Treating Unnecessary `decay`

Document #: WG21 P0777R0
Date: 2017-10-10
Project: JTC1.22.32 Programming Language C++
Audience: LWG
Reply to: Walter E. Brown <webrown.cpp@gmail.com>

Contents

Abstract

We report the results of a Standard Library audit, inspecting each application of the `decay` trait. We find a few occurrences that can be replaced (usually by the recently-approved `remove_cvref` trait) to provide more precise specifications, and recommend wording changes to that effect.

Woe, destruction, ruin, and decay . . . .

— William Shakespeare

Whenever a people or an institution forget its hard beginnings, it is beginning to decay.

— Carl Sandburg

Great effort is required to arrest decay and restore vigor.

— Horace

1 Introduction

[P0550R2] points out that the Standard Library's specifications make more use of `decay` than strictly necessary, i.e., in "places where it would suffice simply to strip *cv* and reference qualifiers." It seems likely that such overuse resulted from several overlapping factors:

- There was no trait exactly corresponding to `remove_const_t<remove_reference_t<>>`.
- It is rather inconvenient to type `remove_const_t<remove_reference_t<>>`.
- It is easy to remove the `const` and reference qualifications in the wrong order, a mental error likely leading to incorrect program behavior.
- The `decay` trait gets the order right, is reasonably convenient to type, and seems (at first glance) to do no harm except for doing more work than strictly necessary.

However, there is harm in such overuse, as succinctly described by Peter Dimov: "When you read code and see `decay_t`, you don't know whether `decay_t` has been used because decay semantics are needed, or because the intent was to merely remove references and *cv*-qualifiers." He continued, "It can be technically correct, as in it'll yield the right answer, but it's still wrong

because it's the wrong word." In Toronto (July 2017), LEWG found such arguments persuasive and, as proposed in [P0550R2], approved a new **`remove_cvref`** trait.

Dimov's observation applies not only to programs, but also to such technical documents as the Standard Library's specification. Accordingly, it seems appropriate to undertake the next step envisioned in [P0550R2]: "In all, the Library clauses directly apply **`decay_t`** circa forty times; we recommend that each be audited" for possible replacement by the new trait or by the even simpler **`remove_reference`**.

We have performed such an audit, inspecting each context in which the standard library applies the **`decay`** trait. In the next section, we present our detailed findings and propose wording to implement our recommendations.

2 Findings and proposed wording[1]

As a rule of thumb, it seems that evaluating a relationship between two types (e.g., via **`is_same`** or **`is_base_of`**) is the major context that typically does not require full **`decay`** semantics. It is certainly useful, in such cases, for the types being compared to be *cv*- and reference-unqualified. However, additionally transforming array and function types into pointer types seems to have no bearing on the outcome of the cases we assessed.

2.1 `make_index_sequence<tuple_size_v<decay_t<Tuple>>>`

We find this expression in each paragraph of [tuple.apply]. We see no need for decaying behavior here; simply stripping qualifiers seems sufficient. Moreover, since **`tuple_size`** is defined (in [tuple.apply]/4) for **`const`**-qualified types, it suffices to remove only a reference qualifier, if any. We therefore recommend **`remove_reference`** as a suitable replacement for each of these uses of **`decay`**:

Edit [tuple.apply]/1 and [tuple.apply]/2 as shown:

```
make_index_sequence<tuple_size_v<decay_tremove_reference_t<Tuple>>> ···
```

Further, since the surrounding context seems to impose no requirements on the **`Tuple`** type, we additionally recommend that LWG consider adding a requirement that **`Tuple`** must be a **`tuple`**-like type.

2.2 `is_same_v<decay_t<U>, in_place_t>`

We find this expression in [optional.ctor]/23. We see no need for decaying behavior here; simply stripping qualifiers seems sufficient. We recommend **`remove_cvref`** as a replacement for this use of **`decay`**:

Edit [optional.ctor]/23 as shown:

```
··· is_same_v<decay_tremove_cvref_t<U>, in_place_t> ···
```

2.3 `is_same_v<optional<T>, decay_t<U>>`

We find this expression in [optional.ctor]/23 and again in [optional.assign]/16. We see no need for decaying behavior here; simply stripping qualifiers seems sufficient. We recommend **`remove_cvref`** as a replacement for each of these uses of **`decay`**:

Edit [optional.ctor]/23 and [optional.assign]/16 as shown:

```
··· is_same_v<optional<T>, decay_tremove_cvref_t<U>> ···
```

[1] Proposed wording changes are provided in the form of editorial notes, displayed against a gray background, containing embedded markup to denote intended additions and ~~deletions~~. All edits are relative to the post-Toronto Working Draft [N4687].

Further, in each of these paragraphs, we additionally recommend that the Project Editor consider also replacing **`optional<T>`** by the equivalent, slightly simpler, *injected-class-name* **`optional`**.

2.4 `is_same<T, decay_t<U>>`

We find this expression in [optional.assign]/16. We see no need for decaying behavior here; simply stripping qualifiers seems sufficient. We recommend **`remove_cvref`** as a replacement for this use of **`decay`**:

Edit [optional.assign]/16 as shown:

··· **`conjunction_v<is_scalar<T>, is_same<T, ~~decay_t~~remove_cvref_t<U>>>`** ···

2.5 `is_same_v<decay_t<T>, variant>`

We find this expression in [variant.ctor]/16 and again in [variant.assign]/14. We see no need for decaying behavior here; simply stripping qualifiers seems sufficient. We recommend **`remove_cvref`** as a replacement for these uses of **`decay`**:

Edit [variant.ctor]/16 and [variant.assign]/14 as shown:

··· **`is_same_v<~~decay_t~~remove_cvref_t<T>, variant>`** ···

2.6 `decay_t<T>` is neither ... nor ...

We find this expression in [variant.ctor]/16. The result of **`decay_t`** is subsequently compared against specializations of certain templates. We see no need for decaying behavior here; simply stripping qualifiers seems sufficient. We recommend **`remove_cvref`** as a replacement for these uses of **`decay`**:

Edit [variant.ctor]/16 as shown:

··· **`~~decay_t~~remove_cvref_t<T>`** **is neither** ···

2.7 `decay_t<decltype(`t_1`)>`

We find this expression four times in [func.require], once in each of four bullets contributing to the definition of the *INVOKE* pseudo-function: In two of the cases, the context asks whether we have "a specialization of **`reference_wrapper`**" on our hands; in the other two cases, the context asks whether we have an inheritance relationship to another type. In none of these do we see any need for decaying behavior; simply stripping qualifiers seems sufficient. We recommend **`remove_cvref`** as a replacement for these uses of **`decay`**:

Edit [func.require]/1 (bullets 1.1, 1.2, 1.4, and 1.5) as shown:

··· **`~~decay_t~~remove_cvref_t<decltype(`**t_1**`)>`** ···

3 Acknowledgment

Many thanks to Peter Dimov for his thoughtful comments regarding the application of the **`remove_cvref`** trait.

4 Bibliography

[N4687] Richard Smith: "Working Draft, Standard for Programming Language C++." ISO/IEC JTC1/SC22/WG21 document N4687 (post-Toronto mailing), 2017–07–30. http://wg21.link/n4687.

[P0550R2] Walter E. Brown: "Transformation Trait **`remove_cvref`**." ISO/IEC JTC1/SC22/WG21 document P0550R2 (post-Toronto mailing), 2017–07–17. http://wg21.link/p0550r2.

5 Document history

Rev.	Date	Changes
0	2017-10-10	• Published as P0777R0, pre-Albuquerque.