Allowing Class Template Specializations in Unrelated Namespaces

Document number: P0665R0

Reply-to: Tristan Brindle tcbrindle@gmail.com

Date: 2017-06-18

Audience: Evolution Working Group

Summary

In order to make certain standard library customisation points easier to implement, we should allow specialising class templates in other namespaces without leaving the current namespace, by fully-qualifying their names (including a leading `::`), as in:

```
namespace my {

struct tuple_like { /* ... */ };

template <>
struct ::std::hash<tuple_like> { /* ... */ };

template <>
struct ::std::tuple_size<tuple_like> { /* ... */ };

template <>
struct ::std::tuple_element<tuple_like> { /* ... */ };

/* ...other declarations in my namespace... */

}
```

Motivation

Consider the following definition of a "tuple-like" type which maintains some invariant (constructor constraints etc omitted for brevity):

```
namespace my {

template <typename... T>
struct tuple_like {
    template <typename... U>
    tuple_like(U&&... args) : items_(std::forward<U>(args)...)
    {
        /* ...establish invariant... */
    }
```

```
private:
    std::tuple<T...> items_;
};

/* ...other declarations... */

}
```

Since this class maintains an invariant, it is not appropriate to make its member tuple public. However, it may still be very useful to implement support for C++17 structured bindings. In order to do so, as well as implementing a `get<I>()` (member or non-member) function in our own class or namespace, we also need to specialize the `std::tuple_size` and `std::tuple_element` classes.

To do so, we must leave the local namespace and enter namespace `std`, before re-entering the local namespace to continue with our definitions:

```
namespace my {

template <typename... T>
struct tuple_like {
    template <typename... U>
    tuple_like(U&&... args) : items_(std::forward<U>(args)...)
    {  /* ... */  }

    template <std::size_t I>
    decltype(auto) get() const { return std::get<I>(items_); }

private:
    std::tuple<T...> items_;
};

} // leave my namespace

namespace std { // enter std namespace

template <typename... T>
struct tuple_size<my::tuple_like<T...>>
        : integral_constant<std::size_t, sizeof...(T)> {};

template <size_t I, typename... T>
struct tuple_element<I, my::tuple_like<T...>>
        : tuple_element<I, const tuple<T...>> {};

} // leave std namespace

namespace my { // re-enter my namespace

/* ...other declarations... */
```

```
}
```

Breaking out of our own namespace like in this way, adding some specializations to namespace `std`, and then re-entering our own namespace is messy; it breaks the "flow" of a header file. For this reason, specializations like this are often tucked away at the bottom of a header, far away from where they (logically) belong, and where they can easily be overlooked during refactoring. With function customisation points like `get()` and `swap()` this is mitigated by the use of unqualified calls and ADL, but for class specializations like those required for hashing and structured bindings there is no alternative but to do the "namespace dance".

Proposal

We should allow defining specializations of class templates in unrelated namespaces by fully-qualifying (including a leading `::`) the name of the type being specialized, as in:

```
namespace my {

template <typename... T>
struct tuple_like {
    template <typename... U>
    tuple_like(U&&... args) : items_(std::forward<U>(args)...)
    {  /* ... */  }

    template <std::size_t I>
    decltype(auto) get() const { return std::get<I>(items_); }

private:
    std::tuple<T...> items_;
};

template <typename... T>
struct ::std::tuple_size<my::tuple_like<T...>>
        : integral_constant<std::size_t, sizeof...(T)> {};

template <size_t I, typename... T>
struct ::std::tuple_element<I, my::tuple_like<T...>>
        : tuple_element<I, const tuple<T...>> {};

/* ...other declarations... */

}
```

Questions

- Why is the initial double-colon required?

The leading `::` serves as a useful marker to the (human) reader that this is a "special specialization", in that it refers to a class template from an unrelated namespace. Moreover, it provides the full "path" to the base template.

In addition, using an initial double-colon as proposed is currently illegal, while the syntax

```
template <>
struct x::y<int> { /* ... */ };
```

can currently be used to provide a specialization of a nested class template `y` within (class or namespace) `x`. Thus there is no worry about ambiguity in the meaning of `x`.

There is no fundamental reason for requiring the initial `::` beyond this, and this restriction could be lifted in future.

- Should we be able to use this syntax to *define* types in an unrelated namespace too?

  No, that would lead to more of the sort of awkward spaghetti code we're trying to avoid. This should be strictly for class template specializations: the compiler must have seen a declaration of the base template beforehand.

- Inside the specialization, are names from the enclosing namespace in scope?

  Yes. Within the specialization, names from both namespaces should be available without qualification (except where necessary for disambiguation), as if by a `using namespace` declaration.

- Should we allow specializations of function templates with this syntax too?

  No, for two reasons. Firstly, the recommended way to implement function customisation points is to provide an overload in one's own namespace. Allowing this syntax for functions would encourage placing specializations in namespace `std` instead.

  Secondly, the syntax

  ```
  template <>
  my_type ::unrelated_ns::func<my_type>(const my_type& arg);
  ```

  looks to the compiler (and to a human at a quick glance) like `my_type::unrelated_ns::func()`, i.e. whitespace is ignored before the double colon. We could get around this by requiring that such specializations must use the trailing-return syntax (because `auto::unrelated::func()` is never allowed), but adding such a special case to parsing doesn't seem worth the benefit since, again, there is much less need to specialise function templates in other namespaces compared with classes.

- Should we allow specializations of variable templates with this syntax too?

  No. As with function templates, there doesn't seem to be sufficient motivation at this time, and the formulation

```
template <>
my_type ::unrelated_ns::var<my_type> = {};
```

is equally syntactically problematic.

Impact on existing code

The proposed syntax (with leading double-colon) is currently illegal, so there should be no impact on existing code.

Conclusion

Allowing this small syntax change strictly for providing class template specializations in unrelated namespaces will reduce boilerplate in headers/modules and make structured bindings and hashing support easier and cleaner to implement.

Proposed wording

To be supplied based on committee feedback.