Proposing `std::is_specialization_of`

Document #: WG21 P2098R1
Date: 2020–04–10
Audience: ~~LEWG-I~~[done] ⇒ LEWG ⇒ LWG
Reply to: Walter E. Brown <webrown.cpp@gmail.com>
Bob Steagall <bob.steagall.cpp@gmail.com>

Contents

Abstract

[P2078R0] "proposes the addition [to the standard library] of a new unary type traits class template, **`is_complex<T>`**." Recognizing that there are numerous similar traits templates that could profitably be added to the standard library, this paper proposes to add instead a single, more general trait that can be straightforwardly customized via a simple alias, to obtain any such trait as needed.

To think is to forget a difference, to generalize, to abstract.
— JORGE LUIS BORGES

Generalizations, like brooms, ought not to stand in a corner forever; they ought to sweep as a matter of course.
— JOHN LUKACS

1 Introduction

[P2078R0] "proposes the addition [to the standard library] of a new unary type traits class template, **`is_complex<T>`**." Such a trait can be straightforwardly implemented as follows:

```
template< class T >
struct
  is_complex : std::false_type;

template< class U >
struct
  is_complex<std::complex<U>> : std::true_type;

template< class T >
inline constexpr bool
  is_complex_v = is_complex<T>::value;
```

However, there are numerous very similar trait templates that could profitably be added to the standard library.

For example, each of the following exemplifies a useful trait much like the above **`is_complex<T>`**. Further, the authors know that these and many others have already been implemented (albeit under an uglified or slightly different name, in some cases) using essentially the same technique as was used above to implement **`is_complex<T>`**.

- **`is_reference_wrapper<T>`**,
- **`is_string<T>`**,
- **`is_pair<T>`**,
- **`is_tuple<T>`**,
- **`is_vector<T>`**,
- and many more!

Therefore, this paper proposes to **add instead a single, more general trait** that can be straightforwardly customized via a simple alias, to obtain any such trait as needed.

2 Proposal details

2.1 Expository implementation

As motivated above, we propose to add a more general **`is_specialization_of`** trait to the standard library. Such a trait can be defined along the following lines such that it corresponds to **`true_type`** when a given type is a specialization of a given template; otherwise, it corresponds to **`false_type`** when the given type is not a specialization of that given template:

```
template< class T
        , template<class...> class Primary >
struct
  is_specialization_of : false_type;

template< template<class...> class Primary
        , class... Args >
struct
  is_specialization_of< Primary<Args...>, Primary> : true_type;

template< class T
        , template<class...> class Primary >
inline constexpr bool
  is_specialization_of_v = is_specialization_of<T, Primary>::value;
```

2.2 Expository application

Given the above primitive, we can produce [P2078R0]'s proposed **`is_complex<T>`** as follows:

```
template< class T >
using
  is_complex = is_specialization_of<T, std::complex>;

template< class T >
inline constexpr bool
  is_complex_v = is_specialization_of_v<T, std::complex>;
```

Each of the other examples listed in §1 can be produced in like manner. This frees programmers from the need to comprehend the intricacies of template template parameters (a topic often not

well-understood by non-experts). Moreover, the nature of the proposed trait is such that it can be used in a very wide variety of contexts, not just within namespace **`std`**.

3 Discussion

3.1 Naming

Q: Isn't the trait's proposed name, **`is_specialization_of`**, somewhat longer than might be considered convenient?

A: While shorter names are of course possible,[1] we prefer the proposed longer name in the interest of clarity of purpose: we believe that the longer name succinctly and correctly captures the mission of the trait and follows the naming precedent of the long-standing **`is_base_of`** trait.

3.2 Prior art

Q: Is there prior art for the proposed **`is_specialization_of`** type trait?

A: Investigation has revealed that yes, there is prior art for the proposed trait. In particular, MSVC's standard library implementation provides this exact trait. Specifically, the MSVC trait:

- is found in their **`<type_traits>`** header,
- bears the uglified name **`_Is_specialization`**,
- is implemented equivalently to the expository implementation shown above, and
- is applied in their library exactly as is shown in the above expository application.

3.3 Template aliases

Q: How does the proposed **`is_specialization_of`** type trait behave when (one or both of) its arguments are template aliases?

A: Because compilers are already required[2] to look through template aliases, the trait behaves as if the template alias argument were replaced by the underlying (i.e., aliased) entity. For example, the following code produces no diagnostic when compiled:

```
template< class > struct S  { };
template< class T > using A = S<T>;

static_assert(     is_specialization_of_v< S<int>, A> );
static_assert(     is_specialization_of_v< A<int>, A> );
static_assert( not is_specialization_of_v< double, A> );
static_assert( not is_specialization_of_v< double, S> );
```

3.4 Inheritance

Q: How does the proposed **`is_specialization_of`** type trait behave when (one or both of) its template arguments have bases?

A: The proposed trait considers only specialization. Since specialization is unrelated to inheritance,[3] the trait's result is unaffected when any template argument happens to be defined via inheritance. For example, the following code produces no diagnostic when compiled:

---

[1] Examples of such shorter names include **`is_specialization`**, **`specializes`**, **`is_spec_of`**, and the like.

[2] See [temp.alias]/2.

[3] We already have a trait, **`std::is_base_of`** (specified in [tab:meta.rel]), to detect an inheritance relationship between two types.

```
template< class > struct B  { };
template< class T > struct D : B<T>  { };

static_assert(     is_specialization_of_v< B<int>, B> );
static_assert(     is_specialization_of_v< D<int>, D> );

static_assert( not is_specialization_of_v< B<int>, D> );
static_assert( not is_specialization_of_v< D<int>, B> );

static_assert( not is_specialization_of_v< double, B> );
static_assert( not is_specialization_of_v< double, D> );
```

3.5 Inherent limitation

Q: Why does the proposed **`is_specialization_of`** type trait not compile when passed a template such as **`std::array`** or **`std::ratio`** as its second argument?

A: The trait is subject to the core language limitation that there is no syntax to declare a parameter pack composed of any but a single kind of parameter. Therefore, any template that has a non-type template parameter (e.g., **`std::array`** and **`std::ratio`**) can't today be passed as an argument where a template taking only a parameter pack of types[4] is specified. Any attempt to do so yields an ill-formed program; for example, each **`static_assert`** declaration in the following code results in a diagnostic when compiled:

```
template< class > struct S  { };

static_assert(     is_specialization_of_v< S<int>, std::ratio> );
static_assert( not is_specialization_of_v< S<int>, std::array> );
```

3.6 Lifting the inherent limitation

Q: How can the proposed **`is_specialization_of`** type trait be generalized to accept, as its second argument, a template taking any combination of type, non-type, and/or template template parameters? (As mentioned above, **`std::array`** and **`std::ratio`** are notable examples of such a template.)

A: To provide such a generalization of the proposed trait would require a significant change to the core language, namely, to allow a *universal* (i.e., any kind of) template parameter feature such as is proposed in [P1985R0].[5] Once there is a syntax to specify such a template parameter, it is merely a matter of adopting that syntax into a (future) minor revision of the present paper's proposed wording for the **`is_specialization_of`** type trait.

3.7 Timing

Q: Should we wait to adopt the proposed **`is_specialization_of`** type trait until it can be fully generalized as discussed above?

A: No, as discussed during this paper's LEWG-I review at the Prague (2020-02) WG21 meeting:

- The proposals should proceed independently, as this one is library-only while the other requires a significant core language adjustment that does not yet have proposed wording.
- Should some form of universal template parameter be ultimately adopted, it will be straightforward to adjust the trait's specification, as well as its implementation, so as to provide the desired generalization with no ABI break.

[4] Thus forced to choose, we opted to support parameter packs composed of types only, as (in our experience) templates taking only type parameters have arisen far more often than have templates taking any non-type or template template parameter.

[5] In fact, that paper puts forth, as one of its major use cases, a variation of this paper's proposed **`is_specialization_of`** trait!

- However, waiting for such a proposal's adoption, and then waiting for compilers to implement it, would needlessly delay what is a demonstrably useful facility already in use in several code bases.

3.8 Summary

For all the reasons above, we believe the **`is_specialization_of`** trait to be a worthy candidate for C++23 standardization.

4 Proposed wording[6]

4.1 After adjusting ***`yyyymm`*** (below) so as to denote this proposal's month of adoption, insert the following line among the similar directives following [version.syn]/2:

```
#define __cpp_lib_is_specialization_of yyyymmL // also in <type_traits>
```

4.2 Augment [meta.type.synop] as shown:

```
namespace std {
  ⋮
  template<class Base, class Derived>
    struct is_pointer_interconvertible_base_of;
  template<class T, template<class...> Primary>
    struct is_specialization_of;
  ⋮
  template<class Base, class Derived>
    inline constexpr bool is_pointer_interconvertible_base_of_v
      = is_pointer_interconvertible_base_of<Base, Derived>::value;
  template<class T, template<class...> Primary>
    inline constexpr bool is_specialization_of_v
      = is_specialization_of<T,Primary>::value;
  ⋮
}
```

4.3 Augment Table [tab:meta.rel] (Type relationship predicates) as shown:

Primary	Condition	Comments
⋮		
`template<class Base, class Derived> struct is_pointer_-interconvertible_base_of;`	**`Derived`** is unambiguously ...	If **`Base`** and **`Derived`** are ...
`template<class T, class<class...> Primary> struct is_specialization_of;`	**`T`** is a specialization ([temp.spec]) of **`Primary`**	
⋮		

[6] Proposed additions (there are no ~~deletions~~) are based on [N4849]. Editorial instructions and drafting notes look like this.

5 Acknowledgments

Many thanks to the readers of early drafts of this paper for their thoughtful comments.

6 Bibliography

[N4849] Richard Smith: "Working Draft, Standard for Programming Language C++." ISO/IEC JTC1/SC22/WG21 document N4849 (pre-Prague mailing), 2020–01–14. https://wg21.link/n4849.

[P1985R0] Mateusz Pusz and Gašper Ažman: "Universal Template Parameters." ISO/IEC JTC1/SC22/WG21 document P1985R0 (pre-Prague mailing), 2020–01–13. https://wg21.link/P1985R0.

[P2078R0] Bob Steagall: "Add new traits type **`std::is_complex<T>`**." ISO/IEC JTC1/SC22/WG21 document P2078R0 (pre-Prague mailing), 2020–01–13. https://wg21.link/P2078R0.

7 Document history

Rev.	Date	Changes
0	2020–02–13	• Published as P2098R0, post-Prague mailing, incorporating minor guidance from very favorable (10|7|0|0|0) LEWG-I review of a draft of this paper.
1	2020–04–10	• Added §3 (*Discussion*) and moved some existing remarks there. • Slightly adjusted §4 (*Proposed wording*). • Performed other minor editorial cleanup. • Published as P2098R1, 2020–04 mailing.