Universal Template Parameters

Document #:	D1985R0
Date:	2019-12-23
Project:	Programming Language C++ Evolution Incubator Working Group

Reply-to:	Mateusz Pusz <mateusz.pusz@gmail.com> Gašper Ažman <gasper.azman@gmail.com>

Contents

1 Introduction

We propose a way to spell a universal template parameter kind. This would allow for a generic apply and other higher-order template metafunctions, and certain type traits.

2 Motivation and Scope

For brevity of the following motivating examples let's assume that:

— `.` in a template parameters list means a single parameter of any kind
— `...` in a template parameters list means a parameter pack of template parameters of any kind

The above syntax itself is not being proposed as of now and is just used to explain the idea of the feature:

2.1 Declare a class template with parameters of any kind

```
// is_instantiation
template<typename T, template<...> typename Type>
inline constexpr bool is_instantiation_impl = false;

template<... Params, template<...> typename Type>
inline constexpr bool is_instantiation_impl<Type<Params...>, Type> = true;

template<typename T, template<...> typename Type>
concept is_instantiation = is_instantiation<T, Type>;
```

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

```
template<is_instantiation<ratio> R1, is_instantiation<ratio> R2> using ratio_add = _see below_;
```

or create named concepts for them:

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

template<is_ratio R1, is_ratio R2> using ratio_add = _see below_;
```

2.2 “Overload sets” of partial specializations of a class template

Before C++20 all template parameters were unconstrained. With this we had to name every class template differently to clearly identify which template we are going to instantiate:

```
template<typename Child, typename Dim, basic_fixed_string Symbol, typename PT>
struct named_coherent_derived_unit { using unit = ::unit</* ... */>; };

template<typename Child, typename Dim>
struct coherent_derived_unit { using unit = ::unit</* ... */>; };

template<typename Child, typename Dim, basic_fixed_string Symbol, typename R, typename PT = no_prefix>
struct named_scaled_derived_unit { using unit = ::unit</* ... */>; };

template<typename Child, typename Dim, basic_fixed_string Symbol, typename PT, typename U, typename... Us
struct named_deduced_derived_unit { using unit = ::unit</* ... */>; };

template<typename Child, typename Dim, typename U, typename... Us>
struct deduced_derived_unit { using unit = ::unit</* ... */>; };

template<typename Child, typename P, typename U>
struct prefixed_derived_unit { using unit = ::unit</* ... */>; };
```

All of the above are factory class templates (all of them result in an instantiation of a `unit` class template) which can be considered a direct counterpart of factory functions. The above can be represented in functions domain as:

```
unit named_coherent_derived_unit(void* dimension, void* symbol, void* prefix_type);
unit coherent_derived_unit(void* dimension);
unit named_scaled_derived_unit(void* dimension, void* symbol, void* ratio, void* prefix_type);
unit named_deduced_derived_unit(void* dimension, void* symbol, void* prefix_type, void* unit, ...);
unit deduced_derived_unit(void* dimension, void* unit, ...);
unit prefixed_derived_unit(void* prefix, void* unit);
```

Above is probably not the best example of a C++ code ;-) In C++ we have strong types so we can type:

```
unit named_coherent_derived_unit(dimension dim, string symbol, prefix_type pt);
unit coherent_derived_unit(dimension dim);
unit named_scaled_derived_unit(dimension dim, string symbol, ratio r, prefix_type pt);
unit named_deduced_derived_unit(dimension dim, string symbol, prefix_type pt, unit u, ...);
unit deduced_derived_unit(dimension dim, unit u, ...);
unit prefixed_derived_unit(prefix p, unit u);
```

but this immediately leads us to an overload set:

```
unit derived_unit(dimension dim, string symbol, prefix_type pt);
unit derived_unit(dimension dim);
unit derived_unit(dimension dim, string symbol, ratio r, prefix_type pt);
unit derived_unit(dimension dim, string symbol, prefix_type pt, unit u, ...);
```

```
unit derived_unit(dimension dim, unit u, ...);
unit derived_unit(prefix p, unit u);
```

Coming back to our factory class templates, with C++20 we got concepts and now we can (and probably should) constrain most of the template parameters:

```
template<typename Child, Dimension Dim, basic_fixed_string Symbol, PrefixType PT>
struct named_coherent_derived_unit { using unit = ::unit</* ... */>; };

template<typename Child, Dimension Dim>
struct coherent_derived_unit { using unit = ::unit</* ... */>; };

template<typename Child, Dimension Dim, basic_fixed_string Symbol, Ratio R, PrefixType PT = no_prefix>
struct named_scaled_derived_unit { using unit = ::unit</* ... */>; };

template<typename Child, Dimension Dim, basic_fixed_string Symbol, PrefixType PT, Unit U, Unit... Us>
struct named_deduced_derived_unit { using unit = ::unit</* ... */>; };

template<typename Child, Dimension Dim, Unit U, Unit... Us>
  requires U::is_named && (Us::is_named && ... && true)
struct deduced_derived_unit { using unit = ::unit</* ... */>; };

template<typename Child, Prefix P, Unit U>
  requires (!std::same_as<typename U::prefix_type, no_prefix>)
struct prefixed_derived_unit { using unit = ::unit</* ... */>; };
```

But again, taking above into account we could form an "overload set" of class template partial specializations:

```
template<typename...>
struct derived_unit;

template<typename Child, Dimension Dim, basic_fixed_string Symbol, PrefixType PT>
struct derived_unit<Child, Dim, Symbol, PT> { using unit = ::unit</* ... */>; };

template<typename Child, Dimension Dim>
struct derived_unit<Child, Dim> { using unit = ::unit</* ... */>; };

template<typename Child, Dimension Dim, basic_fixed_string Symbol, Ratio R, PrefixType PT = no_prefix>
struct derived_unit<Child, Dim, Symbol, R, PT> { using unit = ::unit</* ... */>; };

template<typename Child, Dimension Dim, basic_fixed_string Symbol, PrefixType PT, Unit U, Unit... Us>
struct derived_unit<Child, Dim, Symbol, PT, U, Us...> { using unit = ::unit</* ... */>; };

template<typename Child, Dimension Dim, Unit U, Unit... Us>
  requires U::is_named && (Us::is_named && ... && true)
struct derived_unit<Child, Dim, U, Us...> { using unit = ::unit</* ... */>; };

template<typename Child, Prefix P, Unit U>
  requires (!std::same_as<typename U::prefix_type, no_prefix>)
struct derived_unit<Child, P, U> { using unit = ::unit</* ... */>; };
```

But the above code does not compile as the partial specialization have not only a varying number of template arguments but also mix type and non-type template parameters. Specifying a primary class template with a universal template parameter pack to just state "`derived_unit` is a class template" would make this work:

```
template<typename...>
struct derived_unit;
```

2.3 Better higher order functions

Universal template parameter kind opens the doors to better higher order functions:

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

template<int x, int y, int z>
using plus3 = x + y + z;

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

2.4 Making dependent `static_assert(false)` work

Dependent static assert idea is described in P1936 and P1830. 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**.

With the feature proposed by this paper the solution could look like:

```
template<bool value, ... Args>
inline constexpr bool dependent_bool_value = value;
```

```
template<... Args>
inline constexpr bool dependent_false = dependent_bool_value<false, Args...>;
```

```
template<typename... Args>
struct my_struct
{
 static_assert(dependent_false<Args...>);
};
```

3 Bikeshedding

The syntax used in our examples is terse and probably even good for a universal template parameter pack (`...`) but we expect that the syntax used for a single template parameter (`.`) will be controversial. There are many possible ways to spell it and we leave the concrete spelling discussion for the big-room discussion.

Here are a few proposals to start with:

1. `.` and `...`

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

2. `template auto` and `template auto...`

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