Document number: DXXXX
Date: 2015–10-20
Project: Programming Language C++, Library Evolution Working Group
Reply-to: Vicente J. Botet Escriba <vicente.botet@wanadoo.fr>

Partially specialized functions

This paper proposes to partially specialize functions on the non-deduced types.

Contents

Introduction

This paper proposes to partially specialize functions on the non-deduced types.

Motivation and Scope

Currently (C++14) we cannot partially specialize a function, we overload it.

However a template function can have additional parameters that are not deduced from the function parameters and we can not overload on these parameters as they are not deduced.

In order to obtain something similar with C++14 there are two alternatives: add a parameter that coveys the non-deduced types so that we can overload, or use a helper trait class that can be partially specialized.

The author is not a compiler guy, but wonder if partially specializing the non-deduced template parameters of a function is something that could fill in the current C++ language philosophy.

Let me show how a generic make factory could be defined using this new feature.

Next follows an overload of the make function for template classes

```
template <template <class ...> class TC, class T>
auto make(T v)
{
    return TC<T>(v);
}
```

While the type `T` is deduced, the type `TC` is not as it doesn't appears in the function parameters. We pass the parameter T by value for simplification purposes.

We can add another overload for type constructors

```
template <class TC, class T>
auto make(T v)
{
    return apply<TC,T>(v);
}
```

`TC` is not deduced here neither.

What we want is to partially specialize the non-deduced parameter `TC`.

If we adopt partial specialization for these types, we can refine the overload of the template class for the class template `future` as follows

```
template <class T>
auto make<future>(T v)
{
    return make_future_ready(v);
}
```

We have used here a variation of the syntax used for class template partial specialization.

Other possibilities would be to use auto to mean this parameter is deduced

```
template <class T>
auto make<future, auto>(T v)
{
    return make_future_ready(v);
}
```

An alternative syntax could be to make direct use of the argument on the template parameter list

```
template <future, class T>
auto make(T v)
{
    return make_future_ready(v);
}
```

The same applies for `unique_ptr` and `shared_ptr`

```
template <class T>
unique_ptr<T> make<unique_ptr>(T v)
{
    return make_unique(v);
```

```
}
template <class T>
shared_ptr<T> make<shared_ptr>(T v)
{
    return make_shared<T>(v);
}
```

and last for `expected<_, E>` type constructor

```
template <class E, class T>
expected<T,E> make<expected<_,E>>(T v)
{
    return make_expected<E>(v);
}
```

Possible usage

```
make<optional>(1);
make<future>(1)
make<expected>(1)
make<expected<_,error_code>>(1)
make<unique_ptr>(1)
make<shared_ptr>(1)
```

Possible alternative using current C++14: Forward to a function object that has an additional parameter that conveys the non deduced types that needs to be partially specialized.

We need to first a trampoline that would introduce the non-decuded type as function argument

```
template <template <class ...> class TC, class T>
TC<T> make(T v)
{
    return make(template_<TC>{}, <T>(v);
}
```

Now we need to define the default customization point

```
template <template <class ...> class TC, class T>
TC<T> make(template_<TC>,T v)
{
    return TC<T>(v);
}
```

Another trampoline for type constructors (classes for which apply<TC,T> is a new type

```
template <class TC, class T>
apply<TC,T> make(T v)
{
    return make(id<TC>{}, v);
}
```

and the corresponding default customization point

```
template <class TC, class T>
apply<TC,T> make(id<TC>, T v)
{
    return apply<TC,T>(v);
}
```

Once we have the default behavior we can overload for `future`

```
template <class T>
future<T> make(template_<future>, T v)
{
    return make_future_ready(v);
}
```

and for `expected`

```
template <class E, class T>
expected<T,E> make(id<expected<_,E>>, T v)
{
    return make_expected<E>(v);
}
```

Same usage.

Another alternative using a type trait or function object class that can be partially specialized having the default behavior for type constructors

```
template <class TC>
struct maker {
  template <class T>
  apply<TC,T> operator()(T v)
  {
    return apply<TC,T>(v);
  }
};
```

Partially specialization for templates using template_

```
template <template <class ...> class TC>
struct maker<template_<TC>> {
  template <class T>
  TC<T> operator()(T v)
  {
    return TC<T>(v);
  }
};
```

Specialization for future

```
template <>
struct maker<template_c<future>> {
  template <class T>
  future<T> operator()(T v)
  {
    return make_future_ready(v);
  }
};
```

However the usage changes, as now we need to instantiated the function object and give a type constructor.

```
maker<future<_>>{}(1);
```

This is not very friendly. In order to preserve the same syntax we need yet to add the two overloads that forward the call to the function object

```
template <class TC, class T>
TC<T> make(T v)
{
    return maker<TC>{}(v);
}
template <template <class ...> class TC, class T>
TC<T> make(template_<TC>, T v)
{
    return maker<template_<TC>>{}(v);
}
```

An alternative here is to define a template alias variable that provides a function object having a type constructor

```
template <class T>
constexpr T static_const {}; //See Eric customization paper

template <class TC>
constexpr auto const &make = static_const<maker<TC>>;
```

However the usage would be less friendly for template classes

```
make<template_<optional>>(1);
```

Permitting template aliases variable to be specialized would resolve the issue. The following would add an specialization for template classes

```
template <template <class ...> class TC>
constexpr auto &make = maker<template_c<TC>>{};
```

The advantage of the last version is that we define at a high order function object.

```
make<optional>(1)
```

Proposal

We propose then to allow partial specialization of function templates for the non-deduced types.

We could as well consider template aliases variables specialization.

Design rationale

Open points

The authors would like to have an answer to the following points if there is at all an interest in this proposal:

Technical Specification

To be completed if there is at all an interest in this proposal.

Implementation

None.

Acknowledgements

Many thanks to ....

References