Adjuncts to std::hash

1 Introduction

Lippincott’s paper [P0513R0], adopted\(^1\) for C++17 in Issaquah, introduced new vocabulary to describe specializations of std::hash. Each is now “either disabled (‘poisoned’) or enabled (‘untainted’).”\(^2\)

The paper also suggested “a standard trait hash_enabled<T>.” No such trait was formally proposed, however, because WG21 was at the time focussed on ballot resolution and other C++17 preparations.

To remedy that lack, this paper proposes that trait (under a slightly different name, however). It also proposes a few other adjuncts that seem generally useful to std::hash users.

\(^1\)Addressing the following issues and National Body comments: LWG 2543, FI 15, GB 69, and LWG 2791.

\(^2\)While it is possible to code a hash specialization that is neither enabled nor disabled, such a specialization does not meet the std::hash requirements. See §3 for details.
2 Proposals

2.1 is_enabled_hash

The requirements for an enabled std::hash specialization are specified in [unord.hash]/4. We propose a corresponding new trait, is_enabled_hash, to decide at compile time whether a given specialization meets those specifications.

The following expository implementation illustrates the trait’s proposed semantics:

```
template< typename H >
struct is_enabled_hash : false_type { };

template< typename T >
requires is_default_constructible_v<hash<T>>
and is_copy_constructible_v <hash<T>>
and is_move_constructible_v <hash<T>>
and is_copyAssignable_v  <hash<T>>
and is_moveAssignable_v  <hash<T>>
and is_destructible_v    <hash<T>>
and is_swappable_v       <hash<T>>
and is_invocable_v       <hash<T>, T>
and is_same_v<size_t, decltype(hash<T>{}(declval<T>()))>
and is_same_v<size_t, decltype(hash<T>{}(declval<T &>()))>
and is_same_v<size_t, decltype(hash<T>{}(declval<T const &>()))>
struct
  is_enabled_hash< hash<T> > : true_type { };
```

As part of this proposal, user specialization of this template is not permitted, just as is the case for nearly all type traits.

2.2 hash_for and is_hashable

Upon reviewing and approving a draft of the above-proposed trait, Lippincott commented:

> Also, the question I imagine most people will want answered is “Can I hash T?” rather than “Is H an enabled hasher?” I’d like to add is_hashable as a shortcut . . .

The following expository implementation, a slight expansion of Lippincott’s code, illustrates the intended semantics of this proposed “shortcut”:

```
template< class T >
using hash_for = hash< remove_cvref_t<T> >;

template< class T >
using is_hashable = is_enabled_hash< hash_for<T> >;

template< class T >
inline constexpr bool is_hashable_v = is_hashable<T>::value;
```

---

3See §3 for alternative designs.

4Lisa Lippincott: “Re: Follow-up to P0513R0.” Personal correspondence, 2016–12–09.
2.3 hash_value

Finally, Lippincott suggested:\textsuperscript{5}

And if it’s not there already, we could use a function for calculating hashes. Making
every user instantiate, construct, and call the right specialization is for the birds.

The following expository implementation is adapted from Lippincott’s code; user specialization of
this template, too, is not permitted. By design, attempted instantiation of this template for a type
without an enabled hash yields an ill-formed program:

```cpp
template< class T >
  requires is_hashable_v<T>
size_t
hash_value( T&& t )
  noexcept( noexcept(hash_for<T>{}(std::forward<T>(t))) )
{
  return hash_for<T>{}( std::forward<T>(t) );
}
```

Note that this proposed template shares its name with a seemingly-similar Boost facility. However, the corresponding Boost documentation states\textsuperscript{6}, in pertinent part:

- “Generally shouldn’t be called directly by users ….”
- “This hash function is not intended for general use, and isn’t guaranteed to be equal during
  separate runs of a program . . . .”

The version proposed herein has no such design restrictions.

2.4 is_nothrow_hashable

Recent adoption of [P0599R1] has emphasized the noexcept nature of most of the library-provided
hash specializations. Because this status may be of special interest in the case of operator(),
we propose a corresponding is_nothrow_hashable trait:

```cpp
template< class T >
inline constexpr bool is_nothrow_hashable_v = is_hashable_v<T>
  and noexcept(hash_value(declval<T>()));
```

3 Alternatives

3.1 Non-{en,dis}abled hashes

As we cited in §1, it is convenient to think of std::hash specializations as “either disabled
(‘poisoned’) or enabled (‘untainted’).” However, it is technically possible to code a specialization
that meets neither definition. Of course, a program with such a specialization runs afoif
[namespace.std]:

\begin{quote}
A program may add a template specialization for any standard library template to
namespace std only if . . . the specialization meets the standard library requirements
for the original template . . . .
\end{quote}

\textsuperscript{5}Ibid.

To what lengths, if any, should the standard library go to diagnose such undefined behavior?

1. Should we respecify the proposed `is_enabled_hash` trait as follows?
   - Have a BaseCharacteristic of `true_type` if template parameter `H` is an enabled specialization of `hash`;
   - have a BaseCharacteristic of `false_type` if `H` is a disabled specialization of `hash`; and
   - be ill-formed\(^7\), otherwise.

2. Alternatively, instead of altering the `is_enabled_hash` specification, should we provide, in addition, an `is_disabled_hash` trait, specified as follows?
   - Have a BaseCharacteristic of `true_type` if template parameter `H` is a disabled specialization of `hash`;
   - have a BaseCharacteristic of `false_type`, otherwise.

**Update:** LEWG expressed no opinion on this issue during this paper’s review in San Diego. We therefore provide no accommodation for `std::hash` specializations that are neither enabled nor disabled.

### 3.2 About `hash_value`

Arthur O’Dwyer raised\(^8\) an objection to the above design for function template `hash_value` on the grounds that “it is a function (template), and so ADL kicks in.” Therefore, he demonstrated, there is code that “builds before, fails to build after” as well as code that “builds both before and after, but with a silent breaking change in behavior.”

Moreover, O’Dwyer opined that “WG21 needs to avoid creating ADL situations on userspace names that are in that sweet spot of ‘uncommon, yet plausible,’ which is exactly where [he believes] `hash_value` falls.” He proposed three designs, paraphrased below, that he would find acceptable alternatives:

- implementation as a member function, e.g., `std::hash<void>::operator()(T&& t)`;
- implementation as a Customization Point Object instead of as a function; or
- renaming “with a less ‘user-space’ spelling,” e.g., `__hash_value` or `apply_enabled_hash`.

However, others have strong reservations even while agreeing with O’Dwyer’s premise. For example, Lisa Lippincott writes\(^9\) that “Arthur’s objection is certainly a valid one; adding a function to namespace std can change the meaning of programs. But the breadth of its applicability gives me pause: at its heart, I think it is an argument against adding almost any function to the library.”

Given such divergent opinions, this paper proposes no wording for any of the cited alternatives, so that LEWG can first decide whether it agrees with O’Dwyer’s stated concern. If LEWG does agree, we will then follow LEWG’s chosen design policy, once established. We ask only that any such decisions be made promptly so as to avoid further delay, already considerable, in this paper’s progress toward C++20.

**Update:** During this paper’s review in San Diego, LEWG addressed this issue by changing this template’s name to `hashed_value`. The proposed wording, below, is consistent with this decision and with the other minor LEWG change requests.

---

\(^7\)This can be implemented via a judiciously-placed `static_assert`, for example.

\(^8\)Arthur O’Dwyer: “[isocpp-lib-ext] Priorities in San Diego?” Personal correspondence, 2018–10–26. (Alas, an earlier draft of this paper’s R5 incorrectly described this correspondence as a posting to the WG21 lib-ext reflector; we deeply regret that mischaracterization.)

4 Proposed wording

4.1 Insert the following row into Table 35 — Standard library feature-test macros. Adjust the placeholder Value as needed so as to denote this proposal’s date of adoption.

<table>
<thead>
<tr>
<th>Macro name</th>
<th>Value</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>_cpp_lib_hash_adjuncts</td>
<td>201903L</td>
<td>&lt;bitset&gt; &lt;functional&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;memory&gt; &lt;optional&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;string&gt; &lt;string_view&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;system_error&gt; &lt;thread&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;typeindex&gt; &lt;variant&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;vector&gt;</td>
</tr>
</tbody>
</table>

4.2 Insert into the synopsis in [functional.syn] as shown.

```cpp
namespace std {

    // 19.14.18, hash function primary template and adjuncts
    template<class T> struct hash;

    template<class T> struct is_enabled_hash;
    template<class T> inline constexpr bool is_enabled_hash_v
    = is_enabled_hash<T>::value;

    template<class T> using hash_for = hash< remove_cvref_t<T> >;

    template<class T> struct is_hashable;
    template<class T> inline constexpr bool is_hashable_v
    = is_hashable<T>::value;

    template<class T>
    size_t hashed_value(const T& t) noexcept(see below);

    template<class T> struct is_nothrow_hashable;
    template<class T> inline constexpr bool is_nothrow_hashable_v
    = is_nothrow_hashable<T>::value;

    ...
}
```

4.3 Retitle [unord.hash] as shown. (Note that there is a pre-existing discrepancy between this title and the corresponding entry in the synopsis (see above); we recommend that the Project Editor determine whether and how this mismatch should be resolved.)

19.14.18 Class template `hash` and `adjuncts` [unord.hash]

---

10Proposed additions and deletions are based on [N4791]. Editorial instructions and drafting notes look like this.
As shown, reword the last sentence of paragraph 2 to take advantage of recently-improved terminology. (This is a drive-by fix.)

For any type `Key` for which there is neither the library nor the user provides an explicit or partial library-provided nor a program-provided specialization of the class template `std::hash`, `std::hash<Key>` is disabled.

Append the following new text to the retitled [unord.hash].

The behavior of a program that adds a specialization `std::hash<T>` is undefined unless `std::is_same_v<T, std::decay_t<T>>` is true.

```cpp
template<class T> struct is_enabled_hash;
```

**Remarks:** Each specialization of this template shall meet the Cpp17UnaryTypeTrait requirements ([meta.rqmts]) with a BaseCharacteristic of `true_type` if T is an enabled specialization of `std::hash` ([unord.hash]) and a BaseCharacteristic of `false_type` otherwise. [Note: The latter does not necessarily imply that T is a disabled specialization of `std::hash`. — end note] The behavior of a program that adds specializations for this template is undefined.

```cpp
template<class T> struct is_hashable;
```

**Remarks:** Each specialization of this template shall meet the Cpp17UnaryTypeTrait requirements ([meta.rqmts]) with a BaseCharacteristic of `true_type` if `std::hash_for<T>` is an enabled specialization of `std::hash` ([unord.hash]) and a BaseCharacteristic of `false_type` otherwise. The behavior of a program that adds specializations for this template is undefined.

```cpp
template<class T>
size_t hashed_value(const T& t) noexcept(
```

**Constraints:** `std::is_hashable_v<T>` is true.

**Effects:** Equivalent to: `return std::hash_for<T>{}(t);`

**Remarks:** The expression inside `noexcept` is equivalent to: `noexcept(std::hash_for<T>{}(t));`

```cpp
template<class T> struct is_nothrow_hashable;
```

**Remarks:** Each specialization of this template shall meet the Cpp17UnaryTypeTrait requirements ([meta.rqmts]) with a BaseCharacteristic of `true_type` if `std::is_hashable_v<T> && noexcept(std::hashed_value(std::declval<const T&>())` is true and a BaseCharacteristic of `false_type` otherwise. The behavior of a program that adds specializations for this template is undefined.

5 Acknowledgments

Special thanks to Lisa Lippincott, who inspired essentially all of this proposed functionality. Thanks also to Andrey Semashev and the other readers of this paper’s pre-publication drafts for their thoughtful comments.
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 P0549R0, pre-Kona.</td>
</tr>
<tr>
<td>1</td>
<td>2017–06–11</td>
<td>• Added is_nothrow_hashable (§2.4, etc.). • Updated relative to the post-Kona Working Draft [N4659]. • Made minor editorial improvements. • Published as P0549R1, pre-Toronto.</td>
</tr>
<tr>
<td>2</td>
<td>2017–10–10</td>
<td>• Updated relative to the post-Toronto Working Draft [N4687]. • Revised citations to use wg21.link. • Made minor technical and editorial improvements. • Published as P0549R2, pre-Albuquerque.</td>
</tr>
<tr>
<td>3</td>
<td>2018–02–03</td>
<td>• Updated relative to the post-Albuquerque Working Draft [N4713]. • Added feature-test macro recommendation. • Published as P0549R3, pre-Jacksonville.</td>
</tr>
<tr>
<td>4</td>
<td>2018–10–07</td>
<td>• Rebased on [N4762], taking advantage of recent new library specification elements and new blanket prohibition on specializing library function templates. • Published as P0549R4, pre-San Diego.</td>
</tr>
<tr>
<td>5</td>
<td>2019–01–20</td>
<td>• Rebased on [N4791] (post-San Diego). • Added §3.2. • Tweaked/corrected example code and proposed wording. • Applied LEWG’s and LWG’s guidance from San Diego. • Published as P0549R5, pre-Kona.</td>
</tr>
</tbody>
</table>