Universal Template Parameters

Document #:	P1985R2
Date:	2022-09-14
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>

Contents

1 Introduction

We propose a universal template parameter (UTP) and associated cleanups to make it useful for template metaprogramming. This will 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 centralised and expanded.

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; this paper to author's knowledge does not interact with that paper.
— [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. When our templates mix types and NTTPs, this trait is currently impossible to write in general. However, with a universal template parameter (UTP), we can write a concept for that 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, class>
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!

using r1 = apply<std::array, int, 3>; // ok, 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

We should be able to define new kind-traits based on this feature. Note the partial specialization mechanism should work as one would expect.

Note: We are intending to propose these in a later revision.

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

```
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;

// if we gain 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;

// if we gain 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;

// and their associated types:
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: we are not proposing `box` for the standard, but a paper probably should.

Note: `box` would be less necessary if we had variable-template template-parameters.

```
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...>;
};

// if we get 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...>;
};

// if we get concept template-parameters
template <template <template auto...> concept X>
struct box<X> {
  template <template auto... Args>
  concept result = X<Args...>;
};
```

4.6 `map_reduce` :: Total Example

We believe all the required features are used in the `map_reduce` example, where we have to map metafunction results into a `::result` member (we don't have variable template parameters yet).

```
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`.

Let's count the number of types in the argument list, for instance:

```
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 we can define `map_reduce` slightly better:

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

and 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: alas, we still need `sum` to be a box. We won't ever have universal aliases, but we *can* have universal dependent expressions.

Note: the kind of `Reduce`.

5 Mechanism

This chapter describes how universal template parameters (UTPs) work. Effectively, a UTP acts like a dependent name. This paper cleans up dependent expressions so that they are more useful everywhere.

5.1 Specializing class templates on parameter kind

UTPs introduce a 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 = template X<U>;  // Regardless of U's kind X <U> is a type.
}
```

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 kind 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. To make parsing possible we have to disambiguate UTPs according to dependent name rules.

We have a version of this problem already with overloaded template functions and 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++20, 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++20, 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`.

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++20 rules) (unless disambiguated with `typename` and `template`), but we actually defer the kind check until instantiation time.

5.2.1 Deferring kind matching 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 useful outside of template argument:
    template<typename T> using tpl = template U<T>; // ok if U a template
    typename U x;
    typename template U<int> i;  // Note: New double disambiguation
    auto vi = template U<int>;   // OK if U a variable-template taking a typename
}
```

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.

The promotion of dependent names to universal template parameters also applies to template argument packs and enables utility structs to perform transformations on packs of template parameters involving mixes of types and NTTPs:

```
template<template auto T>
struct unwrap;

template<typename T>
struct unwrap<T>
{
  tempalate auto result = T;
};

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

Using the above, this is valid:

```
template <template auto ... Xs>
constexpr auto sum_unwrapped = (... + unwrap<Xs>::result);
static_assert(sum_unwrapped<1, std::integral_constant<short, 2s>> == 3);
```

6 Universal aliases

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

This is an optional part of the paper, but it just fell out of the grammar when implementing in clang, and it obviates the need for a `box<X>`.

The grammar for a universal alias is simply:

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

The box example is now trivial, but also unnecessary.

```
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>...>;
```

and we can use it more comfortably as well:

```
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 also 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`.

7 Automatic disambiguation when :: is applied

To ensure that UTPs work exactly as dependent names we need to clean up how nested dependent names work. This can be done without breaking backwards compatibility, it just makes a few more cases valid, by generalizing a rule.

The lengthy description below sums up to this TL:DR style conclusion:

Let `::` automatically disambiguate its left hand side as a type, and let typename disambiguate the last dependent name as a type.

This is enough to make everything work consistently and unsurprisingly for both dependent names and UTPs.

7.1 Digging into the nested disambiguation jungle

Today there we have a rather weird situation when it comes to disambiguation of nested dependent names, i.e. when a dependent name contains another dependent name. As only types can have dependent names in them you would imagine that the `ftype` dependent name in the example below would automatically be disambiguated as a type so that ::svalue could be applied. This is not the case:

```
template<typename T> struct S {
    typename T::ftype::stype v1;      // OK
    int x1 = T::ftype::svalue;        // Error
};
```

The typename in the declaration of `v1` disambiguates *both* ftype and stype to type. The error in the x initializer is due to ftype not being disambiguated to type. This leaves us with the unfortunate situation that svalue can't be used as a value. A workaround is to extract `T::ftype` into a type alias. Then svalue can be used as a value by *not* disambiguating it:

```
template<typename T> struct S {
    using FType = T::ftype;           // Intermediate name
    int x2 = FType::svalue;           // Access dependent name svalue 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 S {
    typename T::template tpl<int, 2> v2;
    typename T::template tpl<int, 2>::stype v3;
    int x3 = T::template tpl<int, 2>::svalue;  // Works, but why?
};
```

For v2 the leading typename disambiguates the template instance as being a type. This is required as the template instance would otherwise be assumed to be an instance of a varable template.

For v3 typename disambiguates *both* the type-instance and its member stype as types, consistent with the v1 declaration.

How x3 can be initialized when x1 can't is hard to understand, maybe the reasons can be traced back to a time when variable templates were not a thing, and thus the tpl instance must be a type, which allows `::svalue` 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 S {
    typename T::ftype::template tpl<int, 3> v4;
    int x4 = T::ftype::template tpl<int, 3>;  // Works, but why?
};
```

Just as for the opposite order of template and type this works, the leading typename again disambiguates both ftype and the template instance as types when declaring v4, while the x4 initialization remains inconsistent with x1.

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 S {
    int x = U;                        // U is treated as a value as long as it is not disambiguated
    typename U v5;                    // U can be disambiguated to typename.
    int x5 = template U<int, 3>;      // U can be disambiguated to template. Its instance defaults to a val
```

```
    typename template U<int, 3> v5; // The instance can be re-disambiguated to a type.

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

    // U can be disambiguated as a template. The template instance is automatically disambiguated
    // to a type if followed by ::
    int x7 = template U<int, 3>::svalue;
    typename template U<int, 3>::stype v7;

    // Finally U can be disambiguated as a type containing a template, which is
    done by the trailing :: preceeding the template keyword.
    int x8 = U::template tpl<int, 3>;
    typename U::template tpl<int, 3> v8;
};
```

Claiming that `U::svalue` is a value makes UTPs work "better" than nested dependent names do today, as x1 above does not compile. This breaks the consistency.

To rectify this we should change the rule for typename disambiguation to only affect the last of a set of nested dependent names, while a `::` always automatically disambiguates a dependent name to the left as a type. This does not change the meaning of the `v1` declaration above but makes the `x1` initializer a value as ftype is disambiguated by the trailing `::`and svalue is a value as there is no leading `typename`. The x3 and x4 initializers behave the same as today, but with a more understandable rationale.

TODO: This section needs to be written the other way around: How the T::name can be replaced by U, how this can't be done textually and then that this gives us the novel *leading* template disambiguator, finally dismissing the potential parse conflict:

Everything is then consistent between dependent names and universal template parameters, if you replace the U with T::name in the last example above you can easily see this.

For the template cases x5, x7, v5 and v7 this can't be done just as a textual replacement which is a bit worrying. Instead the template disambiguator appears between T:: and name.

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

int x7 = T::template name<int, 3>::svalue; typename T::template name<int, 3>::stype v7;

This is especially true for the value case where, if the value is just mentioned as a (meaningless) expression statement:

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

This looks like an explicit instantiation of a class template U. I think this is not a real problem as expression statements can't be in namespace scope, while explicit template instantiations *must* be in namespace scope.

However, variable declarations are allowed in namespace scope so this would be scarily similar to the above explicit template instantiation:

template U<int, 3> x;

However, this would require that U is a universal alias in namespace scope, which we can easily forbid, as there are no dependent names in namespace scope that it can alias. Note that this is true only for non-template universal aliases, the templated variety is very useful in namespace scope, as shown by the map_reduce_best example above.

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_int>{}; // ok; type = takes_metafunc<takes_int>
  fwd<takes_metafunc, takes_metafunc>{}; // error (1)
}
```

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

8.2 Pack Expansion Example

It is interesting to think about what happens if one expands a non-homogeneous pack of these. 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;
```

8.3 Example of parsing ambiguity (if late check)

The reason for eager checking is that not doing that could introduce parsing ambiguities. Example courtesy of [P0945R0], and adapted:

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

The issue is that we do not know how to parse `f()`, since `A * a;` is either a declaration or a multiplication.

If we treated `A` as just a dependent name, this always parses as a multiplication. To treat A as a typename and a as a variable being declared the A has to be disambiguated as any dependent name.

Original example from [P0945R0]:

```
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`:

```
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.");
};
```

9.3 Universal alias as a library class.

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.

Observe:

```cpp
template <template auto Arg>
struct alias /* undefined on purpose */;

template <typename T>
struct alias<T> { using value = T; };

template <auto V>
struct alias<V> { auto value = V; };

template <template <template auto...> typename Arg>
struct alias<Arg> {
  template <template auto... Args> using value = Arg<Args...>;
};
```

We can then use alias when we *know* what it holds:

```cpp
template <template auto Arg>
struct Z {
  using type = alias<Arg>;
  // existing rules work
  using value = typename type::value; // dependent
}; // ok, parses

Z<int> z1; // ok, decltype(z1)::value = int
Z<1> z; // error, alias<1>::value is not a type
```

9.4 Impacts on the function template overloading

Universal template arguments impact template overloading:

```cpp
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"; }
```

Function templates taking a universal template argument is considered a worse match than the one taking a specific template parameter kind.

9.5 Bringing CTAD to make_unique et. al.

With the introduction of CTAD 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:

```cpp
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.

9.6 Impacts on the specialization of class templates

Universal template arguments enable what appears like overloading of class templates by specializing a unimplemented primary template taking a pack of UTPs.

```
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>;
```

As the primary template is not defined there is only a default constructor implicit deduction guide. The explicit deduction guides select the first specialization unless an allocator object is given in which case the second specialization is selected. To select the third specialization the template arguments must be explicitly given.

As default values can only be given for the primary template this is not a perfect emulation of class template overloading, instead additinal specializations are needed to emulate use of the defaulted parameters.

9.7 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>>>
{
    // Implementation for templates with floating point parameter types
};

template <typename T>
struct X {};

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

The covariant behavior of `template auto` allows a similar pattern to be used for NTTPs.

```
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
```

This pattern cannot currently be used with NTTPs as `auto` behaves contravariantly:

```
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`
```

With partial specialization, covariant behavior is desired. We rely on SFINAE to check that `T` accepts `U` and wish to accept any type of template which can take an integral NTTP in this example. `template auto` allows

such a pattern to work without modifying the behavior of `auto`.

9.8 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 currently as selecting the base case should trigger an error, but will be selected when none of the specialziations matches. This is solved by [P2041R0], which is assumed in the example below.

```
// 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:

```
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.

9.9 Naming the facility

In this proposal we have choosen to use a spelling of this feature for demonstration purposes. This is not necessarily a suitable spelling. In this section we investigate this and start a so called bike shedding procedure to decide the final spelling of the feature.

9.9.1 Problems with template auto

The initially suggested spelling `template auto` may be problematic in two ways. The most pressing is explicit template instantiation, which has valid use for `template auto` already in the current standard:

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

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

While this is not a parsing problem as such as long as UTPs are just that, it would prevent using the same spelling for an universal alias extension.

In a future standard version variable templates may be allowed as template parameters, with a close enough spelling to cause worries:

```
template<template auto UTP> class X;
template<template<typename> auto VAR> class Y;
```

Here X takes a UTP while Y takes a variable template of any type. The only difference is the `<typename>` part.

9.9.2 New keyword or combination of existing keywords

A distinct feature like UTPs would generally require a new keyword. Due to the extremely large amount of C++ code having been produced introducing new keywors always comes with risks of breaking previously working code. To be noted is however that C++23 introduced 8 new keywords, including the relatively common words `concept` and `requires`.

To avoid this issue for a relatively specialist feature like universal template parameters we initially selected to not introduce a new keyword and instead rely on a combination of two existing keywords. This is a way of thinking that has not been used before in C++, even when keywords are stacked, as they often are in declarations, each of those keywords participate in specifying the declaration in their own right:

```
static const unsigned short int x;
const int f() override noexcept final;
```

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
template auto
auto template
auto typename
typename auto

… | New keywords | | :————————: | | any name | | any kind | | auto kind | | unknown kind | | universal name | | universal kind | | univ name | | univ kind | | indeterminate | | indet name | | dependent name | | dep name |

9.9.3 Keyword discussion

If a new keyword is to be recommended the author's list 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.

9.9.4 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.

After considering these aspects the authors came up with the recommendation `xxxxxxx` which we feel conveys the semantics and is odd enough to not break much code. TODO: Scan github!

9.9.5 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.

9.10 Integration with reflection

The authors expect that code using reflection will have a need for this facility. We should not reach for code-generation when templates will do, and so being able to pattern-match on the result of the reflection expression might be a very simple way of going from the consteval + injection world back to template matching.

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

Importantly, the authors do not see how one could write `is_specialization_of` with the current reflection facilities, because one would have no way to pattern-match. This is, however, also pending discussion with the authors of reflection.

9.11 Library fundamentals II TS detection idiom

TODO. Concievably simplifies the implementation and makes it far more general.

10 Discussion on contravariance correctness

Introduce a way to specify a truly UTP that can bind to anything usable as a template argument.

There are two ways we can go about specifying this, depending on whether we want to be mathematically correct, or less powerful, imprecise, but "easy" to use. (See the co/contravariance discussion below).

10.1 "Easy" version

We spell it `template auto`. The syntax is somewhat up for debate in this version of the solution.

```
template <template <template auto...> typename F, template auto... Args>
using apply = F<Args...>;

template <typename Type> struct G {};
template <int i> struct H {};
template <template <typename> typename Template> struct J {};

apply<G, int>; // OK, G<int>
apply<H, 3>;   // OK, H<3>
apply<J, G>;   // OK, J<G>
apply<J, H>;   // error, H takes int.
```

10.2 "Mathematically Correct" version

We need to introduce both a an *anything* parameter (which we'll spell `template auto` as above), and a *wildcard* parameter (which, given the work in pattern matching, we are probably free to spell `__`).

We can then define `apply` as follows:

```
template <template <__...> typename F, template auto... Args>
using apply = F<Args...>;
```

`fwd` is similar:

```
template <template <__> typename F, temmplate auto arg>
using fwd = F<arg>;
```

In this mechanism, `__` really is an "unconstrained" pattern match, and does not declare a parameter kind - it's illegal to declare a `template <__ x> struct foo {};`, for instance.

10.2.1 Example of the difference between mathematically correct and easy versions.

The basic premise about "mathematically correct" is that the user of the template template parameter must be able to instantiate it in all ways that the declaration indicates. This means that if a template parameter is declared `template<template auto...> typename F` only an argument with the same declaration can be substituted, as this is the only way that F can be instantiated with any number of template parameters of any kind.

In the "Easy" version it is enough that the template argument can be instantiated with *some* template parameters out of the ones permitted by the template parameter declaration.

```
template<template<template auto...> class TPL> void f()
{
    TPL<int, 3> myObject;       // #1
}

f<std::vector>();               // #2
f<std::array>();                // #3
```

With the easy definition both instantiations at #2 and #3 are fine, but then we get substitution failure for #2 at #1 as `std::vector` can't take a numerical second template argument. At #3 we get no errors as f() *happened* to instantiate TPL only with template argument kinds that the `std::array` supports.

With the mathematically correct definition both #2 and #3 fail to substitute into TPL as none of `std::array` and `std::vector` have a variadic UTP, and thus can't support whatever instances that f() may want to create.

To be able to pass any template as the argument to a template parameter in the mathematically correct definition we must use another syntax for the template parameters of the template template parameter, such as `___`. This allows any class template to be used as the template argument and defers checking until inside the instantation process.

```
template<template<__...> class TPL> void f()
{
    TPL<int, 3> myObject;       // #1
}

f<std::vector>();               // #2
f<std::array>();                // #3
```

Here #2 fails late at #1 and #3 succeeds just as for the easy definition.

11 Covariance and Contravariance

11.1 As of C++23

Consider the two concepts:

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

Covariance	Contravariance

`auto returns_a() -> A;`
`auto returns_b() -> B;`
`auto f() {`
`  // OK, requirement less constrained than re`
`  A x = returns_b();`
`  // Error, requirement stricter than returne`
`  B y = returns_a();`
`}`
	`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>;`

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
```

TODO: Ask James Touton where in the standard this is made an error.

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);
```

It does work for virtual function covariant return types:

```
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 ways (but also covariantly) - not what one would expect given the concepts example above.

11.2 Design space exploration for `template auto`

We have inconsistent behavior across the language. Let's say we did the right 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 do we write `apply`? Let's do it for a single argument to avoid complications with a covariant `...`:

```
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;
};
```

If we relaxed the rule that only type concepts could appear in the template parameter list, we could say this:

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

This relaxation is *unlikely to happen* (given the past oral arguments in EWG), but we can still constrain with `requires`, albeit that is a far inferior-looking strategy.

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 to 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 library-provided concept. It is also consistent with the rest of the non-concept template language.

Of course, the library-concept solution requires an additional relaxation of constraints syntax, but at least it's feasible in the future, as opposed to requiring a second difficult-to-teach token sequence.

12 Other Considered Syntaxes

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

12.1 `.` 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.

13 Compendium: Open Design Questions

See discussions above to inform these choices; this section just compiles the design questions for EWG and EWGI.

13.1 Eager or Late Checking

Whether to:

— specify **eager** checking with conservative rules on use of identifiers, or;
— specify **late** checking on instantiation and use the C++23 rules for parsing dependent expressions for them.

13.2 Easy or Correct

Whether to:

— (**correct**) reserve *two* token sequences (`template auto` and `__`) and have a single defined meaning for both, or;
— (**easy**) reserve *one* token sequence (`template auto`) and switch its meaning in a context-dependent manner, losing the ability to mean `template auto` in a *template template-parameter*, since in that context, `template auto` would mean `__`.

14 Further work (for context and planning): Cohesive template parameter theory

This work in **not a part of the proposal**, but we mention it here because this paper falls into the wider framework of cleaning up template parameter theory.

This section is basically wholy thought up by James Touton. We thank him for his ideas.

The core of the problem is that `...` behaves in a context-dependent manner; the "easy" way, as outlined above.

We fundamentally have the following list of template parameter constraints (ignoring concepts - those merely further constrain these atomic kinds):

- atomic constraints (may introduce identifiers, read "accepts" before the bullet):
  - anything: `template auto`
  - type: `typename T`
  - value, type-unconstrained: `auto V`
  - value, type-constrained: `int I`
  - metafunction-of: `template <parameter-constraints> typename F`
  - any-number-of *constraint*: `constraint...`
- metafunction parameter constraints:
  - atomic constraints (accepts *atomic constraint*):

```
template <template <auto...> typename F> struct f {
   // the 'auto' part: F accepts any value
   F<1>    x;      // instantiated with an int
   F<true> y;      // and also a bool
   // the '...' part: F handls any arity
   F<true, 1ll> z; // and also a bool and a 1ll
   F<> w;          // and also without params
   static_assert(!std::is_same_v<decltype(x), decltype(y)>);
};
```

  - constraint relaxation (means "no constraint in this direction"):
    - "no constraint on arity": C++23 `...`

```
template <template <int ...> typename F, int ... Args>
using apply_ints = F<Args...>;

template <auto V> struct f;
using t1 = apply_ints<f, 1>; // works

template <int x> square : std::integral_constant<int, x*x> {};
using t2 = apply_ints<square, 2>; // also works
```

    - "no constraint on kind": covariant `template auto` or `__`
    - "no constraint on type": covariant `auto`, no proposed spelling yet

We need some way to disambiguate the `...` case to be correct.

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 attendence in committee meetings.

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

[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

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