Abstract

Based on unexpected ripple effects of our recently adopted paper [P0435R1], this paper proposes a few small tweaks to the wording for (a) common_type and (b) duration.

Duration is not a test of truth or falsehood.
— ANNE MORROW LINDBERGH

When you can do the common things of life in an uncommon way, you will command the attention of the world.
— GEORGE WASHINGTON CARVER

1 Introduction

Our paper [P0435R1] was recently adopted for C++17 in order to address LWG issues 2465, 2763, 2460 (in part), and “several other concerns.” Certain ripple effects of that paper’s changes have since been brought to our attention. The present paper will explicate those effects and propose a few small tweaks to the wording for (a) common_type and (b) duration.

2 Tweaking common_type

We recently received¹ the following correspondence (lightly reformatted), pointing out a ripple effect of our earlier paper’s adoption:

The new Note B says that common_type<T1, T2> specializations are allowed only if T1 and T2 are distinct types . . . .

If so, that would seemingly outlaw partial specializations like the ones done for chrono::duration, since those will be used even for identical duration types like common_type_t<chrono::seconds, chrono::seconds>[].

¹Tim Song: “Specializing our favorite type trait, again.” Personal communication, 2016–12–27.
This is certainly an unexpected consequence, and one that should be promptly addressed by a change either to the specification of common_type or to the specification of its duration specialization.

That specialization is currently quite simply and elegantly specified in [time.traits.specializations] (reflowed below to fit available space):

```cpp
template <class Rep1, class Period1, class Rep2, class Period2>
struct common_type< chrono::duration<Rep1, Period1>, chrono::duration<Rep2, Period2> >
{
    using type = chrono::duration<common_type_t<Rep1, Rep2>, see below>;}
```

We would like to preserve this and similar specializations. We therefore restrict ourselves to consider how best to adjust common_type to allow this.

On the one hand, it would certainly be easy enough to strike “distinct” from the common_type specification, and thereby again permit partial specializations for chrono::duration, etc. On the other hand, doing so removes what we believe to be an important guarantee regarding common_type’s behavior, namely, that common_type_t<T,T> and common_type_t<T> always denote the same type. (There is even a Note pointing out that “When is_same_v<T1,T2> is true, the effect is equivalent to that of common_type<T1>.”)

We therefore propose the following adjustments to common_type’s specification: (a) to strike “distinct” as discussed above, (b) to redefine the result of common_type’s single-argument case as having common_type_t<T0,T0> (rather than the current decay_t<T0>), and (c) to strike the now-redundant Note cited above.

We note that the net effect of these changes may impact existing code, but only in contrived scenarios that seem unlikely to arise in practice. Here are three examples:

```cpp
using non_reduced_seconds = duration< int, ratio<10, 10> >;
static_assert( is_same_v< common_type_t<non_reduced_seconds>, non_reduced_seconds >,
    "no longer holds");
```

```cpp
struct A { };
template< typename T > struct common_type<A, T> { }; // no common type
... common_type_t<A> ... // previously ignores, now uses the above specialization
```

```cpp
struct B { };
struct common_type<B, B> { using type = common_type_t<B>; }; // now recursive
```

3 Tweaking duration

While examining duration’s use of common_type’s specializations as discussed above, we noted what appeared to be a minor inconsistency between duration’s unary + and - operators and their corresponding binary operators. The relevant - operators are today specified as follows (+ is nearly identical):
template< class Rep, class Period = ratio<1> >
class duration {
    ... 
    constexpr duration operator-() const;
    ...
};

template< class Rep1, class Period1, class Rep2, class Period2 >
common_type_t< duration<Rep1, Period1>, duration<Rep2, Period2> >
constexpr operator-( const duration<Rep1, Period1>& lhs
    , const duration<Rep2, Period2>& rhs
);

Note especially these declarations’ return types. The unary member function returns the type of
which it is a member, while the binary free function returns the common_type_t of its arguments’
types.

Now consider the following example:

using D = duration<int, ratio<10,10>>;
D a(0);
D b(1);
static_assert( is_same_v< decltype(-b), D > );
static_assert( is_same_v< decltype(a-b), duration<int,ratio<1,1>> > );
static_assert( not is_same_v< decltype(-b), decltype(a-b) > );

It seems a bit surprising that all the assertions hold true. In particular:

• If binary minus may implicitly reduce the ratio, shouldn’t unary minus do the same?
• Conversely, if unary minus never reduces the ratio, shouldn’t binary minus behave likewise?
• More simply stated, when a is zero (and at all other times), shouldn’t the type of a–b be the
  same as the type of –b?

We believe that the answer to this last question should be yes, the types ought always be the
same.

How common are such occurrences? <chrono> expert Howard Hinnant reported\(^2\) that “Non-
reduced duration types are an anomaly that may or may not exist in the wild at this point. I
haven’t seen one, so at most they are not common.” He further opined that “we should also be
conservative and not ... propagate more of them into existing code.”

Accordingly, we propose to change the return types for duration’s unary operator+ and
operator– members such that reduced duration types may be produced, as is the case for the
 corresponding binary operators. Given the changes already proposed above for common_type, the
change here becomes very simple: merely replace the operators’ return type, currently duration,
by common_type_t<duration>.\(^3\)

Further, it would be beneficial to indicate more explicitly that non-reduced duration types
are discouraged. They are not propagated via the binary operators, and with the above change
are no longer routinely propagated via the unary operators. To close the door a bit further, we
additionally recommend that duration’s nested type alias period be adjusted so as to alias
Period::type (the reduced ratio) rather than Period (the non-reduced ratio).\(^4\)


\(^3\)Note that this change does not introduce any additional conversion operations. Thus there are no new opportunities
for rounding or other error even if the underlying representation were a floating-point type, for example.

\(^4\)We intend, in a future paper, to explore the possibility of unifying all non-reduced ratio types with their respective
reduced equivalents. While such an approach seems promising, it is far beyond the scope of the present proposal.
4 Proposed wording

4.1 Tweaking common_type

4.1.1 Adjust [meta.trans.other]/3.2 as shown:

If sizeof...(T) is one, let T0 denote the sole type constituting the pack T. The member typedef-name type shall denote the same type, if any, as decay_t<T0> common_type_t<T0,T0>; otherwise there shall be no member type.

4.1.2 Strike, in its entirety, the following Note that concludes [meta.trans.other]/3:

.... [Note: When is_same_v<T1,T2> is true, the effect is equivalent to that of common_type<T1>.
— end note]

4.1.3 Adjust [meta.trans.other]/4 as shown:

... a program may specialize common_type<T1,T2> for distinct types T1 and T2 such that ....

4.2 Tweaking duration

4.2.1 Adjust an alias and two return types in the synopsis following [time.duration]/1 as shown:

... public:
  using rep = Rep;
  using period = typename Period::type;

private:
  ...
  // 20.17.5.3, arithmetic
  constexpr common_type_t<duration> operator+() const;
  constexpr common_type_t<duration> operator-() const;
  ...

4.2.2 Adjust the return types in [time.duration.arithmetic]/1-2 as shown:

  constexpr common_type_t<duration> operator+() const;

1 Returns: common_type_t<duration>(*this).

  constexpr common_type_t<duration> operator-() const;

2 Returns: common_type_t<duration>(-rep_).

5 All proposed additions and deletions are relative to the post-Issaquah Working Draft [N4618]. Editorial notes are displayed against a gray background.
5 Acknowledgments

Tim Song first brought these matters to our attention, Howard Hinnant was instrumental in devising an acceptable strategy to address them, and they, Casey Carter, and Andrey Semashev served as reviewers of pre-publication drafts. Thank you, gentlemen, for your contributions.

6 Bibliography


7 Document history

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2017-02-01</td>
<td>• Published as P0548R0.</td>
</tr>
<tr>
<td>0</td>
<td>2017-03-03</td>
<td>• Adjusted proposed return types per LWG guidance. • Published as P0548R1.</td>
</tr>
</tbody>
</table>