A SFINAE-Friendly `std::iterator_traits`

Document #:	WG21 N3844
Date:	2014-01-01
Revises:	None
Project:	JTC1.22.32 Programming Language C++
Reply to:	Walter E. Brown <webrown.cpp@gmail.com>

Contents

Abstract

This paper proposes to reformulate the specification of **`iterator_traits`** so as to avoid a hard error when its template argument does not have the member types expected of a non-pointer iterator, and thus to make the trait conveniently usable in a SFINAE context.

1 Introduction

The paper "**`std::result_of`** and SFINAE" (by Niebler et al.) was adopted for C++14 at the 2012 Portland meeting. It addressed "the use of **`result_of`** in contexts where SFINAE is a consideration. . . " [N3462]. More specifically, "a signature with a hard template error will poison usage of the entire overload set when it's not selected by [overload] resolution."[1]

Although that paper proposed wording for only the **`result_of`** trait, it also briefly spoke to the more general question, "Shouldn't we also address the other traits that could be made SFINAE-friendly?":

> As noted by some users and by Marc Glisse in [reflector message] c++std-lib-32994, **`result_of`** is far from the only trait that could benefit from the SFINAE treatment. **`iterator_traits`** and **`iterator_traits`** are obvious candidates. [We] have chosen to narrowly focus on **`result_of`** for lack of time.

The present paper provides wording that reformulates the specification of **`iterator_traits`** to the same purpose. Note that we have preserved full backwards compatibility in all cases where the C++14 formulation is well-formed. In particular, no use of **`iterator_traits`** in any standard library specification needs any adjustment. However, where the current formulation is ill-formed, the revised formulation is now well-formed, and has been made detectable in SFINAE contexts.

With benefit of hindsight, it has from time to time been argued that the SGI STL (and consequently C++98) erred in specifying **`iterator_traits`** as a bundle of five type aliases,[2] and

Copyright © 2014 by Walter E. Brown. All rights reserved.

[1] David Krauss, personal communication, 2013-12-30.

[2] See http://www.sgi.com/tech/stl/iterator_traits.html.

that individual iterator-related traits would have been a better design.[3] Even if true, this paper proposes no change to the basic bundled design, keeping to an all-or-nothing principle.

We first discuss a representative implementation of the reformulated trait.

2 Discussion

As shown below, our proposed reformulation of **iterator_traits** is implemented in terms of the helper template **iterator_types**, which in turn employs the helper **void_t**, which (for now) relies on a **voider** template. We will discuss each of these in subsequent subsections. We neither show nor further discuss the **iterator_traits<T*>** and **iterator_traits<T const*>** specializations, as their specifications are unaffected by our proposal.

```
namespace _ {
  template< class... >
    struct voider  { using type = void; };

  template< class... T0toN >
    using void_t = = typename voider<T0toN...>::type;

  template< class It, class = void >
    struct iterator_types
  { };
  template< class It >
    struct iterator_types<It, tytr::void_t< typename It::difference_type
                                          , typename It::value_type
                                          , typename It::pointer
                                          , typename It::reference
                                          , typename It::iterator_category
                          >               >
  {
    using difference_type   = typename It::difference_type;
    using value_type        = typename It::value_type;
    using pointer           = typename It::pointer;
    using reference         = typename It::reference;
    using iterator_category = typename It::iterator_category;
  };
}

template< class Iter >
    struct iterator_traits : _::iterator_types<Iter>  { };
```

2.1 The void_t and voider helpers

We first described a **void_t** helper in our companion paper proposing "A SFINAE-Friendly **std::common_type**" [N3843]. While the version presented here is a more general form, it has a similar design and intent and so we will repeat some of our earlier explanation.

The purpose of the **void_t** alias template is simply to map any given sequence of types to a single type, **void**. Although it seems a trivial transformation, it is nonetheless exceedingly useful, for it makes an arbitrary number of well-formed types into one completely predicable type. Consider the following example of **void_t**'s utility, a trait-like metafunction to determine whether a type **T** has a type member named **type**:

[3] In [Mad00], John Maddock and Steve Cleary made exactly this argument regarding the design of the then-new Boost type traits library. See also [N1424].

```
template< class, class = void >
  struct has_type_member : false_type  { };
template< class T >
  struct has_type_member<T, void_t<typename T::type>> : true_type  { };
```

Compared to traditional code that computes such a result, this version seems considerably simpler, and has no special cases (e.g., to avoid forming any pointer-to-reference type). The code features exactly two cases, each straightforward: (a) when there is a type member named **`type`**, the specialization is well-formed (with **`void`** as its second argument) and will be selected, producing a **`true_type`** result; (b) when there is no such type member, SFINAE will apply, the specialization will be nonviable, and the primary template will be selected instead, yielding **`false_type`**. Thus, each case obtains the appropriate result.

Incidentally, our above implementation of **`void_t`** may seem somewhat curious: since the programmer knows that the final result is **`void`**, there seems literally nothing that the compiler need do; why, then, do we involve the **`voider`** helper? Indeed, we would prefer a more straightforward formulation; contrast:

```
template<class T...> using void_t = typename voider<T...>::type;
template<class...   > using void_t = void;   // preferred
```

As we reported in our companion paper, we have encountered implementation divergence (Clang vs. GCC) while working with the preferred version shown above. We continue to conjecture that this is because of CWG issue 1558: “The treatment of unused arguments in an alias template specialization is not specified by the current wording of 14.5.7 [temp.alias].” While the issue is currently in drafting status, the notes from the CWG issues list indicate that CWG intends “to treat this case as substitution failure,” a direction entirely consistent with our intended uses. It therefore seems likely that we will at some future time be able to make portable use of our preferred simpler form. Until then, we employ our **`voider`** workaround to ensure that our template’s argument is always used.

Note that this proposal’s generalized version of **`void_t`** works with an arbitrary number of types instead of with a single type only (as was the case in our companion paper). We have found such a generalization to be useful in connection with multiple type members when an all-or-none approach is desired, as is the case in the present proposal. (While we have not yet found a use for the degenerate case of zero types, we also see no reason to forbid it.)

2.2 The `iterator_types` helper

This helper consists of a primary template and one specialization. The primary template handles cases in which its type parameter does not have all the nested types expected of a non-pointer iterator type. In turn, the specialization handles the remaining cases, i.e., those in which the iterator type does have all the expected nested types.

3 Proposed wording[4]

Augment [iterator.traits], paragraph 2, as shown:

2 The template **`iterator_traits<Iterator>`** is defined as either

[4] All proposed additions and ~~deletions~~ are relative to the post-Chicago Working Draft [N3797]. Editorial notes are displayed against a gray background.

```
namespace std {
  template<class Iterator> struct iterator_traits {
    typedef typename Iterator::difference_type   difference_type;
    typedef typename Iterator::value_type        value_type;
    typedef typename Iterator::pointer           pointer;
    typedef typename Iterator::reference         reference;
    typedef typename Iterator::iterator_category iterator_category;
 };
}
```

or as

```
namespace std {
  template<class Iterator> struct iterator_traits { };
}
```

The second form is used if and only if `Iterator` is lacking one or more of the nested types used in the first form of the template definition.

4 Feature-testing macro

For the purposes of SG10, we recommend the macro name `__cpp_lib_iterator_traits_sfinae`. This name was selected for consistency with `__cpp_lib_result_of_sfinae` as documented in [N3745].

5 Acknowledgments

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

6 Bibliography

[Mad00] John Maddock and Steve Cleary: "C++ Type Traits." Dr. Dobb's 2000-10-01. http://www.drdobbs.com/cpp/c-type-traits/184404270.

[N1424] John Maddock: "A Proposal to add Type Traits to the Standard Library." ISO/IEC JTC1/SC22/WG21 document N1424 (pre-Oxford mailing), 2003-03-03. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1424.htm.

[N3436] Eric Niebler, Daniel Walker, and Joel de Guzman: "`std::result_of` and SFINAE." ISO/IEC JTC1/SC22/WG21 document N3436 (pre-Portland mailing), 2012-09-21. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3436.html.

[N3462] Eric Niebler, Daniel Walker, and Joel de Guzman: "`std::result_of` and SFINAE." ISO/IEC JTC1/SC22/WG21 document N3462 (post-Portland mailing), 2012-10-18. A revision of [N3436]. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3462.html.

[N3745] Clark Nelson: "Feature-testing recommendations for C++." ISO/IEC JTC1/SC22/WG21 document N3745 (pre-Chicago mailing), 2013-08-28. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3745.htm.

[N3797] Stefanus Du Toit: "Working Draft, Standard for Programming Language C++." ISO/IEC JTC1/SC22/WG21 document N3797 (post-Chicago mailing), 2013-10-13. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3797.pdf.

[N3843] Walter E. Brown: "A SFINAE-Friendly `std::common_type`." ISO/IEC JTC1/SC22/WG21 document N3843 (pre-Issaquah mailing), 2013-11-30. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3843.pdf.

7 Document history

Version	Date	Changes
1	2014-01-01	• Published as N3844.