# Resolving LWG Issues re **common_type**

*Document #:* WG21 P0435R0
*Date:* 2016-10-14
*Project:* JTC1.22.32 Programming Language C++
*Audience:* LWG
*Reply to:* Walter E. Brown <webrown.cpp@gmail.com>

## Contents



**Abstract**

LWG issues 2465 and 2763 have seen considerable recent churn. Experimentation has revealed that these Issues' latest Proposed Resolutions do not pass all expected tests. This paper presents revised wording as well as a corresponding implementation that does pass the tests of desired behavior. This wording also addresses the first part of LWG issue 2460 as well as several other concerns, and takes a step toward addressing a recent renewed request for a "code-based definition."

*The longer we listen to one another — with real attention — the more commonality we will find in all our lives. That is, if we are careful to exchange with one another life stories and not simply opinions.*

— BARBARA DEMING

*On dit quelquefois: «Le sens commun est fort rare.»*
*[People sometimes say: "Common sense is quite rare."]*

— VOLTAIRE, *né* FRANÇOIS-MARIE AROUET

## 1 Introduction: what's wrong?

Among the metafunctions in **<type_traits>**, **common_type** is unique: it is the only one that programmers may specialize. However, programmers should not have to provide specializations for all combinations of cv-qualifications and reference qualifications. Alas, correct specification of this part of the design has proven to be exceptionally elusive, as evidenced by the recent churn in addressing LWG 2465 as well as the recently-added LWG 2763. Judicious use of the **decay** trait has materially improved the specification, but the timing of its application has not always been clear.

At the time of writing, there are some problems with the cited issues' latest Proposed Resolutions. In addition to several wording nits, there are two issues of vital substance:

- LWG 2763 presents a "merged" resolution with LWG 2465, but it is incomplete in that it omits 2465's necessary changes to **common_type**'s table entry.
- LWG 2465 observes that "the current P/R no longer decays the type of the conditional expression," but incorrectly describes that lack as "harmless."

The cumulative effect of the above seems fatal to the Proposed Resolutions as written in the LWG Issues List [LWG]. In particular, experimentation has revealed that implementations conforming to those specifications do not exhibit intended behavior: they fail several of the relevant conformance tests supplied with libc++, for example.

In addition to the above issues, the following concerns were privately pointed out:

- [namespace.std]/1 already contains a blanket provision that restricts what specializations a program may write in namespace **`std`**:

  > A program may add a template specialization for any standard library template to namespace **`std`** only if the declaration depends on a user-defined type. . . .

  Therefore, the **`common_type`** specification can be slightly simplified by appealing to those restrictions.

- The requirement on applying the **`common_type`** trait omits the possibility of an unbounded array type. Other multi-parameter traits have no such restriction, and there seems no reason to enforce it here. We have opted to treat this somewhat more generally to avoid the possibility of an ODR violation such as the following:

```
struct A; struct B;  // incomplete types
··· common_type_t<A*, B*> ···  // no member 'type' => ill-formed

struct A { }; struct B : A { };  // now complete types
··· common_type_t<A*, B*> ···  // now well-formed => ODR violation
```

- **`common_type`** specializations have no required semantics. Given our understanding of **`common_type`**'s original intent and of its behavior when applied to fundamental types, we believe it reasonable to require (a) that each of the argument types in such a specialization be explicitly convertible to the common type, and (b) that the common type of **`T`** and **`U`** always denote the same type as the common type of **`U`** and **`T`**.

While such concerns are not part of the LWG issues under discussion, we have nonetheless addressed them herein. Therefore, for all these reasons, this paper's proposed wording (§3), which makes the appropriate adjustments to the wording from the Issues List, is recommended.

## 2 Expository implementation

This section presents an implementation that conforms to the specifications proposed in §3. It has successfully passed all relevant parts of the libc++ tests and of proprietary tests. Namespaces have been omitted here for clarity of exposition; the unedited code was compiled using GCC version 7.0.0 20160807 with significant options **`-std=c++1z`** and **`-fconcepts`**.[1]

### 2.1 Exposition-only helpers

```
// result type of conditional operator:
template< class T, class U >
using
  cond_t = decltype( false ? declval<T>() : declval<U>() );

```

[1] See http://melpon.org/wandbox/permlink/Y0E3NFXYO66Y7zOK for an alternate implementation, by Tomasz Kamiński, that employs only C++14 technology. In particular, nested **`enable_if`**s, combined with the detection idiom, replace *requires-clauses* to select among the code segments corresponding to the various bullets of the wording proposed in §3. However, in private correspondence granting permission to share the link, Kamiński commented that the present paper's "current implementation. . . is a lot cleaner in expressing intent."

```
// verify that neither type needs further decay:
template< class T, class U >
constexpr bool
  are_already_decayed_v = is_same_v< T, decay_t<T> >
                      and is_same_v< U, decay_t<U> >;
```

## 2.2 Declarations per [meta.type.synop]

```
// other transformation trait:
template< class... >
struct
  common_type;

// result alias:
template< class... Ts >
using
  common_type_t = typename common_type<Ts...>::type;
```

## 2.3 Per-bullet definitions[2]

```
// (3.1):
template< class... >
struct
  common_type  { };

// (3.2):
template< class T >
struct
  common_type<T0> : decay<T0>  { };

// (3.3), case (3.3.1):
template< class T1, class T2 >
  requires not are_already_decayed_v<T1,T2>
struct
  common_type<T1,T2> : common_type< decay_t<T1>, decay_t<T2> >  { };

// (3.3), case (3.3.2):
template< class T1, class T2 >
  requires are_already_decayed_v<T1,T2>
       and requires { typename cond_t<T1,T2>; }
struct
  common_type<T1,T2> : decay< cond_t<T1,T2> >  { };

// (3.4):
template< class T1, class T2, class... R >
  requires sizeof...(R) > 0
       and requires { typename common_type_t<T1,T2>; }
struct
  common_type<T1,T2,R...> : common_type< common_type_t<T1,T2>, R... >  { };
```

[2] Comments identify the corresponding numbered bullets and subbullets in §3.

## 3 Proposed wording[3]

The following wording is intended (a) to resolve LWG Issues 2465, 2763, and (the first part of) 2460, and (b) to address the other concerns raised in §1.

**3.1** Edit the entry for **`common_type`** in Table 46 — "Other transformations" as shown below. Note that the sentences deleted here will reappear (with significant adjustments) in the new Note B, below.

Unless this trait is specialized (as specified in Note B, below), ~~T~~the member ~~typedef~~ **`type`** shall be defined or omitted as specified in Note A, below. If it is omitted, there shall be no member **`type`**. ~~All types~~Each type in the parameter pack **`T`** shall ~~be~~not depend on any incomplete type ~~or~~other than on (possibly cv-qualified) **`void`** or on an array of unknown bound. ~~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 when only explicit conversions are desired among the template arguments. — end note]~~

**3.2** Edit 20.15.7.6 [meta.trans.other]/3 (and its subbullets) as shown below.

3 Note A: 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:

(3.1) — If **`sizeof...(T)`** is zero, there shall be no member **`type`**.

(3.2) — If **`sizeof...(T)`** is one, let **`T0`** denote the sole type ~~in~~comprising the pack **`T`**. The member ~~typedef~~*typedef-name* **`type`** shall denote the same type as **`decay_t<T0>`**.

(3.3) — If **`sizeof...(T)`** is two, let the first and second types comprising **`T`** be denoted by **`T1`** and **`T2`**, respectively, and let **`D1`** and **`D2`** denote the same types as **`decay_t<T1>`** and **`decay_t<T2>`**, respectively.

(3.3.1) — If **`is_same_v<T1, D1>`** is **`false`** or **`is_same_v<T2, D2>`** is **`false`**, let **`C`** denote the same type, if any, as **`common_type_t<D1, D2>`**.

(3.3.2) — Otherwise, let **`C`** denote the same type, if any, as **`decay_t< decltype(false ? declval<D1>() : declval<D2>())>`**. [Note: This will not apply if there is a specialization **`common_type<D1, D2>`**. — end note]

In either case, the member *typedef-name* **`type`** shall denote the same type, if any, as **`C`**. Otherwise, there shall be no member **`type`**. [Note: When **`is_same_v<T1, T2>`** is **`true`**, the effect is equivalent to that of **`common_type<T1>`**. — end note]

(3.4) — If **`sizeof...(T)`** is greater than ~~one~~two, let **`T1`**, **`T2`**, and **`R`**, respectively, denote the first, second, and (pack of) remaining types comprising **`T`**. ~~[Note: . . . — end note] Let . . . whose first operand is . . . , whose second operand is . . . , and whose third operand is . . . .~~ Let **`C`** denote the same type, if any, as **`common_type_t<T1, T2>`**. If there is such a type **`C`**, the member ~~typedef~~*typedef-name* **`type`** shall denote the same type, if any, as **`common_type_t<C, R...>`**. Otherwise, there shall be no member **`type`**.

**3.3** Insert the following new paragraph immediately after the last bullet of the above Note A paragraph, and renumber subsequent paragraphs accordingly. Note that this text was initially relocated here after its excision from Table 46.

[3] All proposed additions and ~~deletions~~ are relative to the post-Oulu Working Draft [N4606]. Editorial instructions and drafting notes are displayed against a gray background.

4 Note B: Notwithstanding the provisions of [meta.type.synop], and pursuant to [namespace.std], ~~A~~a program may specialize ~~this trait if at least one template parameter in the specialization is a user-defined type~~**`common_type<T1, T2>`** for distinct types **`T1`** and **`T2`** such that **`is_same_v<T1, decay_t<T1>>`** and **`is_same_v<T2, decay_t<T2>>`** are each **`true`**. [Note: Such specializations are needed when only explicit conversions are desired ~~among~~between the template arguments. — end note] Such a specialization need not have a member named **`type`**, but if it does, that member shall be a *typedef-name* for an accessible and unambiguous cv-unqualified non-reference type **`C`** to which each of the types **`T1`** and **`T2`** is explicitly convertible. Moreover, **`common_type_t<T1, T2>`** shall denote the same type, if any, as does **`common_type_t<T2, T1>`**. No diagnostic is required for a violation of this Note's rules.

## 4 Acknowledgments

Special thanks to Alisdair Meredith, Casey Carter, Eric Niebler, Howard Hinnant, Marshall Clow, Nevin Liber, Stephan T. Lavavej, and Tomasz Kamiński for productive discussions regarding numerous subtleties of this surprisingly difficult-to-specify component.

## 5 Bibliography

[LWG] Marshall Clow, et al.; "C++ Standard Library Active Issues List (Revision D100)." Retrieved 2016-08-14. Online: http://cplusplus.github.io/LWG/lwg-active.html.

[N4606] Richard Smith: "Working Draft, Standard for Programming Language C++," ISO/IEC JTC1/SC22/WG21 document N4606 (post-Oulu mailing), 2016-07-12.
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/n4606.pdf.
Same content as "C++17 CD Ballot Document," ISO/IEC JTC1/SC22/WG21 document N4604 (post-Oulu mailing), http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/n4604.pdf.

## 6 Document history

| Version | Date | Changes |
|---|---|---|
| 0 | 2016-10-14 | • Published as P0435R0. |