A Simple Approach to Universal Template Parameters

Document #:	P2989R1
Date:	2024-02-15
Programming Language C++	
Audience:	EWG
Reply-to:	Corentin Jabot <corentin.jabot@gmail.com>
	Gašper Ažman <gasper.azman@gmail.com>

Abstract

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

Revisions

R1

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

R0

- Initial revision

Difference with P1985R3 [3]

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

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

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

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

Examples

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

Template CPOs

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

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

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

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

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

Universal template parameters solve that problem nicely:

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

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

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

[Full example on Compiler Explorer].

Generic rebind

Consider this allocator:

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

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

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

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

template <template <class, universal template...> typename Alloc, class T, universal template
    ... Args, class U>
struct rebind_alloc_t<Alloc<T, Args...>, U> {
    using type = Alloc<U, Args...>;
};
template <class Alloc>
struct allocator_traits {
    template<class T>
    using rebind_alloc = rebind_alloc_t<Alloc, T>::type;
};
using rebound = allocator_traits<BlockAllocator<int>>::rebind_alloc<long>; // now ok
using rebound2 = allocator_traits<BlockAllocator<int, 128>>::rebind_alloc<long>; // ok too
```

[Demo on compiler explorer]

`ranges::to`

The code :

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

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

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

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

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

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

Design

The design involves as few elements as possible.

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

Partial ordering of universal template parameters

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

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

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

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

Deduction of universal template arguments

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

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

typenames

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

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

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

template names

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

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

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

Expressions

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

We considered the following options:

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

  Consequently, the following would be ill-formed:

  ```
  template <int&> struct A { };
  ```

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

And it makes partial specialization somewhat subtle:

```
static constexpr int a = 0;

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

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

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

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

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

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

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

- Defer type deduction of *id-expressions*.

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

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

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

Under this model:

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

int i;

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

To achieve this behavior, *id-expression*s bound to a universal template parameter would behave as sort of symbolic references to the variable they denote, a bit like structured binding of aggregates. They would presumably always be type-dependent even if the variable they denote is not. We think this might be the better model, as it allows ""perfect forwarding"" of variables through universal parameters.

However, this idea came quite close to Kona, such that we have not had time to implement it or to iron out the specifics. Nevertheless, we think this approach is the most promising, as it is uncompromising.

- We considered letting users control whether to deduce a reference with some syntax, but this would add more complexity than it would solve problems. Deducing the exact type has the big advantage of making universal template parameters always be the exact same thing as the argument from which they are deduced, which is easier to teach.

Concepts

Because concepts cannot be specialized, and because we have no good motivation for doing otherwise, we do not propose to support universal template parameters in the template heads of concepts. Universal template parameters also cannot be constrained with anything resembling a *type-constraint*. However, universal template parameter names can appear in template arguments in `requires` clauses. Constraining an entity to accept only specific kinds of universal template parameters is therefore possible.

Function template

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

Library Support: Universal template parameters with `is`

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

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

These type traits can be specialized as follow:

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

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

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

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

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

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

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

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

[Demo on compiler explorer]

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

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

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

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

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

[Demo on compiler explorer]

In which we mention reflection

One of the things we do not propose in this paper is to force the interpretation of a universal template as a specific kind ”in place”, in contrast with P1985R3 [3]. That is, to use a universal template parameter, you have to pass it to another template, which is then specialized for templates, NTTP, template names, and so on. Naming a universal template parameter anywhere except as a template argument is ill formed.

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

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

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

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

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

[Demo on Compiler Explorer]

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

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

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

Quoting from P1240R2 [8]:

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

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

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

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

Syntax

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

- `universal template Foo` works (i.e., is not ambiguous). A search for `#define universal` finds one instance in a no-longer maintained project.
- `template auto` which is the syntax used by circle and P1985R0 [2] reused `auto` in inconsistent ways which we and many others found undesirable.
- `template` as an isolated keyword is perfectly fine, but some committee members have expressed opposition to that.
- `__any` or `__universal` would also work, but C++ doesn't traditionally embrace underscore-prefixed keywords (unlike C).
- `universal_template` or similar would also work but again we don't often use underscore in keywords (though, `co_yield` and co do).
- `anytmplarg` or any such weird enough keyword to be unlikely to be used as identifier would also work. (we can't pick something non-weird, to not break existing code).

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

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

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

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

```
template <
   auto, // variable
   template auto, // universal parameter
   template <auto> auto, // variable template
```

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

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

Status of this proposal and implementation

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

We provide a prototype implementation in Clang to demonstrate implementability in a second C++ compiler and discover any interesting design questions we might otherwise miss. Our implementation compile the examples presented in this paper. However it remains a prototype that is not production ready. The implementation represented a few weeks of work. The main challenge of bringing this proposal to production ready implementations certainly resides in the testing of all possible combination of template parameter and arguments.

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

Wording

TBD!

Acknowledgments

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

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

We also want to thank Lori Hughes for helping edit this paper on short notice.

References

[1] Ruslan Arutyunyan and Alexey Kukanov. P2769R1: get_element customization point object. https://wg21.link/p2769r1, 5 2023.

[2] Gašper Ažman and Mateusz Pusz. P1985R0: Universal template parameters. https://wg21.link/p1985r0, 1 2020.

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

[4] Lewis Baker, Corentin Jabot, and Gašper Ažman. P2547R1: Language support for customisable functions. https://wg21.link/p2547r1, 7 2022.

[5] Walter E Brown and Bob Steagall. P2098R1: Proposing std::is_specialization_of. https://wg21.link/p2098r1, 4 2020.

[6] Corentin Jabot and Gašper Ažman. P2841R0: Concept template parameters. https://wg21.link/p2841r0, 5 2023.

[7] Barry Revzin and Andrew Sutton. P1170R0: Overload sets as function parameters. https://wg21.link/p1170r0, 10 2018.

[8] Daveed Vandevoorde, Wyatt Childers, Andrew Sutton, and Faisal Vali. P1240R2: Scalable reflection. https://wg21.link/p1240r2, 1 2022.