Universal Template Parameters

Document #: D1985R2
Date: 2020-05-10
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>

Contents

1 Introduction

We propose a universal template parameter. This would allow for a generic `apply` and other higher-order template metafunctions, as well as certain type traits.

2 Change Log

2.1 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 leave the co/contravariance as is.

2.2 R0 -> R1

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

3 Motivation

Imagine trying to write the `apply` metafunction. While `apply` is very simple, a metafunction like `bind` or `curry` runs into the same problems; for demonstration, `apply` is clearest.

It works in C++20 if one only considers type template parameters:

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

template <typename X>
class G { using type = X; };

static_assert(std::is_same<apply<G, int>, G<int>>{}); // OK
```

As soon as `G` tries to take any kind of NTTP (non-type template parameter) or a template-template parameter, `apply` becomes impossible to write; we need to provide analogous parameter kinds for every possible combination of parameters:

```
template <template <class> class F>
using H = F<int>;
apply<H, G> // error, can't pass H as arg1 of apply, and G as arg2
```

4 Proposed Solution

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

Let's spell it `template auto`. The syntax is the best we could come up with; but there are plenty of unexplored ways of spelling such a template parameter.

As an example, let's implement `apply` from above:

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

```
apply<G, int>; // OK, G<int>
apply<H, G>;   // OK, G<int>
```

4.1 Mechanism

4.1.1 Specializing class templates on parameter kind

The new universal template parameter introduces 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> : std::integral_constant<decltype(val), val> {
  // val is an NTTP
};

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

This alone 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.

4.1.2 Eager or late checking?

There exists a choice in whether we check the usage of a universal template parameter eagerly or late. Consider:

```
template <template auto Arg>
struct Y {
  using type = Arg; // can only alias a type
  // Q: late (on Y<1>) or early (on parse) error?
};
```

We could either:

- Eagerly check this and hard-error on parsing the template.
- Late-check and hard-error only if `Y` is *instantiated* with anything but a type.

This paper **proposes eager checking**, thus allowing the utterance of a universal template parameter only in:

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

This avoids any parsing issues by being conservative. We can always extend the grammar to allow usage in more contexts later.

Examples that inform the following reasoning:

```
template <template auto Arg>
void f() {
  using type = Arg<int>; // error, not valid for types and values
  using value = Arg.foo(); // error, not valid for types and class templates
  auto x = Arg::member; // error, not valid for class templates and values
};
```

We avoid ambiguity by allowing the use of the universal template parameter solely in ways that are valid for any kind of thing (value, type, or class template) it may represent. It just so happens that the only such use is passing it as an argument to a template and wait for something else to pattern-match it out.

4.1.3 Template-template parameter binding

We have to consider the two phases of class template (and template alias) usage: parsing, and substitution.

Let's take `apply` again:

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

This *parses*, because `F` is *declared* as a class template parameter taking a pack of `template auto`, while `Args` is being expanded into a place that takes `template auto` parameters.

When we pass a metafunction such a `is_same` as `F`:

```
using result = apply<std::is_same, int, 1>; // error
```

everything is known, so the compiler is free to check dependent expansions at instantiation time.

4.2 Clarifying examples

4.2.1 Single parameter examples

```
template <int>                          struct takes_int {};
template <typename T>                   using  takes_type = T;
template <template auto>                struct takes_anything {}
template <template <class> class F> struct takes_metafunc {};

template <template <template auto> class 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*).

4.2.2 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 + ...)> {};

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

4.2.3 Example of parsing ambiguity (if late check, not proposed)

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.

Original example from [P0945R0]:

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

5 Example Applications

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

5.1 Enabling higher order metafunctions

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

Further example: `curry`:

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

5.2 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...>
inline constexpr bool dependent_bool = value;
template<template auto... Args>
inline constexpr bool dependent_false = dependent_bool<false, Args...>;

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

5.3 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 an specialization of a given template. When our templates mix types and NTTPs, this trait is currently impossible to write in general. However, with the universal template parameter, we can write a concept for that easily as follows.

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

template<template auto... Params, template<template auto...> class Type>
inline constexpr bool is_specialization_impl<Type<Params...>, Type> = true;

template<typename T, template<template auto...> class Type>
concept is_specialization_of = is_specialization_impl<T, Type>;
```

With the above we are able to easily constrain various utilities taking class templates:

```
template <auto N, auto D>
struct ratio {
  static constexpr decltype(N) n = N;
  static constexpr decltype(D) d = D;
};

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

or create named concepts for them:

```
template<typename T>
concept is_ratio = is_specialization_of<ratio>;

template<is_ratio R1, is_ratio R2>
using ratio_mul = simplify<ratio<
```

```
    R1::n * R2::n,
    R1::d * R2::d
  >>;
```

This concept can then be easily used everywhere:

```
template <is_specialization_of<std::vector> V>
void f(V& v) {
  // valid for any vector
}
```

5.4 Universal alias as a library

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:

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

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

template <auto V>
struct alias<V> : std::integral_constant<decltype(V), V> {};

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

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

```
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
```

5.5 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 universal template parameters.

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

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

```
template<typename T, typename A> class 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> class 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.

5.6 Impacts on the specialization of variable templates

Universal template parameters 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 [**???**], which is assumed in the example below.

```
// Metafunction to find a tuple element by a type predicate.
template<template<typename> class Pred, size_t Pos, typename Tuple>
constexpr size_t tuple_find() {
    if constexpr (Pos == tuple_size<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> class 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...> TPL, template auto... Bs> struct curry {
    template<template auto... Ts> using func = TPL<Bs..., Ts>;
};

template<template auto... Ps>
constexpr auto tuple_find_v = delete;

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

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

// Convenience specialization for use with binary predicate
```

```
template<template<typename, typename> class 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 is a metafunction to find a matching element in a tuple based only on its type. Unfortunately in C++20 `tuple_find` can only be implemented as a constexpr function if we want it to be usable with or without the start position (which mimics the `std::find` function in the value domain).

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

Given a further overloaded constexpr function `tuple_size` we could simplify this to:

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

This relies on the power of universal template parameters in another way, and has the same simplicty 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 universal template parameters as shown in the previous example instead of as the function it must be in C++20.

5.7 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.

5.8 Library fundamentals II TS detection idiom

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

6 Covariance and Contravariance

Consider the two concepts:

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

Covariance	Contravariance

Covariance

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

Contravariance

```
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>; // OK??? MSVC, GCC and clang all accept
```

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>; // Error in 14, OK in 17 and 20.
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.

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 template, 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") `auto template`:

```
template <template <auto 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 `args` 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 `auto template` behaves above) if used that place. 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 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.

7 Other Considered Syntaxes

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

7.1 `.` and `...` instead of `template auto` and `template auto ...`

```
template<template<...> class 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.

8 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.

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

9 References

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

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

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

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