Universal Template Parameters

Document #:	P1985R2
Date:	2022-09-15
Project:	Programming Language C++
Audience:	Evolution Working Group Incubator

Reply-to: Mateusz Pusz (Epam Systems)
<mateusz.pusz@gmail.com>
Gašper Ažman
<gasper.azman@gmail.com>
Bengt Gustafsson
<bengt.gustafsson@beamways.com>
Colin MacLean
<ColinMacLean@lbl.gov>
Corentin Jabot
<corentinjabot@gmail.com>

Contents

1 Introduction

This paper proposes a unified model for universal template parameters (UTPs) and dependent names, enabling more comprehensive and consistent template metaprogramming. Universal template parameters allow for a generic `apply` and other higher-order template metafunctions, including certain type traits.

2 Change Log

2.1 R2 -> R3

- Complete paper rewrite to make it more obvious what is proposed and what has been considered.
- Proposed examples have been expanded.
- Added universal aliases
- Merged variable template template parameters paper into this one
- Merged the concept template parameters paper into this one

— Respecified UTPs as dependent names and continued cleanup started with “down with `typename`”

2.2 R1 -> R2

Having found overwhelming support for the feature in EWGI, and a concern about contra- and co-variance of the `template auto` parameters, we include discussion of these topics in the paper, together with the decision and comparison tables that underpin the decision.

The decision was to explore both and put it to a vote.

Added list of questions for EWGI.

2.3 R0 -> R1

— Greatly expanded the number of examples based on feedback from the BSI panel.
— Clarified that we are proposing eager checking

3 Related work

— [P0634R3] [accepted] got rid of `typename` where it was obviously redundant; we hope to get rid of it in even more places (though not all).
— [P0945R0] [discarded] explored universal aliases; this proved unworkable, and the minutes of discussion informed this paper.
— [P2601R1] explores dropping empty `<>`. This might take up syntactic space that we need, but we haven't explored whether it does yet due to lack of time.
— [P0522R0] explores related partial ordering around partial template specialization; To our knowledge, this paper does not interact with the present proposal.
— [P2008R0] proposes variable-template template-parameters

4 Motivation and Examples

This paper unifies the model of template parameters with the model for dependent tokens (types, values, templates, and at some point hopefully concepts). This model is not uniform in C++23 because it lacks a way to treat all of the above uniformly, ironically denying the ability of generic code to treat itself generically.

Note: `template auto` is a placeholder syntax for such a parameter, albeit not a bad one; see the spelling discussion section later in the paper.

Consider the following examples this paper aims to enable.

4.1 Checking whether a type is a specialization of a given template

Main discussion of feature in [P2098R0] by Walter Brown and Bob Steagall.

When writing template libraries, it is useful to check whether a given type is a specialization of a given template. Such a trait is currently impossible to implement, as a template may potentially take an arbitrary mix of type and non-type template parameters. By introducing universal template parameters (UTPs), such a concept may be written easily, as follows:

```
// is_specialization_of
template <typename T, template <template auto...> typename Type>
constexpr bool is_specialization_of_v = false;

template <template auto... Params, template <template auto...> typename Type>
constexpr bool is_specialization_of_v<Type<Params...>, Type> = true;
```

```
template <typename T, template <template auto...> typename Type>
concept specialization_of = is_specialization_of_v<T, Type>;
```

This enables constraining to specific class templates:

```
// example from a units library
template <auto N, auto D>
struct ratio {
  static constexpr decltype(N) n = N;
  static constexpr decltype(D) d = D;
};

template <specialization_of<ratio> R1, specialization_of<ratio> R2>
using ratio_mul = simplify<ratio<R1::n * R2::n, R1::d * R2::d>>;

// std::array<class, size_t>
static_assert(specialization_of<std::array<int, 4>, std::array>);
// std::vector<class, class>, but with a default argument.
static_assert(specialization_of<std::vector<int>, std::vector);
```

4.2 `apply1`

A contrived-for-simplicity example which avoids the complexity of variadics:

```
template <template <template auto> typename F, template auto Arg>
using apply1 = F<Arg>;

template <typename X> struct takes_type {};
template <auto X>     struct takes_value {};
template <template <template auto...> typename X> struct takes_template {};

using r1 = apply1<takes_type, int>; // takes_type<int>
using r2 = apply1<takes_value, 3>;  // ok, takes_value<3>
using r3 = apply1<takes_template, takes_template>; // takes_template<takes_template>
```

4.3 Full `apply` metafunction

The non-contrived example is the `apply` metafunction, achievable like so:

```
template <template <template auto...> typename F, template auto... Args>
using apply = F<Args...>; // easy peasy!

// ok, r1 is std::array<int, 3>
using r1 = apply<std::array, int, 3>;
// ok, r2 is std::vector<int, std::pmr::allocator>
using r2 = apply<std::vector, int, std::pmr::allocator>;
```

In C++23, this is impossible to do; the various metaprogramming libraries get around that by boxing, or by only supporting type-taking metafunctions.

4.4 New Traits (also an example)

Universal template parameters allow new and useful kind-traits to be implemented. Note the partial specialization mechanism should work as one would expect.

Note: We define variable templates first because they compile faster.

```cpp
template <template auto> constexpr bool is_typename_v    = false;
template <typename T>    constexpr bool is_typename_v<T> = true;
template <template auto> constexpr bool is_value_v       = false;
template <auto V>        constexpr bool is_value_v<V>    = true;
template <template auto> constexpr bool is_template_v    = false;
template <template <template auto...> typename A>
constexpr bool is_template_v<A>                          = true;

// As we propose variable-template template-parameters
template <template auto>
constexpr bool is_variable_template_v                   = false;
template <template <template auto...> auto A>
constexpr bool is_variable_template_v<A>                = true;

// As we propose concept template-parameters
template <template auto> constexpr bool is_concept_v    = false;
template <template <template auto...> concept A>
constexpr bool is_concept_v<A>                          = true;

// The associated type for each trait:
template <template auto X> struct is_typename : std::bool_constant<is_typename_v<X>> {};
template <template auto X> struct is_value    : std::bool_constant<is_value_v<X>> {};
template <template auto X> struct is_template : std::bool_constant<is_template_v<X>> {};
template <template auto X> struct is_variable_template
    : std::bool_constant<is_variable_template_v<X>> {};
template <template auto X> struct is_concept
    : std::bool_constant<is_concept_v<X>> {};
```

4.5 A variable-to-type adaptor that exposes `::result`

A `box` is an important way of bridging to type-based metaprogramming, so we need to define it, also for the purposes of further examples.

Note: `box` is less necessary as we propose variable-template template-parameters.

```cpp
template <template auto> struct box; // impossible to define body

template <auto X>
struct box<X> { static constexpr decltype(X) result = X; };

template <typename X>
struct box<X> { using result = X; };

template <template <template auto...> typename X>
struct box<X> {
  template <template auto... Args>
  using result = X<Args...>;
};

// As we prropose variable-template template-parameters
template <template <template auto...> auto X>
struct box<X> {
  template <template auto... Args>
  static constexpr decltype(X<Args...>) result = X<Args...>;
```

```
};
```

4.6 `map_reduce` :: Total Example

We believe all the required features are used in the `map_reduce` example, where metafunction results are mapped into a `::result` member (we don't use the proposed variable template parameters here yet and instead use the box defined above).

```
template <template <template auto> typename Map,
          template <template auto...> typename Reduce,
          template auto... Args>
using map_reduce = Reduce<Map<Args>::result...>;
```

Note: notice that the above is a type-alias template. `box` allows us to return anything because it's a type.

Note: notice we expect the metafunction `Map` to return a `box`.

As an example usage of map_reduce let's count the number of types in the argument list:

```
template <int... xs> using sum = box<(0 + ... + xs)>;
template <template auto X> using boxed_is_typename = box<is_typename_v<X>>;
static_assert(2 == map_reduce<boxed_is_typename, sum, int, 1, long, std::vector>::result);
```

With variable-template template-parameters, we don't need to box `is_typename_v` and a better definition of `map_reduce` can be written. This version still needs box for the rsult of Reduce as it does not rely on *universal aliasese*.

```
template <template <template auto> typename Map,
          template <template auto...> template auto Reduce,
          template auto... Args>
using map_reduce_better = Reduce<Map<Args>...>;
```

We can use it more comfortably, as well:

```
static_assert(2 == map_reduce_better<is_typename_v, sum, int, 1, long, std::vector>::result);
```

Note: the kind of `Reduce` can be anything as it is enclosed in a box, which is a type.

5 Mechanism

This chapter describes the mechanics of universal template parameters (UTPs). Effectively, a UTP acts like a dependent name. This paper also cleans up dependent expressions so that they are more useful and consistent.

5.1 Specializing class templates on parameter kind

UTPs introduce similar generalizations as the `auto` universal NTTP did; in order to make it possible to pattern-match on the parameter, class templates need to be able to be specialized on the kind of parameter as well:

```
template <template auto> struct X;

template <typename T>
struct X<T> {
  // T is a type
  using type = T;
};

template <auto val>
struct X<val> {
```

```
    using type = decltype(val);
};

template <template <typename> typename F>
struct X<F> {
    // F is a unary metafunction
    template <typename T>
    using type = F<T>;
};

template<template auto U> user()
{
    using type = X<U>;
}
```

This basic mechanism allows the utterance of a UTP only in:

- *template-parameter-list* as the declaration of such a parameter.
- *template-argument-list* as the usage of such a parameter. It can bind only to a template parameter declared `template auto`.

This allows building enough traits to connect the new feature to the rest of the language with library facilities and rounds out template parameters as just another form of a compile-time parameter. However, it has the same kinds of limitations we saw before `if constexpr` was introduced: Function templates often had to rely on helper structs to do simple things.

5.2 Allowing UTPs in code

To make this feature useful, UTPs must be usable in code. Parsing of UTPs is made possible via disambiguation according to dependent name rules.

The disambiguation problem occurs often. Consider the following overloaded template functions and their interaction with dependent names:

```
template<typename T> int f() { return sizeof(T); } // f#1
template<auto V> int f() { return V; } // f#2

template<typename T> int caller()
{
    std::vector<T::name> x; // error in C++23, for little reason.
    return f<T::name>();    // "down with typename"++
    return f<T::name*>();    // Error: T::name* is syntactically not a constant expression
}
```

In C++23, the `f#2` is called if `T::name` is a value; but if a type, it is an error, instead of dispatching to `f#1`.

Note: No C++23 code is broken by this change as `caller` is only callable when `T::name` resolves to a value in C++23.

With UTPs this problem is more articulated:

```
template<template auto U> int caller2()
{
    return f<U>();
}
```

This paper proposes we keep *parsing* dependent expressions as-if they are values (C++23 rules) (unless disambiguated with `typename` and `template`), but we actually defer checking their kind until instantiation time.

Conceptually, dependent expressions can be thought of as universal template parameters whose disambiguation can be deferred when knowing what kind of entity they are is not needed in the immediate context.

5.2.1 Deferring kind checking to instantiation

Dependent expressions are still parsed as *constant-expression*, but kind-checks are always deferred to substitution time.

UTPs are just dependent expressions.

```
template<template auto U> int caller3()
{
    auto u = f<U>();        // Instantiates for values and types
    auto v = f<U*>();       // Error: U* is not syntactically a constant expression
    auto v = f<typename U*>();  // OK: U* is disambiguated
    using type = X<U>;      // Instantiates for all kinds.

    // Can fail instantiation if argument kind is wrong:
    auto v = U;             // Parses U as a value.
    using t = U*;           // Parses U as a type thanks to "down with typename".

    // Error during parse:
    U x;                    // Error: U is parsed as a value
    template<typename T> using tpl = U<T>;  // Error: U is parsed as a value.

    // Disambiguation needed when UTP is used outside of template argument:
    template<typename T> using tpl = template U<T>; // OK if U is bound to a class template
    typename U x;
    typename template U<int> i;  // Note: New double disambiguation. See below.
    auto vi = template U<int>;   // OK if U bound to a variable-template taking a typename
}
```

Dependent names also bind to universal template parameters without being disambiguated to value when they appear as template arguments, thanks to the deferred kind checking. This behavior enables utility `structs` to perform transformations on packs of template parameters involving mixes of types and NTTPs:

5.2.2 Automatic disambiguation when :: is applied

When a UTP is parsed it is presumed to be a value like any dependent name. However, we need to express that it is a type when it has a subordinate value or type.

```
template<template auto U> struct S {
    int x = U::value;
    typename U::type v;
};
```

This had worked fine if U was a typename template parameter, and we want it to work just as fine when U is a UTP. But a non-disambiguated UTP is not a type and we can't disambiguate it as a type by writing `int x  = typename U::value;` as this is the same syntax as when disambiguating `U::value` as a type.

To solve this we propose that :: automatically disambiguates its left hand side as a type for all dependent left hand sides and that typename disambiguation applies to the final nested name only.

This solves the problem and as a side effect also solves a similar problem we already have with nested dependent names.

A full discussion can be found in a separate chapter below.

5.2.3 Manual disambiguation to template

To disambiguate a UTP as a template requires using a prefix `template` keyword. This is novel but unsurprising. As a specialization of a dependent name disambiguated as template is treated as a value it must in itself be disambiguated as a type to be used as such:

```cpp
template<template auto U> struct TT {
    int x = template U<int, 3>;
    typename template U<int, 3> v;
};
```

The double disambiguation when declaring v is required as U is first disambiguated as a template and then the template specialization U<int, 3> also has to be disambiguated as a type.

6 Universal aliases

A universal alias is a name given to a dependent name or universal template parameter. A universal alias is in itself a dependent name, just like a UTP.

The grammar for a universal alias is simply:

universal-alias: **template auto** *identifier* = *template-argument* ;

The box example is now trivial, but also unnecessary.

```cpp
template<template auto U> struct box {
    template auto result = U;
};
```

With this definition the `map_reduce` example can be further refined:

````
template<template<template auto> template auto Map,
         template<template auto...> template auto Reduce,
         template auto... Args>
template auto map_reduce_best = Reduce<Map<Args>...>; ```

We can use it more comfortably, as well:

```cpp
template<int... xs> constexpr int sum = (0 + ... + xs);
static_assert(2 == map_reduce_best<is_typename_v, sum, int, 1, long, std::vector>);
````

Note: `Map` is now declared as a `template auto` metafunction, meaning that it could be a class template or as in the case of `sum` a variable template.

Note: Now we no longer need `sum` to be a box. `map_reduce_best` is a universal alias template. We still need to disambiguate these when used. In this example, however, as the result of `sum` is a value, we don't have to disambiguate inside the `static_assert`.

6.1 Properties of universal aliases

6.2 Universal aliases are purely compile-time entities

— if values, their initializer is `consteval`
— if types, that is trivially true.

In other words, a universal alias behaves like a UTP, but one that is introduced by a declaration as opposed to introduced as a template parameter.
````

The grammar production with *template-argument* as the initializer is correct, with one of its alternatives being *constant-expression*.

Note: in a previous iteration of [P0945R0], the initializer of value kind did not have to be `constexpr`, and the universal alias was just another name for a variable or function.

6.3 Universal aliases are always treated as dependent names

This is regardless of whether their initializer is dependent or not. This is to allow changing whether the initializer is dependent without impact to parsing.

7 Variable and Concept template template parameters.

Initially proposed in [P2008R0], variable template template parameters would allow passing variable templates, such as value type traits as template parameter.

Because concepts almost act like variable template of type `bool`, while offering more capabilities, we should also allow concept template template parameters.

This allows to express important ideas such as `range_of<std::integral>` or `tuple_like<std::regular>`

```
template <typename R, template<typename> concept C>
concept range_of =
            ranges::input_range<R>
            && C<remove_cvref_t<ranges::range_reference_t<R>>>;
```

The syntax for these new entities is straightforward (and leaves little room for invention).

```
template <auto N>                // Variable template parameter
template <template <> typename> // Type-template template-parameter
template <template <> auto>     // Variable-template template-parameter
template <template <> concept>  // Concept-template template-parameter
```

We can not have `template <concept>` as a concept is by definition a template.

Variable templates of a specific type (`template <template <> int>`) are also impossible, as the type of a template variable can vary across specializations, and there is no real way to restrict specializations from doing that.

In addition of the "sequence of entity satisfying some concept" use cases (like `range_of<integral>`), we can use this feature to better express complex, repetitive concepts. Barry Revzin offered a number of extremely compelling such use cases in this blog post. For example, we can describe more succinctly and expressively the many "indirect" iterator concepts used to support projections.

```
template<class F, class I, template <typename> concept direct>
concept indirect =
    indirectly_readable<I> &&
    copy_constructible<F> &&
    direct<F&, iter_value_t<I>&> &&
    direct<F&, iter_reference_t<I>> &&
    direct<F&, iter_common_reference_t<I>> &&
    common_reference_with<
      invoke_result_t<F&, iter_value_t<I>&>,
      invoke_result_t<F&, iter_reference_t<I>>>
;

template<class F, class I>
concept indirectly_unary_invocable =
```

```
    indirect<F, I, invocable>;

template<class F, class I>
concept indirectly_regular_unary_invocable =
    indirect<F, I, regular_invocable>;
```

Universal template parameters and variable/concept template parameters are orthogonal features, in that they could be standardized separately. However, when combined, these features are even more expressive. Notably, we can support both `range_of<int>` and `range_of<std::integral>`, by combining both features. IE:

```
template <typename R, template auto T> // Primary universal template
constexpr bool is_range_of = delete;

template <typename R, template <typename> concept C> // Specialization for concepts
constexpr bool is_range_of<R, C> = C<R>;

template <typename R, typename T> // Specialization for concrete types
constexpr bool is_range_of<R,T> = std::is_same_v<R, T>;

template <typename R, template auto T>
concept range_of = is_range_of<std::remove_cvref_t<std::ranges::range_reference_t<R>>, T>;

// We can now constrain a range to a specific type
static_assert(range_of<std::string, char>);
// Or a concept
static_assert(range_of<std::string, std::integral>);
```

In addition of the compelling synergy, it would be hard to ensure these features integrate well with one another if standardized separately, given the interactions. In particular, it means that:

— Universal parameters can be used as parameter to concept/variable template template parameters.

```
// A pack of variable template template parameter
// parametrized on universal template parameters
template <template<template auto...> auto... V>
```

`template auto` and `auto` are not ambiguous here, but it might be confusing for the reader.

A UTP can then be:

— A type
— A value
— A type template
— A variable template
— A concept

We need to ensure that the product of these interactions is coherent, especially for dependant expressions (Even if neither variable or concept template template parameters should not require new disambiguators, `template` is enough to disambiguate all types of template template parameters). The best way to ensure that coherence is to evolve both features at the same time, in the same paper.

8 Clarifying examples

8.1 Single parameter examples

```
template <int>                          struct takes_int {};
template <typename T>                   using  takes_type = T;
```

```
template <template auto>                   struct takes_anything {};
template <template <typename> typename F> struct takes_metafunc {};

template <template <template auto> typename F, template auto Arg>
struct fwd {
  using type = F<Arg>; // ok, passed to template auto parameter
}; // ok, correct definition

void f() {
  fwd<takes_int, 1>{};    // ok; type = takes_int<1>
  fwd<takes_int, int>{};  // error, takes_int<int> invalid
  fwd<takes_type, int>{}; // ok; type = takes_type<int>
  fwd<takes_anything, int>{}; // ok; type = takes_anything<int>
  fwd<takes_anything, 1>{};   // ok; type = takes_anything<1>
  fwd<takes_metafunc, takes_type>{}; // ok; type = takes_metafunc<takes_type>
  fwd<takes_metafunc, takes_int>{}; // error. (1)
}
```

(1): `takes_int` is not a metafunction on a *type*, so `takes_metafunc<takes_int>` is invalid (true as of *C++98*).

8.2 Variadic Examples

Consider the expansion of a non-homogeneous pack of universal template parameters. The result should not be surprising:

```
template <template auto X, template auto Y>
struct is_same : std::false_type {};
template <template auto V>
struct is_same<V, V> : std::true_type {};

template <template auto V, template auto ... Args>
struct count : std::integral_constant<
    size_t,
    (is_same<V, Args>::value + ...)> {};

// ok, ints = 2:
constexpr size_t ints = count<int, 1, 2, int, is_same, int>::value;
// ok, twos = 1:
constexpr size_t twos = count<2, 1, 2, int, is_same, int>::value;
```

Similarly a transformation of pack element kind can be easily implemented, such as unwrapping any integral_constant types into their enclosed value:

```
template<typename T>
struct unwrap
{
  using result = T;
};

template<typename T, T t>
struct unwrap<std::integral_constant<T, t>>
{
  static constexpr T result = t;
};
```

Using the above, the following is valid:

```cpp
template <template <template auto...> typename T, typename... Params>
using apply_unwrap = T<unwrap<Params>::result...>;

apply_unwrap<std::array, int, std::integral_constant<std::size_t, 5>> arr;
```

`unwrap<Params>::result...` forms a pack of UTPs with any integral_constant unwrapped to its underlying value.

8.3 Example of parsing ambiguity

UTPs must follow dependent name rules to avoid parsing ambiguities. Example courtesy of [P0945R0], and adapted:

```cpp
template <template auto A>
struct X {
  void f() { A * a; } // multiplication
};
```

If `A` were indeterminate, potentially being a type or a value, the expression could be parsed as either a declaration or a multiplication. By treating `A` as just a dependent name, this expression always parses as a multiplication. To treat A as a `typename` and a as a variable being declared, `A` has to be disambiguated, as any dependent name would.

Original example from [P0945R0]:

```cpp
template <typename T> struct X {
  using A = T::something; // P0945R0 proposed universal alias
  void f() { A * a; }
};
```

9 Example Applications

This feature is very much needed in very many places. This section lists examples of usage.

9.1 Enabling higher order metafunctions

This was the introductory example. Please refer to the [Proposed Solution].

Further example: `curry`:

```cpp
template <template <template auto...> typename F,
          template auto ... Args1>
struct curry {
  template <template auto... Args2>
  using func = F<Args1..., Args2...>;
};
```

9.2 Making dependent `static_assert(false)` work

Dependent static assert idea is described in [P1936R0] and [P1830R1]. In the former the author writes:

> Another parallel paper [P1830R1] that tries to solve this problem on the library level is submitted. Unfortunately, **it cannot fulfill all use-case since it is hard to impossible to support all combinations of template template-parameters in the dependent scope**.

The above papers are rendered superfluous with the introduction of this feature. Observe:

```
// stdlib
template <bool value, template auto... Args>
constexpr bool dependent_bool = value;
template <template auto... Args>
constexpr bool dependent_false = dependent_bool<false, Args...>;

// user code
template <template <typename> typename Arg>
struct my_struct {
  // no type template parameter available to make a dependent context
  static_assert(dependent_false<Arg>, "forbidden specialization.");
};
```

However, a language change, such as proposed by [P2593R0] would still be beneficial.

9.3 Universal alias as a library class.

See the initial `box<>` example.

While this paper does not try to relitigate Richard Smith's [P0945R0], it does provide a solution to aliasing anything as a library facility, without running into the problem that [P0945R0] ran into, even if EWG chooses to not allow the universal aliases described herein.

9.4 Bringing CTAD to `make_unique` et. al.

With the introduction of CTAD (constructor template argument deduction) a discrepancy was created which favors using plain new instead of `make_unique` as the latter needs the template arguments of a template class to be spelled out.

With UTPs we can add overloads to `make_unique` which make CTAD work in these situations:

```
auto a = std::tuple(1, true, 'a');       // ok
auto ap = new std::tuple(1, true, 'a');  // ok

// not ok in C++23.
auto up = std::make_unique<std::tuple>(1, true, 'a');

int arr[] = {1, 2, 3};

auto s = std::span(arr);                 // ok
auto sp = new std::span(arr);            // ok

// not implementable in C++23 as span has a NTTP.
auto sup = std::make_unique<std::span>(arr);
```

An overload to implement this would look something like:

```
template<template<template auto...> class C, typename... Ps>
auto make_unique(Ps&&... ps) {
    return unique_ptr<decltype(C(std::forward<Ps>(ps)...))>(new C(std::forward<Ps>(ps)...));
}
```

Note that the `decltype` is required as there is no deduction guide for plain pointers, this is however not a problem specific to the universal template overload.

10 Interactions with other language features

10.1 Impacts on function template overloading

The existence of UTPs requires backwards-compatible fixes to template overloading; there aren't two ways about it though, it behaves as a universal match that is the worst match in all cases.

```
template<template auto X> const char* kind_name() { return "type"; }
template<template<template auto...> X> const char* kind_name() { return "template"; }
template<auto X> const char* kind_name() { return "value"; }
```

10.2 Impact on class template specialization

UTPs can be used to implement what appears like overloading of class templates.

Class template specialization is finally unified with function template overloading with regards to its power - with a completely unconstrained base template (`template <template auto...> struct my_container;`), partial template specializations can span the entire universe of possible template parameter kinds and arities.

```
template <template auto...> struct my_container;

template <typename T> struct my_container<T> {
    my_container(T* data, size_t count);
    // A basic implementation
};

template <typename T, typename A> struct my_container<T, A> {
    my_container(T* data, size_t count);
    my_container(T* data, size_t count, const A& alloc);
    // An implementation using an allocator A
};

template <typename T, size_t SZ> struct my_container<T, SZ> {
    my_container(T* data, size_t count);
    // An implementation with an internal storage of SZ bytes
};

template <typename T> my_container(T*, size_t) -> my_container<T>;
template <typename T, typename A> my_container(T*, size_t, const A&) -> my_container<T, A>;
```

10.3 Impact on the partial specialization of NTTP templates

It is a common pattern in C++ to use SFINAE to constrain template parameter types:

```
template <typename T, typename=void>
struct A;

template <template <typename> typename T, typename U>
struct A<T<U>,std::enable_if_t<std::is_integral_v<U>>>
{
    // Implementation for templates with integral parameter types
};

template <template <typename> typename T, typename U>
struct A<T<U>,std::enable_if_t<std::is_floating_point_v<U>>>
{
```

```cpp
    // Implementation for templates with floating point parameter types
};

template <typename T>
struct X {};

A<X<int>> a; // Uses integral partial specialization
```

The loose matching behaviour of `template auto` allows a similar pattern to be used for NTTPs.

```cpp
template <typename T, typename=void>
struct B;

template <template <template auto> typename T, auto U>
struct B<T<U>,std::enable_if_t<std::is_integral_v<decltype(U)>>>
{
    // Implementation for templates with integral parameter types
};

template <template <template auto> typename T, auto U>
struct B<T<U>, std::enable_if_t<std::is_floating_point_v<decltype(U)>>>
{
    // Implementation for templates with floating point parameter types
};

template <int I>
struct Y {};

B<Y<5>> b; // Uses integral partial specialization
```

For reference, `auto` used with NTTPs behaves as follows:

```cpp
template <typename T, typename=void>
struct C;

template <template <auto> typename T, auto U>
struct C<T<U>,std::enable_if_t<std::is_integral_v<decltype(U)>>>
{};

template <int I>
struct Z1 {};

template <auto I>
struct Z2 {};

C<Z1<5>> c1; // Error: T does not match Z1
C<Z2<5>> c2; // Ok: NTTP can only be `auto`
```

10.4 Impacts on the specialization of variable templates

UTPs can be used to implement what appears like overloading of variable templates.

The problem of not being able to delete the base case then becomes more pressing than in C++23, as the designer might wish that selecting the base case trigger an error, when it is still selected when none of the specializations matches. This is solved by [P2041R0], which is assumed in the example below.

```cpp
// Metafunction to find a tuple element by a type predicate.
template <template <typename> typename Pred, size_t Pos, typename Tuple>
constexpr size_t tuple_find() {
    if constexpr (Pos == tuple_size_v<Tuple>())
        return npos;
    else if constexpr (Pred<remove_cvref_t<tuple_element_t<Pos, Tuple>>>::value)
        return Pos;
    else
        return tuple_find<Pred, Pos + 1, Tuple>();
}
template <template <typename> typename Pred, typename Tuple>
constexpr size_t tuple_find() { return tuple_find<Pred, 0, Tuple>(); }

// Helper to bind the first arguments of a provided template
template <template <template auto...> typename TPL, template auto... Bs> struct curry {
    template <template auto... Ts> using func = TPL<Bs..., Ts...>;
};

// Unimplemented base case.
template <template auto... Ps>
constexpr size_t tuple_find_v = delete;

template <template <typename> typename Pred, typename Tuple>
constexpr size_t tuple_find_v<Pred, Tuple> = tuple_find<Pred, Tuple>();

template <template <typename> typename Pred, size_t Pos, typename Tuple>
constexpr size_t tuple_find_v<Pred, Pos, Tuple> = tuple_find<Pred, Pos, Tuple>();

// Convenience specialization for use with binary predicate
template <template <typename, typename> typename Pred, typename M, typename Tuple>
constexpr size_t tuple_find_v<Pred, M, Tuple> = tuple_find_v<curry<Pred, M>::template func, Tuple>;

// Convenience specialization to match particular type.
template <typename T, typename Tuple>
constexpr size_t tuple_find_v<T, Tuple> = tuple_find_v<std::is_same, T, Tuple>;
```

This example contains a metafunction `tuple_find` to find a matching element in a tuple based only on its type. Unfortunately, it must be implemented as a `constexpr` function in C++23 if we want it to be usable with or without the start position (which would mimic the `std::find` function in the value domain).

With UTPs we can specialize a template variable `tuple_find_v` to regain the symmetry with current tuple oriented metafunctions such as `tuple_size_v` and `tuple_element_t`.

Given further overloads of the `constexpr` function `tuple_find` we could simplify the variable template definition to:

```cpp
template <template auto... Ps> constexpr size_t tuple_find_v = tuple_find<Ps...>();
```

This relies on the power of UTPs in another way, and has the same simplicity as the current variable templates and type aliases of the standard library type traits.

To take the consistency a step further `tuple_find` could be implemented as a class template with UTPs as shown in the previous example instead of as the function it must be in C++23.

11 Choice of syntax and keyword

The `template auto` syntax in this proposal is a placeholder. EWG needs to decide on a spelling.

This section is written to aid this process.

11.1 Drawbacks of `template auto`

The initially suggested spelling `template auto` is close to existing syntax to explicit function template specialization.

```
template <typename T> auto f(T x) { return x; }

template auto f<float>(float y);
```

11.2 Choices

We can do any of the following:

- — repurpose an existing keyword (like `register`, `inline` or `constexpr`)
- — use a combination of keywords (see table)
- — make a new keyword
- — put a `?` after a keyword

A distinct feature like UTPs would generally require a new keyword.

Introducing a new keyword may have backward compatibility difficulties. Nonetheless, C++23 introduced 8 new keywords, including the relatively common words `concept` and `requires`.

This paper proposes a specialist feature, which led us to try to use a compound keyword instead of introducing a new one. *This is novel.*

As shown by the previous section subtle use cases of `template auto` went undetected for years, and even 1.5 implementations were made without detecting it as the grammar productions where the two uses are valid are distinct.

In conclusion, introducing a new keyword has a risk of breaking old code, while using two keywords in combination to mean a distinct thing is unprecedented in C++ and carries some risk of its own.

Therefore we suggest an initial poll to make the selection whether to search for a usable and understandable combination of current keywords, or to select a new keyword with suitably obscure spelling to avoid too much code breakage.

To aid in this decision we have brainstormed a set of reasonable token combinations and a set of possible new keywords. None of the lists is exhaustive and additional suggestions are welcomed by the authors.

Keyword combinations	New Keywords
`template auto`	`any_name`
`auto template`	`any_kind`
`auto typename`	`auto_kind`
`typename auto`	`unknown_kind`
`template?`	`universal_name`
	`universal_kind`
	`univ_name`
	`univ_kind`
	`indeterminate`
	`indet_name`
	`dependent_name`

Keyword combinations	New Keywords
	`any_kind_name`

11.2.1 Keyword discussion

If a new keyword is to be recommended, the list above contains two word combinations except for one long and complicated word that could be useable in isolation.

Apart from the collision risk it is of course important that the keyword describes the semantics of the feature as closely as possible. Succinctly the feature semantics can be stated as:

A template parameter that can be bound to any kind of template argument.

This sentence conveys a few facts:

- It is a template parameter
- It can be bound to any kind
- It can have a name (as can all template parameters)

Of these facts we can disregard the first as it is conveyed by the context where the keyword is used, and thus need not be indicated by the keyword. The second fact is the central idea and the word *any* is what conveys this information. The third fact is also given by the context but the spelling `typename` still includes `name` which is to be noted.

This said the list of suggested names include mostly synonyms of `any` in a wide sense: any, auto, unknown, universal, indeterminate, dependent. The word `dependent` was included as it exactly describes how a UTP works to a C++ expert. We also included somewhat obscure abbreviations of universal and dependent which could help reduce the amount of code breakage.

For the last half of the keyword we only came up with name or kind. We think that name is probably best although it may viewed as redundant. One reason is that `typename` exists and another is that kind seems to refer to a reflection of a UTP, which indicates which *kind* it was bound to. When reflections on UTPs the kind it was bound to for a certain instantiation will be of interest.

11.2.2 To underscore or not to underscore

While the keyword parts are written separately in the table above the intent is to either write the words together or with an interposed underscore.

Checking the current C++ keywords list the keywords consisting of two English words sometimes have an underscore, sometimes not, in a fairly even mix. The authors can't see a pattern which could direct whether to use an underscore in a new keyword. We tested hypotheses that old keywords had no underscore and that longer words requires an underscore, but could not find any correlation.

There are 15 keywords without underscore: `alignas`, `alignof`, `bitand`, `bitor`, `consteval`, `constexpr`, `constinit`, `decltype`, `inline`, `noexcept`, `nullptr`, `sizeof`, `typedef`, `typeid`, `typename`.

There are 12 keywords with underscore: `and_eq`, `co_await`, `co_return`, `co_yield`, `const_cast`, `dynamic_cast`, `not_eq`, `or_eq`, `reinterpret_cast`, `static_assert`, `static_cast`, `thread_local`, `xor_eq`.

11.2.3 Second and third polls

A second poll is either to select a suitable combination of tokens or to select the preferred new keyword spelling recommended from LWG(I?).

If a keyword is selected in the first poll a third poll regarding underscore or not is warranted. It is the opinion of the author that more two word lower case identifiers in C++ code have an underscore than not, therefore making a keyword without underscore less likely to break code.

11.3 Other Considered Syntaxes

In addition to the syntax presented in the paper, we have considered the following syntax options:

11.3.1 `.` and `...` instead of `template auto` and `template auto ...`

```
template <template <...> typename F, . x, . y, . z>
using apply3 = F<x, y, z>;
```

The reason we discarded this one is that it is very terse for something that should not be commonly used, and as such uses up valuable real estate.

12 Integration with reflection

The authors expect that code using reflection will have a need for this facility. Spliced entities also have to follow the same set of disambiguation rules, as they are fundamentally dependent entities.

The examples in this section are pending discussion with reflection authors.

An important contrast would be to see the implementations of `is_specialization_of_v` by this paper and the reflection one.

13 Digging into the nested disambiguation jungle

The situation for disambiguating nested dependent names in C++23 is as follows. Consider this example code:

```
template<typename T> struct S1 {
    typename T::type1::type2 v1;  // OK
    int x1 = T::type1::value1;    // Error(C++23), OK(proposed)
};
```

Only types (and namespaces and enumerations) can have named members accessible with operator `::`.

One would imagine, therefore, that `T::type1` could be automatically disambiguated as a type so that `::value1` could be applied.

This is not currently the case: The `typename` in the declaration of `v1` disambiguates *both* `type1` and `type2` to be `typename`s. The error in the initializer of `x1` is due to `type1` not being treated as a type.

We are left in an unfortunate situation where `value1` can't be used as a value. A workaround is to extract `T::type1` into a type alias, and apply the `::value1` portion to the alias. Then, `value1` can be treated as a value by *not* prefixing `typename`:

```
template<typename T> struct S2 {  // OK(C++23)
    using type1 = T::type1;       // Intermediate name
    int x2 = type1::value1;       // Access dependent name value1 as a value.
};
```

If we take a look at dependent names which are subordinate to a dependent name disambiguated as a template we have a similar situation, but not exactly equivalent:

```
template<typename T> struct S3 {
    typename T::template t_tpl<int, 2> v2; // typename required (C++23)
    typename T::template t_pl<int, 2>::type2 v3;
    int x3 = T::template t_tpl<int, 2>::value1;  // OK(C++23), but WHY?
};
```

For `v2` the leading `typename` disambiguates `t_tpl<int, 2>` as being a type, as it would otherwise be parsed as a variable-template specialization.

For `v3`, `typename` disambiguates *both* `t_tpl<int, 2>` and its member `type2` as types, consistent with the `v1` declaration.

That `x3` can be initialized while `x1` can't (in C++23) is curious; clearly, though, `t_tpl<int, 2>` is parsed as a `typename`, which allows `::value1` to be applied. This is not consistent with the non-template case `x1`.

We can also reverse the order between the template and the `typename`, so that `T` must contain a type containing a template. This template can be a class template or variable template as above.

```
template<typename T> struct S4 {
    typename T::type1::template t_tpl<int, 3> v4; // OK
    int x4 = T::type1::template v_tpl<int, 3>;  // OK(C++23), but WHY?
};
```

The leading `typename` again disambiguates both `type1` and `t_tpl<int, 3>` as types when declaring `v4`, while the declaration of `x4`'s `v_tpl<int, 3>` is inconsistent with `x1`'s `T::type1::value1`.

Clearly, if we want UTPs to work like dependent names, we don't want dependent names to behave this strangely. Below all the cases above are listed inside one class template taking a universal template parameter `U`.

```
template<template auto U> struct S5 { // all proposed
    // Please note every line is independent :)

    int x5 = U;     // U is treated as a value
    typename U v5; // U can be disambiguated to typename.

    // U disambiguated as variable template.
    int x5 = template U<int, 3>;
    // U disambiguated as a class template
    typename template U<int, 3> v5;

    // U can be a type containing a value or subtype
    int x6 = U::value1;
    typename U::type2 v6;

    // U disambiguated as a template,
    // U<int, 3> followed by :: parsed as a type
    int x7 = template U<int, 3>::value1;
    typename template U<int, 3>::type2 v7;

    // U followed by :: parsed as a type
    // v_tpl<int, 3> disambiguated as a template variable
    int x8 = U::template v_tpl<int, 3>;
    // t_tpl<int, 3> disambiguated as a class template by leading typename
    typename U::template t_tpl<int, 3> v8;
};
```

In C++23 `int x1 = T::type1::value1;` is invalid, whereas this paper defines it to be valid, as long as `T::type1` ends up being a type at instantiation time, and is parsed as a type at parse time.

To rectify the `x1` situation, we change the rule for `typename` disambiguation to only affect the last of a set of nested dependent names, while a `::` automatically marks a dependent name to its left as a type. This does not change the meaning of `v1`'s declaration, but makes `x1`'s initializer a value as `type1` is disambiguated by the trailing `::` and `value1` is a value as there is no leading `typename`. The `x3` and `x4` initializers behave the same as in C++23, but with a clearer rationale.

In the template cases `x5`, `x7`, `v5` and `v7`, we must explicitly specify `template` to correctly parse `<` as the *template-head* introducer. This is novel (`template` could only appear after `::` in C++23) but utterly unsurprising.

```
int x5 = T::template name<int, 3>
typename T::template name<int, 3> v5;

int x7 = T::template name<int, 3>::value1;
typename T::template name<int, 3>::type2 v7;
```

Note: the proposed syntax is close to explicit template instantiation.

```
template SomeClassTemplate<int, 3>;
```

This is an explicit instantiation of a class template `SomeClassTemplate`.

Variable declarations are allowed in namespace scope, which in the presence of universal alias templates comes close to the proposed syntax:

```
template <typename auto ... >
template auto U = int; // for the purposes of this example

typename template U<int, 3> x;
```

14 Compendium: Former design questions

14.1 Discussion on co- and contra-variance

14.2 A survey of covariance and contravariance in C++23

14.2.1 Explanatory example

For the people who, like most of us, don't do type theory every day, let's start with an explanation of contravariance and covariance.

It's basically about match-into-wider and match-into-narrower.

Consider the two concepts:

```
template <typename T> concept A = true;
// B subsumes A
template <typename T> concept B = A<T> && true;
```

Covariance

```
auto returns_a() -> A auto;
auto returns_b() -> B auto;

// Return types generally should behave covaria
auto f() {
    // OK, requirement less constrained than re
    A auto x = returns_b();
    // Error, requirement stricter than returne
    B auto y = returns_a();

}
```

Contravariance

```
// Parameter matching should behave contravariantly

template <template <B> typename f>
using puts_b = void;
template <template <A> typename f>
using puts_a = void;
template <B> using takes_b = void;
template <A> using takes_a = void;
using x = puts_b<takes_a>; // ok
// Error, constraint mismatch (gcc)
// clang accepts (in error)
using w = puts_a<takes_b>;
```

14.2.2 Discussion

We are used to the covariant case, but the usage of contravariant cases is not as common. The issue with `puts_a<takes_b>` is that `puts_a` requires a metafunction with a *wider interface* than one that just accepts `B`s.

We have seen that concept-constrained template parameters behave correctly - covariantly on returns, contravariantly on parameters; but do other template parts as well?

Let's just replace `A` with `auto` and `B` with `int`:

```
template <template <int> typename f> using puts_int = void;
template <template <auto> typename f> using puts_auto = void;
template <int> using takes_int = void;
template <auto> using takes_auto = void;
using x = puts_int<takes_auto>; // OK
using w = puts_auto<takes_int>; // Error, but MSVC, GCC and clang all accept
```

This behaviour is specified in [P0552R0].

Function pointers do not convert in either co- or contravariant ways:

```
struct X {}; struct Y : X {};
using f_of_x = void(*)(X&);
using f_of_y = void(*)(Y&);
// Error, no conversions between function pointers
f_of_y fy = static_cast<f_of_x>(nullptr);

using f_to_x = X&(*)();
using f_to_y = Y&(*)();
// Error, no conversions between function pointers
f_to_x xf = static_cast<f_to_y>(nullptr);
```

Virtual functions can have covariant return types, though:

```
struct ZZ {};
struct Z : ZZ {
   virtual auto f() -> Z&;
   virtual void g(Z&);
};
struct W : Z {
   auto f() -> W& override; // OK, covariant return type
   // Error, no contravariant parameter types
   void g(ZZ&) override;
};
```

How about parameter packs?

```
template <template <typename> typename f> using puts_one = void;
template <template <typename...> typename f> using puts_var = void;
template <typename> using takes_one = void;
template <typename...> using takes_var = void;
using x = puts_one<takes_var>; // OK in 17 and 20, Error in 14
using w = puts_var<takes_one>; // OK, compiles
```

Turns out parameter packs behave **both** co- and contra-variantly (what we call loosely) - while inconsistent with the above examples, it is the authors' opinion the choice was correct.

14.3 Why settle on loose matching for `template auto`

Let's say we did the strict thing and made `template auto` behave contravariantly, like concepts.

```
template <template <auto> typename f> using puts_value = void;
template <template <template auto> typename f> using puts_any = void;
template <auto> using takes_value = void;
template <template auto> using takes_any = void;
using x = puts_value<takes_any>; // OK
using w = puts_any<takes_value>; // Error because contravariant.
```

But then, how to write `apply`? Let's do it for a single argument to avoid complications with a loose `...`:

```
template <template <template auto T> typename f, template auto arg>
using apply1 = f<arg>;
```

The above is correct, but *useless* - it requires `f`s signature to be `template <template auto T>`. What we want to express is that `f` is any kind of unary template metafunction, so we can pass in something like `template <int x> using int_constant = std::integral_constant<int, x>;`.

As a thought experiment, let's call the covariant version of `template auto` (with the meaning "deduce this from the argument") `__`:

```
template <template <__ T> typename f, template auto arg>
using apply1 = f<arg>;
```

This is what we want to express (and check at instantiation time), but now the `arg`s constraint is spelled differently from `f`'s constraint, and that might be very, very difficult to teach.

It also requires us to reserve an additional combination of tokens. Contrast what happens if we just made `template auto` behave covariantly (the way `__` behaves above) if used in that position. What do we lose?

At first glance, we lose the ability to define the contravariant meaning of `template auto` - a template parameter that *can* bind to anything. But can we get that back?

Recall that concepts behave contravariantly. Consider this one:

```
namespace std {
  template <template auto arg>
  concept anything = true;
};
```

We could say this:

```
template <template <std::anything ARG> typename takes_anything,
          template auto arg>
using apply1 = takes_anything<arg>;
```

This would behave *contravariantly*! While `template auto ARG` means "deduce", a metafunction taking a concept (which behaves contravariantly!) that *anything* satisfies *has* to be a metafunction taking `template auto` (or `std::anything`).

In light of this, the paper authors have come to the conclusion that trying to make `template auto` behave contravariantly is all downside and little upside. The example to follow with `template auto` should be the behavior of `...` (deduction behavior, both co- and contravariant).

This allows for usage to look the same as declaration, and *like* bind to *like*. We anticipate the feature being much easier to teach this way, and in the rare cases when someone really needs the contravariant behavior, they will know to use a concept. It is also consistent with the rest of the non-concept template language.

This paper takes the stance that mathematically correct behaviour can be obtained by defining a concept, upon which subsumption and concept matching rules will obey the correct norms. Defining a concept also gives a library name and frees up a keyword.

The language facility, however, should stay loosely matched, principally because packs are already loosely matched, and because loose matching is not something the programmer is able to reclaim, whereas strict matching is recoverable by defining a concept.

15 Acknowledgements

Special thanks and recognition goes to Epam Systems for supporting Mateusz's membership in the ISO C++ Committee and the production of this proposal.

Gašper would likewise like to thank his employer, Citadel Securities Europe, LLC, for supporting his attendance in committee meetings.

Colin MacLean would also like to thank his employer, Lawrence Berkeley National Laboratory, for supporting his ISO C++ Committee efforts.

Bengt Gustafsson would like to thank his employer, ContextVision AB, for supporting his attendance in committee meetings.

Corentin would like to thank Bloomberg for supporting this work.

A big thanks also to the members of the BSI C++ panel for their review and commentary.

Thanks also to James Touton for walking Gašper through the covariance/contravariance design space.

16 References

[P0522R0] James Touton, Hubert Tong. 2016-11-11. DR: Matching of template template-arguments excludes compatible templates.
https://wg21.link/p0522r0

[P0552R0] Walter E. Brown. 2017-02-01. enable_if vs. requires.
https://wg21.link/p0552r0

[P0634R3] Nina Ranns, Daveed Vandevoorde. 2018-03-14. Down with typename!
https://wg21.link/p0634r3

[P0945R0] Richard Smith. 2018-02-10. Generalizing alias declarations.
https://wg21.link/p0945r0

[P1830R1] Ruslan Arutyunyan. 2019-10-07. std::dependent_false.
https://wg21.link/p1830r1

[P1936R0] Ruslan Arutyunyan. 2019-10-07. Dependent Static Assertion.
https://wg21.link/p1936r0

[P2008R0] Mateusz Pusz. 2020-01-10. Enable variable template template parameters.
https://wg21.link/p2008r0

[P2041R0] David Stone. 2020-01-11. Deleting variable templates.
https://wg21.link/p2041r0

[P2098R0] Walter E Brown, Bob Steagall. 2020-02-17. Proposing std::is_specialization_of.
https://wg21.link/p2098r0

[P2593R0] Barry Revzin. 2022-05-21. Allowing static_assert(false).
https://wg21.link/p2593r0

[P2601R1] Justin Cooke. 2022-07-16. Make redundant empty angle brackets optional.
https://wg21.link/p2601r1