A Generic Library for Compile-time Routing

Document number:	<unknown>
Date:	2019-05-08
Project:	Programming Language C++
Audience:	LEWG, LWG

Authors:	Mingxin Wang
Reply-to:	Mingxin Wang <mingxwa@microsoft.com>

Table of Contents

1 Introduction

`if constexpr` and "SFINAE/Concepts based class/function template specialization" are generally used for compile-time 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 library based on this library to have more extendibility and testability.

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 Extended library [P????] and the Concurrent Invocation library [P0642].

2 Motivation

2.1 A Motivating Example

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]]
```

2.2 Preconditions

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 */;
```

2.3 Solve with if constexpr

Beginners may try **`if constexpr`** to solve this question, 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>);  // 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, this code is incorrect because it never compiles. For example, if **`std::to_string`** is applicable to a value of **`const T&`** but **`const T&`** is not convertible to **`std::string`**, there will be always be a compile error in the second **`if constexpr`** body.

To avoid such compile error, we need to find a way to split these logics into separate compile units, and **`if constexpr`** won't help.

2.4 Solve with Function Template Overloads

One way to split these logics into separate compile units is to design proper overload resolution strategy using function overloads or template specializations. One may produce the following code with the SFINAE:

```
template <class T, class = std::enable_if_t<is_primitive_v<T>>>
std::string my_to_string_impl(const T& value) {
  return std::to_string(value);
}

template <class T, class = std::enable_if_t<is_string_convertible_v<T>>>
std::string my_to_string_impl(const T& value) {
  return static_cast<std::string>(value);
}

...
```

And there are two issues with the code:

1. it won't compile because the function templates will be treated as redefinition rather than overloads, and
2. even if the two function templates could be treated as different set of overloads (by using the Concepts), they share the same priority. For example, if a generic container type is convertible to **`std::string`** (for example, the conversion is defined by users), the overload resolution could be ambiguous and won't proceed.

Therefore, if we want to solve this problem with function template overloads, it is required to use the Concepts and carefully design the constraints so that to avoid ambiguation. For example:

```
template <class T> requires is_primitive_v<T>
std::string my_to_string_impl(const T& value) { return std::to_string(value); }

template <class T> requires (!is_primitive_v<T> && is_string_convertible_v<T>)
std::string my_to_string_impl(const T& value)
    { return static_cast<std::string>(value); }

template <class T> requires (
    !is_primitive_v<T>
    && !is_string_convertible_v<T>
    && is_tuple_v<T>
    && !is_container_v<T>)
std::string my_to_string_impl(const T& value) { return /* ... */; }

template <class T> requires (
    !is_primitive_v<T>
    && !is_string_convertible_v<T>
    && !is_tuple_v<T>
    && is_container_v<T>)
std::string my_to_string_impl(const T& value) { return /* ... */; }

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

Although the code works, the constraints could be error-prone as each of the basic constraints is used more than once, and could be even more obfuscated if there are more rules for the function template. For example, if we want to make a default option rather than producing a compile error if all of the rules above does not apply, and return the string "<unknown>", we will need to write another overload set of **`my_to_string_impl`** contains ALL of the basic constraints:

```
template <class T> requires (
    !is_primitive_v<T>
    && !is_string_convertible_v<T>
    && !is_tuple_v<T>
    && !is_container_v<T>)
std::string my_to_string_impl(const T& value) { return "<unknown>"; }
```

2.5 Solve with Class Template Specialization

Another way to split these logics into separate compile units, we can use class template specialization either with the SFINAE or the Concepts. If we translate the code in the last section into SFINE based class template specializations, we may need 4 different helper class templates.

The first one should judge whether **`std::to_string`** should be used or control flow should move to the second step:

```
template <class T, class = void>
struct first_step_helper {
  static inline std::string apply(const T& value)
      { return second_step_helper<T>::apply(value); }
};

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

The second one should judge whether **`static_cast<std::string>`** should be used or control flow should move to the third step:

```
template <class T, class = void>
struct second_step_helper {
  static inline std::string apply(const T& value)
      { return third_step_helper<T>::apply(value); }
};
```

```
template <class T>
struct second_step_helper<T, std::enable_if_t<is_string_convertible_v<T>>> {
  static inline std::string apply(const T& value)
      { return static_cast<std::string>(value); }
};
```

The third one should judge whether **T** should be treated as a generic tuple or control flow should move to the fourth step:

```
template <class T, class = void>
struct third_step_helper {
  static inline std::string apply(const T& value)
      { return fourth_step_helper<T>::apply(value); }
};

template <class T>
struct third_step_helper<T, std::enable_if_t<is_tuple_v<T>>> {
  static inline std::string apply(const T& value) {
    static_assert(!is_container_v<T>);  // To avoid ambiguation
    return /* ... */;
  }
};
```

And finally, the fourth step is well-formed if **T** could be treated as a generic container type:

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

With these helpers, we could finally implement the **my_to_string** function template:

```
template <class T>
std::string my_to_string(const T& value) {
  return first_step_helper<T>::apply(value);
}
```

Actually, the third and the fourth steps could merge into one step as they shall have a same priority in resolution, and could be simplified as:

```
template <class T, class = void>
struct third_step_helper;
```

```
template <class T>
struct third_step_helper<T, std::enable_if_t<is_tuple_v<T>>> {
  static inline std::string apply(const T& value) {
    static_assert(!is_container_v<T>);
    return /* ... */;
  }
};

template <class T>
struct third_step_helper<T, std::enable_if_t<is_container_v<T>>> {
  static inline std::string apply(const T& value) {
    static_assert(!is_container_v<T>);
    return /* ... */;
  }
};
```

Even though we could merge the third and the fourth steps to save some code, the implementation contains 7 declarations of types. Although it is more complication than the last solution to make it work, it has more extendibility. For example, if we want to make a default option based on the implementation, we could simply define the default implementation for the third step without using any of the basic constraints:

```
template <class T, class = void>
struct third_step_helper {
  static inline std::string apply(const T&) { return "<unknown>"; }
};
```

However, this solution is still not easy enough to maintain or test. For example, if we want to adjust the sequence of each step or add/delete steps, a large part of the code needs to be rewritten.

2.6 Solve with the Proposed Library

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. For the motivating example, we could define a series of type traits for stringification with corresponding constraints (whether implemented with the SFINAE or the Concepts):

```
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);
}
```

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. A complete implementation for **`my_to_string`** could be found here.

I think this solution is more concise than the previous solutions with function template overloads or class template specializations, 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`**.

2.7 Another Possible Solution in the Future

If C++ supports code synthesis [P0633] in the future, we may be able to implement **`my_to_string`** like:

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

Although there could be a smaller amount of code than the solution with the proposed library, I think the code synthesis solution require more efforts to maintain or test, as it will become impossible to perform syntax checking before instantiation of the function template.

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`**.