A SFINAE-Friendly `std::common_type`

Document #: WG21 N3843
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 the `common_type` trait so as to avoid a hard error when there is no common type, 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 the Niebler 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 `common_type` 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 `common_type`[2] to the same purpose. Note that we have preserved full backwards compatibility in all cases where the C++14 formulation is well-formed, modulo a subtlety (discussed below) in the application of LWG issue 2141. In particular, no use of `common_type` in any standard library specification (such as those throughout `<chrono>`) 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.[3]

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

[2] The `common_type` trait was first proposed for C++0x by Hinnant et al. in [N2615] and adopted at the 2008 Sophia-Antipolis meeting via the revised paper [N2661].

[3] In reflector message c++std-lib-33011, Howard Hinnant terms the underlying technique a *Voldemort* ***type***. ☺

We first discuss a representative implementation of the reformulated trait. Since the C++11 and C++14 **common_type** traits are specified in terms of code, the proposed wording will then substantially excerpt this implementation so as to remain consistent in specification style. We also provide, for LWG's consideration, alternative proposed wording via prose.

2 Discussion

As shown below, our proposed reformulation of **common_type** is implemented in terms of the helper template **ct**, which in turn employs helpers **ct2** and **void_t**. We will discuss each of these helpers in subsequent subsections; thereafter, our proposed wording will treat the helpers as *exposition-only*.

```
namespace _ {
  template< class T, class U >
    using ct2 = decltype(declval<bool>() ? declval<T>() : declval<U>());

  template< class T >
    using void_t = conditional_t<true,void,T>;

  template< class, class... >               // no types, or no common type
    struct ct  { };
  template< class T >                       // single type
    struct ct<void,T>  { using type = decay_t<T>; };
  template< class T, class U, class... V >  // two or more types
    struct ct<void_t<ct2<T,U>>, T, U, V...>
    : ct<void, ct2<T,U>, V...>  { };
}

template< class... T >
  struct common_type : _::ct<void, T...>  { };
```

2.1 The ct2 helper

This alias template encapsulates the basic mechanism for determining the type, if any, common to two given types **T** and **U**. While, strictly speaking, the body of this template could be embedded where it is used, we find that factoring this mechanism into an alias template makes it both clearer to understand and more convenient to use.

Note that we use **declval<bool>()** instead of **true**, as currently formulated. We do so because, in an unevaluated context such as the above **decltype**, any specific **bool** value is irrelevant; we could equally correctly have specified **false**. Absent a reason to prefer one over the other, we decline an unnecessary choice. (Implementers are, of course, free under the as-if rule to choose differently.)

It is important to observe that a specialization of the **ct2** template is ill-formed whenever its conditional expression is ill-formed, i.e., when **T** and **U** have no type in common. This insight explains why the C++11 and C++14 formulation leads to ill-formed programs under the same circumstances.

2.2 The void_t helper

The purpose of the **void_t** alias template is simply to map any given type to **void**. Although a trivial transformation, it is nonetheless exceedingly useful, for it makes an arbitrary well-formed type into one that is completely predicable. 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**:

```
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.

Indeed, we can use the above **`has_type_member`** example in conjunction with our proposed reformulation of **`common_type`**:

```
has_type_member< common_type<int,double> >{}()  // true
has_type_member< common_type<int,string> >{}()  // false
```

Note that the C++11 and C++14 versions of **`common_type`** are not usable in this way, as (a) they produce a hard error whenever there is no common type, and therefore (b) the SFINAE inside **`has_type_member`** can't apply.

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 **`conditional_t`**? Indeed, we would prefer a more straightforward formulation; contrast:

```
template<class T> using void_t = conditional_t<true,void,T>;  // as above
template<class  > using void_t = void;                        // preferred
```

Alas, we have as of this writing encountered implementation divergence (Clang vs. GCC) while working with the preferred version shown above, probably 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 **`conditional_t`** as a workaround to ensure that our template's argument is always used.

Finally, let us mention for completeness that we have experimented with a more general version of **`void_t`**. Instead of a single type only. our expanded version takes a parameter pack of types as its template parameter. While not needed for the present purpose, such a generalization seems useful in connection with multiple type members when an all-or-none approach is desired. See our companion paper [N3844] for an application of just such a generalized **`void_t`**.

2.3 The `ct` helper

This helper consists of a primary template and two specializations. The primary template handles all cases where there is no common type. Such cases typically arise when, in a given list of types, there are two adjacent types such that (a) neither can be converted to any form of the other, or (b) there is no third type to which both can be converted. This case also arises when the given list of types is empty. This primary template thus ensures that there is now a well-defined result (albeit a vacuous one[4]) in such circumstances.

[4]Note that the D programming language made a different design decision for its analogous *CommonType* template: "Returns *void* if passed an empty list, or if the types have no common type" [http://dlang.org/phobos/std_traits.html]. Such a choice would defeat this paper's principal purpose: we would be unable, for example, to distinguish cases having no common type from even the trivial-yet-valid **`common_type_t<void,void>`**.

The first specialization is designed to handle the case of a single type. Moreover, it is here that we apply our understanding of the intent underlying LWG issue 2141's resolution in C++14, namely to apply the **`decay`** trait to the result type as previously specified in C++11. While the resolution technically calls for a decay at each processing step, we believe this was an unintended artifact of the resolution's specification.[5] As documented by the outcomes of reflector threads starting with messages c++std-lib-32298 and c++std-lib-33104, we believe that LWG intended a single, final, decay step, as that completely addresses the issue as raised. We drafted this specialization accordingly.

Finally, the second specialization handles the general case of two or more types. Note that this specialization will be viable only so long as the first two types in the argument list of types have a common type as defined by the above-described **`ct2`** helper. If at any point the first two types have no common type, both specializations will be nonviable and so the primary template will be selected to terminate the recursion. Otherwise, once the list has been recursively pairwise reduced to a single type, the first specialization will be selected and again we will achieve a result consistent with LWG 2141.

3 Proposed wording[6]

3.1 Common wording

There are two alternative wording proposals (marked below as Alternative 1 and Alternative 2), but both encompass the following wording changes:

Change, as shown, the entry in row **`common_type`**, column "Comments" of **Table 57 — Other transformations**:

The member typedef `type`, if any, shall be defined as ~~set out~~ specified below; otherwise there shall be no member `type`. All types in the parameter pack `T` shall be complete or (possibly cv) `void`. A program may specialize this trait if at least one template parameter in the specialization is a user-defined type. [*Note:* Such specializations are needed only when ~~only~~ explicit conversions are desired ~~among the~~ between two consecutive template arguments. — *end note*]

3.2 Alternative 1

Replace paragraph 3 of subclause **[meta.trans.other]** (20.10.7.6) with the following mostly-code specification. (This wording follows the presentation style of this trait's C++11 and C++14 versions.)

3 For the **`common_type`** trait, the member **`type`** shall be either defined or not present according to the following specification in terms of the exposition-only templates **`ct2`**, **`void_t`**, and **`ct`**:

[5] We mention this because there appear to be some subtle corner cases, apparently not previously fully explored in the resolution's context, in which the results may differ.

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

```
template <class T, class U>
using ct2 = decltype(declval<bool>() ? declval<T>() : declval<U>());

template <class T> using void_t = conditional_t<true,void,T>;

template <class, class...> struct ct {};
template <class T> struct ct<void,T> { using type = decay_t<T>; };
template <class T, class U, class... V>
struct ct<void_t<ct2<T,U>>, T, U, V...> : ct<void, ct2<T,U>, V...> {};

template <class... T> struct common_type : ct<void, T...> {};
```

3.3 Alternative 2

Replace paragraph 3 of subclause **[meta.trans.other]** (20.10.7.6) with the following prose specification.

3 For the **`common_type`** trait applied to a parameter pack **`T`** of types, the member **`type`** shall be either defined or not present as follows:

- If **`sizeof...(T)`** is zero, there shall be no member **`type`**.
- If **`sizeof...(T)`** is one, let **`T0`** denote the sole type comprising **`T`**. The member typedef **`type`** shall denote the same type as **`decay_t<T0>`**.
- If **`sizeof...(T)`** is greater than one, let **`T1`**, **`T2`**, and **`R`** respectively denote the first, second, and (pack of) remaining types comprising **`T`**. [*Note:* **`sizeof...(R)`** may be zero. — *end note*] Finally, let **`C`** denote the type, if any, of an unevaluated conditional expression ([expr.cond]) whose first operand is an arbitrary value of type **`bool`**, whose second operand is an xvalue of type **`T1`**, and whose third operand is an xvalue of type **`T2`**. If there is such a type **`C`**, the member typedef **`type`** shall denote the same type, if any, as **`common_type_t<C,R...>`**. Otherwise, there shall be no member **`type`**.

3.4 Feature-testing macro

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

4 Acknowledgments

Many thanks to the readers of early drafts of this paper for their thoughtful comments. Special thanks to Stephan T. Lavavej for his extensive experimentation and consequent elegant contributions to the final form of the trait's implementation, to Daniel Krügler for pointing out CWG issue 1558, and to Jens Maurer for inspiring the prose form of the proposed wording.

5 Bibliography

[N2615] Howard E. Hinnant, Walter E. Brown, Jeff Garland, and Marc Paterno: "A Foundation to Sleep On." ISO/IEC JTC1/SC22/WG21 document N2615 (pre-Sophia-Antipolis mailing), 2008-05-18. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2615.html.

[N2661] Howard E. Hinnant, Walter E. Brown, Jeff Garland, and Marc Paterno: "A Foundation to Sleep On." ISO/IEC JTC1/SC22/WG21 document N2661 (post-Sophia-Antipolis mailing), 2008-06-11. A revision of [N2615]. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2661.html.

[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.

[N3844] Walter E. Brown: "A SFINAE-Friendly **`std::iterator_traits`**." ISO/IEC JTC1/SC22/WG21 document N3844 (pre-Issaquah mailing), 2014-01-01. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3844.pdf.

6 Document history

Version	Date	Changes
1	2014-01-01	• Published as N3843.