A Generic Library for Compile-time Routing

Document number:	P1649R0
Date:	2019-06-16
Project:	Programming Language C++
Audience:	LEWG, LWG

Authors:	Mingxin Wang (Microsoft (China) Co., Ltd.)
Reply-to:	Mingxin Wang <mingxwa@microsoft.com>

Table of Contents

A Generic Library for Compile-time Routing 1
1 Introduction 1
2 Motivation 1
2.1 Type Selection 2
2.2 Path Selection 3
2.2.1 A Motivating Problem 3
2.2.2 Solve with `if constexpr` 4
2.2.3 Solve with the Proposed Library 6
3 Technical Specification 7
3.1 Header <utility> synopsis 7
3.2 Class template `applicable_template` 7

1 Introduction

`if constexpr` and "SFINAE/Concepts based class/function template specialization" are generally used for compile-time type/path selection (routing) in complex template libraries. However, according to my experience, they are not so easy to code, maintain or test. Therefore, I designed a template library specifically for compile-time routing with more usability, enabling template libraries to select type templates with flexible routing rules at compile-time without dependent specializations.

This proposal will compare several existing ways and the proposed library to implement compile-time routing with a motivating example, then illustrate the technical specifications of the library.

Here is a sample implementation of the library and it has already been used in the implementation of the Extending Argument library [P1648R0] and the Concurrent Invocation library [P0642R2].

2 Motivation

According to my experience, "SFINAE/Concepts based class/function template specialization" is usually used in

compile-time implementation selection for types, and **`if constexpr`** is widely adopted in compile-time path selection. However, I think they have certain limitations in engineering.

2.1 Type Selection

"SFINAE/Concepts based class/function template specialization" only support one default implementation with a lower priority and multiple specializations with a same higher priority. Therefore, it is inconvenient to define a type template with a set of selection rules that is not the same as class template specialization. For example,

```
template <class T> class X { /* ... */ };  // #1
template <class T> requires A<T> class X<T> { /* ... */ };  // #2
template <class T> requires B<T> class X<T> { /* ... */ };  // #3
template <class T> requires C<T> class X<T> { /* ... */ };  // #4
```

In the declaration of **`X`** above, the default implementation (#1) has the lowest priority, and the rest three (#2~4) specializations has a same higher priority. When specifying **`X`** with a class **`T`**, **`A<T>`**, **`B<T>`** and **`C<T>`** shall be well-formed and zero or strictly one of the constraints is satisfied.

However, when we want to change the priority in the sample code by assigning the highest priority to implementation #2, it will become complicated. Before this proposed library was designed, I have tried two ways:

1. Narrow the constraints on implementation #3 and #4:
   ```
   template <class T> requires (!A<T> && B<T>) class X<T> { /* ... */ };  // #3
   template <class T> requires (!A<T> && C<T>) class X<T> { /* ... */ };  // #4
   ```

2. Rely on another "helper class":
   ```
   template <class T> class Y { /* ... */ };  // #1
   template <class T> requires B<T> class Y<T> { /* ... */ };  // #3
   template <class T> requires C<T> class Y<T> { /* ... */ };  // #4

   template <class T> class X : public Y<T>
      { /* delegated ctor and assignments */ };  // delegate #3 and #4
   template <class T> requires A<T> class X<T> { /* ... */ };  // #2
   ```

For the first solution, although it generally works, it is not rigorously reasonable, because it implicitly requires that **`B<T>`** and **`C<T>`** shall always be well-formed even when **`A<T>`** is already well-formed to perform template resolution. Moreover, if there are more specializations for X and more priorities, the constraints on each specialization will become increasingly difficult to maintain or read.

For the second solution, the good thing is that **`Y<T>`** will not be instantiated if **`A<T>`** is satisfied, and the constraints will be easier to maintain comparing to the previous solution. However, it is still confusing since it introduces another vocabulary **`Y`**, and the default implementation of **`X`** shall define its constructor and assignments delegating every potential implementation of **`Y`** (each specialization of **`Y`** may have various customized constructors) for priority. But after all, it could be a correct solution.

To simplify illustration, the term **compile-time routing** is defined to describe the requirements where different code shall be generated based on type traits. To solve a compile-time routing problem with the proposed library, we should define corresponding helper type traits for each route. When working with the proposed library, life will become easier,

because we are able define each specialization independently with more specific constraints and less interference with each other:

```
template <class T> requires A<T> class XA { /* ... */ };
template <class T> requires B<T> class XB { /* ... */ };
template <class T> requires C<T> class XC { /* ... */ };
template <class T> class XD { /* ... */ };
```

And we can define the priority when using the types:

```
template <class T>
using X = applicable_template<
    equal_templates<XA>,
    equal_templates<XB, XC>,
    equal_templates<XD>>::type<T>;
```

In general, the proposed library provides a mechanism to decouple the implementation of a type from template resolution. In the code above, **`applicable_template`** and **`equal_templates`** are the only facilities in the proposed library, where **`equal_templates`** is a tag representing class templates with the same priority and **`applicable_template<...>::type<...>`** will select the right template according to priority. Compile-time errors are expected if there are more than one applicable template with the same priority or there is no applicable template among all priorities.

2.2 Path Selection

2.2.1 A Motivating Problem

Suppose we are writing a simple template library for stringification with the following expression and semantics (**`s`** denotes a value of **`std::string`**):

```
template <class T>
std::string my_to_string(const T& value);
```

Effects: Stringify the input **`value`** with the following strategy:

- if **`std::to_string(value)`** is well-formed and the return value is convertible to **`std::string`**, return **`std::to_string(value)`**, or
- if **`value`** is convertible to **`std::string`**, return **`static_cast<std::string>(value)`**, or
- if **`T`** is a general tuple or container, recursively apply this function to each element in the aggregation, and return a string containing the stringified result for each element in the format of: **`[`***stringified first element*`, `*stringified second element*`, ...]`**, or
- otherwise, the expression is ill-formed.

Note that a generic tuple could be an instance of **`std::tuple`**, **`std::pair`**, **`std::array`** or other future standard facilities where **`std::get`** and **`std::tuple_size`** are well-formed; a generic container could be any standard or

customized type that support for-range loop.

For example, if the function is used as below:

```
my_to_string(
    std::make_tuple(
        123,
        std::vector<double>{1, 2, 3.14},
        std::list<std::vector<std::string>>{{}, {"Hello"}, {"W", "or", "ld"}},
        std::make_tuple(std::deque<int>{3, 2, 1}, "OK")));
```

The value of the returned **std::string** should be:

```
[123, [1.000000, 2.000000, 3.140000], [[], [Hello], [W, or, ld]], [[3, 2, 1], OK]]
```

To simplify the illustration, we may assume the following type traits are well-formed and have specific semantics:

```
template <class T>
constexpr bool is_primitive_v
    = /* whether std::to_string is applicable to a value of const T& */;

template <class T>
constexpr bool is_string_convertible_v
    = std::is_convertible_v<const T&, std::string>;

template <class T>
constexpr bool is_tuple_v = /* whether const T& is a generic tuple type */;

template <class T>
constexpr bool is_container_v = /* whether const T& is a generic container type */;
```

2.2.2 Solve with `if constexpr`

We may try **if constexpr** to solve this problem, and may come up with the code as follows:

```
template <class T>
std::string my_to_string(const T& value) {
  if constexpr (is_primitive_v<T>) {
    return std::to_string(value);
  } else if constexpr (is_string_convertible_v<T>) {
    return static_cast<std::string>(value);
  } else if constexpr (is_tuple_v<T>) {
    static_assert(!is_container_v<T>);  // #1: To avoid ambiguation
    return /* ... */;
  } else if constexpr (is_container_v<T>) {
```

```
    return /* ... */;
  } else {
    static_assert(false);  // T does not match any rule
  }
}
```

Although **`if constexpr`** works at compile-time, I think there are two defects in this implementation:

1. it may not compile because **`static_assert(false)`** may fire even if the path is not selected, and
2. for paths having the same priority (for generic tuple and containers in this case), it is required to manually maintain a list of **`static_assert`**, like the one on the marked line #1, to avoid ambiguation for each path with a same priority.

For the first issue, we could manually write a LONG **`static_assert`** at the beginning of the function, and keep it consistent with the semantics of the function:

```
static_assert(is_primitive_v<T> || is_string_convertible_v<T>
    || is_tuple_v<T> || is_container_v<T>);
```

Although it looks verbose, but it seems to be the simplest way to report potential abuse.

For the second issue, we could solve it by constrained function template overloads (**Thanks to Nicolas Lesser**). And the final solution with **`if constexpr`** may look like:

```
template <class T> requires is_tuple_v<T>
std::string my_to_string_for_aggregation(const T& value) { /* ... */ }

template <class T> requires is_container_v<T>
std::string my_to_string_for_aggregation(const T& value) { /* ... */ }

template <class T>
std::string my_to_string(const T& value) {
  static_assert(is_primitive_v<T> || is_string_convertible_v<T>
      || is_tuple_v<T> || is_container_v<T>);
  if constexpr (is_primitive_v<T>) {
    return std::to_string(value);
  } else if constexpr (is_string_convertible_v<T>) {
    return static_cast<std::string>(value);
  } else if constexpr (is_tuple_v<T> || is_container_v<T>) {
    return my_to_string_for_aggregation(value);
  }
}
```

Although the last **`if constexpr`** expression seems to be redundant, we may need it when there are potentially more paths with lower priority. Additionally, designers are also responsible for maintaining the list of constraints for the last path as there are more function overloads with a same priority.

2.2.3 Solve with the Proposed Library

Similar to type selection, we could define type traits for stringification with corresponding constraints in this example. Each implementation may contain a unique path:

```
template <class T> requires is_primitive_v<T>
struct primitive_stringification_traits {
  static inline std::string apply(const T& value)
      { return std::to_string(value); }
};

template <class T> requires is_string_convertible_v<T>
struct string_stringification_traits {
  static inline std::string apply(const T& value)
      { return static_cast<std::string>(value); }
};

template <class T> requires is_tuple_v<T>
struct tuple_stringification_traits {
  static inline std::string apply(const T& value) { return /* ... */; }
};

template <class T> requires is_container_v<T>
struct container_stringification_traits {
  static inline std::string apply(const T& value) { return /* ... */; }
};
```

Afterwards, we could implement the **my_to_string** directly with these stringification traits:

```
template <class T>
std::string my_to_string(const T& value) {
  return applicable_template<
      equal_templates<primitive_stringification_traits>,
      equal_templates<string_stringification_traits>,
      equal_templates<
          container_stringification_traits, tuple_stringification_traits>
  >::type<T>::apply(value);
}
```

Although the solution based on the proposed library look a little bit longer than the one with **if constexpr**, I think this solution is more concise and has better maintainability as well as testability because the stringification traits are independent from each other, and the priority is only defined in the implementation of **my_to_string**. A complete implementation for **my_to_string** could be found here.

3 Technical Specification

3.1 Header <utility> synopsis

The following content is intended to be merged into [utility.syn].

```
namespace std {

template <template <class...> class... TTs>
struct equal_templates {};

template <class... ETs>
struct applicable_template {
  template <class... Args>
  using type = see below;
};

}
```

3.2 Class template `applicable_template`

```
template <class... ETs>
struct applicable_template;
```

Requires: Each type in the template parameter pack **`ETs`** shall be an instantiation of the class template **`equal_templates`**.

```
template <class... Args>
using type = see below;
```

Definition: **`TT<Args...>`** if there is exactly one class template **`TT`** with the highest priority among all the class templates defined in each instantiation of **`equal_templates`** in **`ETs`** that is able to be instantiated with **`Args...`**, or otherwise, the expression is ill-formed. For any type template **`TT1`** and **`TT2`** defined in any instantiation of **`equal_templates`** in the template parameter pack **`ETs`**, the priority of the two type templates is defined as the reverse ordering of the smallest index in **`ETs`** where the type template is a template parameter of the instantiation of **`equal_templates`**.