template<C2 T> void f(T);       // #2
template<typename T> void g(T); // #3
template<C1 T> void g(T);       // #4

f(0);                           // selects #1
f((int*)0);                     // selects #2
g(true);                        // selects #3 because C1<bool> is not satisfied
g(0);                           // selects #4
```

— *end example*]

[*Example:*

```
template <template <typename T> concept X, typename T>
struct S {};
template <typename T>
concept A = true;
template <typename T>
concept B = true && A<T>;
```

```
template <typename T>
concept C = true && B<T>;

template <template <typename T>  concept X, typename T>
int answer(S<X, T>) requires B<T> { return 42; } // #1
template <template <typename T> concept X, typename T>
int answer(S<X, T>) requires X<T> { return 43; } // #2

// error: the 3 following calls are ambiguous because #1 and #2 are not subsumption eligible
// (their associated constraints depend on a concept template parameter X)
answer(S<A, int>{});
answer(S<C, int>{});
answer(S<B, int>{});
```

— end example]

� 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.

  [*Editor's note:* We need wording to substitute C when C names a concept template parameter]

  [*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*]

[*Editor's note:* The wording in blue is added by P2963]

- For a *fold-operator* [expr.prim.fold] that is either `&&` or `||`
  - The normal form of an expression `( ... fold-operator E )` is the normal form of `( E fold-operator...)`.
  - The normal form of an expression `( E1 fold-operator ... fold-operator E2 )` is the the normal form of
    - `(E1 fold-operator...) fold-operator E2` if `E1` contains an unexpanded pack, or
    - `E1 fold-operator (E2 fold-operator...)` otherwise.

  [*Editor's note:* The wording in pink is added by P2963 and removed by this paper]

  - The normal form of `(E && ...)` is a fold expanded conjunction constraint [temp.constr.fold] whose constraint is the normal form of `E`.
  - The normal form of `(E || ...)` is a fold expanded disjunction constraint whose constraint is the normal form of `E`.

  [*Editor's note:* Add the following two bullets]

  - If `E` contains an unexpanded pack `P` naming a concept-name, the normal form of `(E fold-operator ...)` is the normal form of `( (`$(\mathrm{E}'_0$ *op* $\mathrm{E}'_1)$ *op* $\cdots$`)` *op* E'_{N-1} `)` where E'_i is formed by substituting the ith element of the corresponding template parameter of `P` in `E`.
  - Otherwise, the normal form of `(E && ...)` is a fold expanded conjunction constraint [temp.constr.fold] whose constraint is the normal form of `E` and the normal form of `(E || ...)` is a fold expanded disjunction constraint whose constraint is the normal form of `E`.
- 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*]

� 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 template parameter pack that is a pack expansion []:
  - 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.
- [...]

[...]

The instantiation of a pack expansion considers items $\mathtt{E}_1, \mathtt{E}_2, \ldots, \mathtt{E}_N$, where N is the number of elements in the pack expansion parameters. Each $\mathtt{E}_i$ is generated by instantiating the pattern and replacing each pack expansion parameter with its i^{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 an *id-expression* (for a ~~non-type~~ constant template parameter pack), a *typedef-name* (for a type template parameter pack ~~declared without `template`~~), or a *template-name* (for a ~~type~~ template template parameter pack ~~declared with `template`~~), designating the i^{th} corresponding type or value template argument;
- [...]

[...]

� Partial specialization [temp.spec.partial]

� General [temp.spec.partial.general]

[...]

A ~~non-type~~ constant argument is non-specialized if it is the name of a ~~non-type~~ constant parameter. All other ~~non-type~~ constant arguments are specialized.

Within the argument list of a partial specialization, the following restrictions apply:

- The type of a template parameter corresponding to a specialized ~~non-type~~ constant argument shall not be dependent on a parameter of the partial specialization.

[...]

� Function template overloading [temp.over.link]

[...]

When an expression that references a template parameter is used in the function parameter list or the return type in the declaration of a function template, the expression that references the template parameter is part of the signature of the function template. This is necessary to permit a declaration of a function template in one translation unit to be linked with another declaration of the function template in another translation unit and, conversely, to ensure that function templates that are intended to be distinct are not linked with one another. [*Example:*

```
template <int I, int J> A<I+J> f(A<I>, A<J>);   // #1
template <int K, int L> A<K+L> f(A<K>, A<L>);   // same as #1
template <int I, int J> A<I-J> f(A<I>, A<J>);   // different from #1
```

— *end example*] [*Note:* Most expressions that use template parameters use ~~non-type~~ constant template parameters, but it is possible for an expression to reference a type parameter. For example, a template type parameter can be used in the `sizeof` operator. — *end note*]

[...]

Two *template-head*s are *equivalent* if their *template-parameter-list*s have the same length, corresponding *template-parameter*s are equivalent and are both declared with *type-constraint* s that are equivalent if either *template-parameter* is declared with a *type-constraint*, and if either *template-head* has a *requires-clause*, they both have *requires-clause*s and the corresponding *constraint-expression*s are equivalent. Two *template-parameter*s are *equivalent* under the following conditions:

- they declare template parameters of the same kind,
- if either declares a template parameter pack, they both do,
- if they declare ~~non-type~~ constant template parameters, they have equivalent types ignoring the use of *type-constraint*s for placeholder types, and
- if they declare template template parameters, their template parameters are equivalent.

[...]

� 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~~ constant, 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~~ constant 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:

[...]

� General [temp.res.general]

A qualified or unqualified name is said to be in a *type-only context* if it is the terminal name of

- [...]
- a *decl-specifier* of the *decl-specifier-seq* of a
  - [...]
  - *parameter-declaration* in a *lambda-declarator* or *requirement-parameter-list*, unless that *parameter-declaration* appears in a default argument, or
  - *parameter-declaration* of a (~~non-type~~ constant) *template-parameter*.

� Locally declared names [temp.local]

Like normal (non-template) classes, class templates have an injected-class-name[class.pre]. The injected-class-name can be used as a *template-name* or a *type-name*. When it is used with a *template-argument-list*, as a *template-argument* for a template ~~*template-parameter*~~ *type-template-parameter*), or as the final identifier in the *elaborated-type-specifier* of a friend class template declaration, it is a *template-name* that refers to the class template itself. Otherwise, it is a *type-name* equivalent to the *template-name* followed by the template argument list[temp.decls.general,temp.arg.general] of the class template enclosed in <>.

When the injected-class-name of a class template specialization or partial specialization is used as a *type-name*, it is equivalent to the *template-name* followed by the *template-argument*s of the class template specialization or partial specialization enclosed in <>.

[...]

� Dependent types [temp.dep.type]

[...]

A template argument that is equivalent to a template parameter can be used in place of that template parameter in a reference to the current instantiation. For a template *type-parameter*, a template argument is equivalent to a template parameter if it denotes the same type. For a ~~non-type~~ constant template parameter, a template argument is equivalent to a template parameter if it is an *identifier* that names a variable that is equivalent to the template parameter.

[...]

A type is dependent if it is

- a template parameter,
- denoted by a dependent (qualified) name,
- a nested class or enumeration that is a direct member of a class that is the current instantiation,
- a cv-qualified type where the cv-unqualified type is dependent,
- a compound type constructed from any dependent type,
- an array type whose element type is dependent or whose bound (if any) is value-dependent,
- a function type whose parameters include one or more function parameter packs,
- a function type whose exception specification is value-dependent,
- denoted by a dependent placeholder type,
- denoted by a dependent placeholder for a deduced class type,
- denoted by a *simple-template-id* in which either the template name is a template parameter or any of the template arguments names a template template parameter or is a dependent type or an expression that is type-dependent or value-dependent or is a pack

expansion, [*Footnote:* This includes an injected-class-name[class.pre] of a class template used without a *template-argument-list*. — *end note*]

- a *pack-index-specifier*, or
- denoted by `decltype(`*expression*`)`, where *expression* is type-dependent[temp.dep.expr].

[*Note:* Because typedefs do not introduce new types, but instead simply refer to other types, a name that refers to a typedef that is a member of the current instantiation is dependent only if the type referred to is dependent. — *end note*]

� Type-dependent expressions [temp.dep.expr]

Except as described below, an expression is type-dependent if any subexpression is type-dependent.

`this` is type-dependent if the current class[expr.prim.this] is dependent[temp.dep.type].

An *id-expression* is type-dependent if it is a *template-id* that is not a concept-id and is dependent; or if its terminal name is

- associated by name lookup with one or more declarations declared with a dependent type,
- associated by name lookup with a ~~non-type~~ constant *template-parameter* declared with a type that contains a placeholder type[dcl.spec.auto],
- associated by name lookup with a variable declared with a type that contains a placeholder type[dcl.spec.auto] where the initializer is type-dependent,
- [...]

[...]

� Value-dependent expressions [temp.dep.constexpr]

Except as described below, an expression used in a context where a constant expression is required is value-dependent if any subexpression is value-dependent.

An *id-expression* is value-dependent if:

- it is a concept-id and any of its arguments are dependent,
- it is type-dependent,
- it is the name of a ~~non-type~~ constant template parameter,
- it names a static data member that is a dependent member of the current instantiation and is not initialized in a *member-declarator*,
- it names a static member function that is a dependent member of the current instantiation, or

- it names a potentially-constant variable [expr.const] that is initialized with an expression that is value-dependent.

[...]

� Dependent template arguments [temp.dep.temp]

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

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

Furthermore, a ~~non-type~~ constant *template-argument* is dependent if the corresponding ~~non-type~~ 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 *template-parameter* is dependent if it names a *template-parameter* or its terminal name is dependent.

[...]

� Template argument deduction [temp.deduct]

� General [temp.deduct.general]

[...]

[*Note:* Type deduction can fail for the following reasons:

- Attempting to instantiate a pack expansion containing multiple packs of differing lengths.
- Attempting to create an array with an element type that is `void`, a function type, or a reference type, or attempting to create an array with a size that is zero or negative. [*Example:*

```
template <class T> int f(T[5]);
int I = f<int>(0);
int j = f<void>(0);             // invalid array
```

  — *end example*]
- Attempting to use a type that is not a class or enumeration type in a qualified name. [*Example:*

```
template <class T> int f(typename T::B*);
int i = f<int>(0);
```

  — *end example*]
- Attempting to use a type in a *nested-name-specifier* of a *qualified-id* when that type does not contain the specified member, or

- the specified member is not a type where a type is required, or
- the specified member is not a template where a template is required, or
- the specified member is not a ~~non-type~~ constant where a ~~non-type~~ constant is required.

[*Example:*

```
template <int I> struct X { };
template <template <class T> class> struct Z { };
template <class T> void f(typename T::Y*) {}
template <class T> void g(X<T::N>*) {}
template <class T> void h(Z<T::TT>*) {}
struct A {};
struct B { int Y; };
struct C {
    typedef int N;
};
struct D {
    typedef int TT;
};

int main() {
    // Deduction fails in each of these cases:
    f<A>(0);          // A does not contain a member Y
    f<B>(0);          // The Y member of B is not a type
    g<C>(0);          // The N member of C is not a non-type constant
    h<D>(0);          // The TT member of D is not a template
}
```

— end example]

- Attempting to create a pointer to reference type.
- Attempting to create a reference to `void`.
- Attempting to create "pointer to member of `T`" when `T` is not a class type. [*Example:*

```
template <class T> int f(int T::*);
int i = f<int>(0);
```

— end example]

- Attempting to give an invalid type to a ~~non-type~~ constant template parameter. [*Example:*

```
template <class T, T> struct S {};
template <class T> int f(S<T, T{}>*);   // #1
class X {
    int m;
};
int i0 = f<X>(0);   // #1 uses a value of non-structural type X as a non-type
constant template argument
```

— *end example*]

— *end note*]

[...]

� Deducing template arguments from a function call [temp.deduct.call]

Template argument deduction is done by comparing each function template parameter type (call it `P`) that contains *template-parameter*s that participate in template argument deduction with the type of the corresponding argument of the call (call it `A`) as described below. If removing references and cv-qualifiers from `P` gives `std::initializer_list<P′>` or `P′[N]` for some `P′` and `N` and the argument is a non-empty initializer list [dcl.init.list], then deduction is performed instead for each element of the initializer list independently, taking `P′` as separate function template parameter types P'_i and the i^{th} initializer element as the corresponding argument. In the `P′[N]` case, if `N` is a ~~non-type~~ constant template parameter, `N` is deduced from the length of the initializer list. Otherwise, an initializer list argument causes the parameter to be considered a non-deduced context [temp.deduct.type].

[...]

� 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 ~~non-type~~ constant template parameter, or a template for a template parameter) that will make `P`, after substitution of the deduced values (call it the deduced `A`), compatible with `A`.

In some cases, the deduction is done using a single set of types `P` and `A`, in other cases, there will be a set of corresponding types `P` and `A`. Type deduction is done independently for each `P/A` pair, and the deduced template argument values are then combined. If type deduction cannot be done for any `P/A` pair, or if for any pair the deduction leads to more than one possible set of deduced values, or if different pairs yield different deduced values, or if any template argument remains neither deduced nor explicitly specified, template argument deduction fails. The type of a type parameter is only deduced from an array bound if it is not otherwise deduced.

A given type `P` can be composed from a number of other types, templates, and ~~non-type~~ constant template argument values:

- A function type includes the types of each of the function parameters, the return type, and its exception specification.
- A pointer-to-member type includes the type of the class object pointed to and the type of the member pointed to.

- A type that is a specialization of a class template (e.g., `A<int>`) includes the types, templates, and ~~non-type~~ constant template argument values referenced by the template argument list of the specialization.

- An array type includes the array element type and the value of the array bound.

In most cases, the types, templates, and ~~non-type~~ constant template argument values that are used to compose `P` participate in template argument deduction. That is, they may be used to determine the value of a template argument, and template argument deduction fails if the value so determined is not consistent with the values determined elsewhere. In certain contexts, however, the value does not participate in type deduction, but instead uses the values of template arguments that were either deduced elsewhere or explicitly specified. If a template parameter is used only in non-deduced contexts and is not explicitly specified, template argument deduction fails. [*Note:* Under **??**, if `P` contains no *template-parameter*s that appear in deduced contexts, no deduction is done, so `P` and `A` need not have the same form. — *end note*]

The non-deduced contexts are:

- The *nested-name-specifier* of a type that was specified using a *qualified-id*.

- A *pack-index-specifier* or a *pack-index-expression*.

- The *expression* of a *decltype-specifier*.

- A ~~non-type~~ constant template argument or an array bound in which a subexpression references a template parameter.

- [...]

[*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`, a template template argument denoting a variable template or a concept `VV`, or a template ~~non-type~~ constant argument `i` can be deduced if `P` and `A` have one of the following forms:

```
cv_opt T
T*
T&
T&&
T_opt [i_opt ]
T_opt (T_opt ) noexcept(i_opt )
T_opt T_opt ::*
TT_opt <T>
TT_opt <i>
TT_opt <TT>
TT_opt <>
TT_opt <VV>
```

[...]

Template arguments cannot be deduced from function arguments involving constructs other than the ones specified above.

When the value of the argument corresponding to a ~~non-type~~ constant template parameter `P` that is declared with a dependent type is deduced from an expression, the template parameters in the type of `P` are deduced from the type of the value.

[*Note:* Except for reference and pointer types, a major array bound is not part of a function parameter type and cannot be deduced from an argument:

```
template<int i> void f1(int a[10][i]);
template<int i> void f2(int a[i][20]);
template<int i> void f3(int (&a)[i][20]);

void g() {
    int v[10][20];
    f1(v);                          // OK, i deduced as 20
    f1<20>(v);                      // OK
    f2(v);                          // error: cannot deduce template-argument i
    f2<10>(v);                      // OK
    f3(v);                          // OK, i deduced as 10
}
```

— *end note*]

[...]

[*Note:* If, in the declaration of a function template with a ~~non-type~~ constant template parameter, the ~~non-type~~ constant template parameter is used in a subexpression in the function parameter list, the expression is a non-deduced context as specified above. [*Example:*

```
template <int i> class A { };
template <int i> void g(A<i+1>);
template <int i> void f(A<i>, A<i+1>);
void k() {
    A<1> a1;
    A<2> a2;
    g(a1);                          // error: deduction fails for expression i+1
    g<0>(a1);                       // OK
    f(a1, a2);                      // OK
}
```

— *end example*] — *end note*]

[*Editor's note:* Adjust the library wording as follow]

� Alias template `make_integer_sequence` [intseq.make]

```
template<class T, T N>
using make_integer_sequence = integer_sequence<T, see below>;
```

Mandates: `N` ≥ 0.

The alias template `make_integer_sequence` denotes a specialization of `integer_sequence` with `N` ~~non-type~~ constant template arguments. The type `make_integer_sequence<T, N>` is an alias for the type `integer_sequence<T, 0, 1, ..., N-1>`. [*Note:* `make_integer_sequence<int, 0>` is an alias for the type `integer_sequence<int>`. — *end note*]

� Random number engine class templates [rand.eng]

� General [rand.eng.general]

Descriptions are provided in **??** only for engine operations that are not described in **??** or for operations where there is additional semantic information. In particular, declarations for copy constructors, for copy assignment operators, for streaming operators, and for equality and inequality operators are not shown in the synopses.

Each template specified in **??** requires one or more relationships, involving the value(s) of its ~~non-type~~ constant template parameter(s), to hold. A program instantiating any of these templates is ill-formed if any such required relationship fails to hold.

� Random number engine adaptor class templates [rand.adapt]

� In general [rand.adapt.general]

Each template specified in this subclause **??** requires one or more relationships, involving the value(s) of its ~~non-type~~ constant template parameter(s), to hold. A program instantiating any of these templates is ill-formed if any such required relationship fails to hold.

[*Editor's note:* Finally, get rid of "non-type" in the compatibility annex]

� templates [diff.cpp14.temp]

Change: Allowance to deduce from the type of a ~~non-type~~ constant template argument.
Rationale: In combination with the ability to declare ~~non-type~~ constant template arguments with placeholder types, allows partial specializations to decompose from the type deduced for the ~~non-type~~ constant template argument.
Effect on original feature: Valid C++ 2014 code may fail to compile or produce different results in this revision of C++. For example:

```
template <int N> struct A;
template <typename T, T N> int foo(A<N> *) = delete;
void foo(void *);
void bar(A<0> *p) {
    foo(p);            // ill-formed; previously well-formed
}
```

Acknowledgments

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

We would like to thanks Daveed Vandevoorde for convince us to find a design that does not affect partial ordering.

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. P2963R0: Ordering of constraints involving fold expressions. https://wg21.link/p2963r0, 9 2023.

[5] 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.

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

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

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