Universal Template Parameters

Document #: P1985R0
Date: 2020-01-13
Project: Programming Language C++
Evolution Working Group Incubator
Reply-to: Mateusz Pusz (Epam Systems)
<mateusz.pusz@gmail.com>
Gašper Ažman
<gasper.azman@gmail.com>

Contents

1 Introduction 1
2 Motivation 1
3 Proposed Solution 2
4 Implications 2
5 Example Applications 3
  5.1 Enabling higher order metafunctions 3
  5.2 Making dependent static_assert(false) work 3
  5.3 Checking whether a type is an instantiation of a given template 3
6 Other Considered Syntaxes 4
  6.1 ... instead of template auto and template auto 4
7 Acknowledgements 4
8 References 4

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 Motivation

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

It works for pure types:

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

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

```cpp
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
```

3 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:

```cpp
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 Implications

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:

```cpp
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<C> {
    // C 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.
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.

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:

```cpp
// 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> 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 an instantiation of a given template

When writing template libraries, it is useful to check whether a given type is an instantiation 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.

```cpp
// is_instantiation_of
template<typename T, template<template auto...> typename Type>
inline constexpr bool is_instantiation_impl = false;
template<template auto... Params, template<template auto...> typename Type>
inline constexpr bool is_instantiation_impl<Type<Params...>, Type> = true;
template<typename T, template<template auto...> typename Type>
concept is_instantiation_of = is_instantiation_impl<T, Type>;
```

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

```cpp
template <auto N, auto D>
struct ratio {
    static constexpr decltype(N) n = N;
    static constexpr decltype(D) d = D;
};
template<is_instantiation_of<ratio> R1, is_instantiation_of<ratio> R2>
using ratio_mul = simplify<ratio<
or create named concepts for them:

```cpp
template<typename T>
concept is_ratio = is_instantiation_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:

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

## 6 Other Considered Syntaxes

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

### 6.1 . and ... instead of template auto and template auto ...

```cpp
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.

## 7 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.

## 8 References

   https://wg21.link/p1830r1

   https://wg21.link/p1936r0