Adjuncts to `std::hash`

Contents

Abstract

Inspired by Lippincott's paper [P0513R0] and subsequent correspondence with her, this paper proposes, for the standard library, a few templates of general use in connection with `std::hash`.

> *HASH, x. There is no definition for this word—nobody knows what hash is.*
> — AMBROSE BIERCE

> *He took the Who's feast,*
> *he took the Who pudding, he took the roast beast.*
> *He cleaned out that ice box as quick as a flash.*
> *Why, the Grinch even took their last can of Who hash.*
> — DR. SEUSS (né THEODOR SEUSS GEISEL)

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`[3]

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:

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

4   template< typename T >
5     requires is_default_constructible_v<hash<T>>
6           and is_copy_constructible_v    <hash<T>>
7           and is_move_constructible_v    <hash<T>>
8           and is_copy_assignable_v       <hash<T>>
9           and is_move_assignable_v       <hash<T>>
10          and is_destructible_v          <hash<T>>
11          and is_swappable_v             <hash<T>>
12          and is_invocable_v             <hash<T>, T>
13          and is_same_v<size_t, decltype(hash<T>{}(declval<T        >()))>
14          and is_same_v<size_t, decltype(hash<T>{}(declval<T      &>()))>
15          and is_same_v<size_t, decltype(hash<T>{}(declval<T const&>()))>
16  struct
17    is_enabled_hash< hash<T> > : true_type { };

19  template< typename H >
20  inline constexpr bool is_enabled_hash_v = is_enabled_hash<H>::value;
```

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:[4]

> 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":

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

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

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

---

[3]See §3 for alternative designs.

[4]Lisa Lippincott: "Re: Follow-up to P0513R0." Personal correspondence, 2016–12–09.

2.3 `hash_value`

Finally, Lippincott suggested:[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:

```
1  template< class T >
2    requires is_hashable_v<T>
3  size_t
4    hash_value( T&& t )
5    noexcept( noexcept(hash_for<T>{}(std::forward<T>(t))) )
6  {
7    return hash_for<T>{}( std::forward<T>(t) );
8  }
```

Note that this proposed template shares its name with a seemingly-similar Boost facility. However, the corresponding Boost documentation states[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:

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

5  template< class T >
6  using is_nothrow_hashable = bool_constant< is_nothrow_hashable_v >;
```

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 afoul of [namespace.std]:

> 1 . . . . 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 . . . .

---

[5]*Ibid.*

[6] See http://www.boost.org/doc/libs/1_63_0/doc/html/hash/reference.html#boost.hash_value_idp743313104.

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.)

[9]Lisa Lippincott: "[isocpp-lib-ext] D0549R5: 'Adjuncts to std::hash'." lib-ext reflector message, 2018–11–04.

4 Proposed wording[10]

Macro name	Value	Header(s)
⋮		
`__cpp_lib_hash_adjuncts`	201903L	`<bitset> <functional>` `<memory> <optional>` `<string> <string_view>` `<system_error> <thread>` `<typeindex> <variant>` `<vector>`
⋮		

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

```
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]

---

[10]Proposed additions and ~~deletions~~ are based on [N4791]. Editorial instructions and drafting notes look like `this` .

4.4 As shown, reword the last sentence of paragraph 2 to take advantage of recently-improved terminology. (This is a drive-by fix.)

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

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

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

template<class T> struct is_enabled_hash;

7 *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 **hash** ([unord.hash]) and a BaseCharacteristic of **false_type** otherwise. [*Note:* The latter does not necessarily imply that **T** is a disabled specialization of **hash**. — *end note*] The behavior of a program that adds specializations for this template is undefined.

template<class T> struct is_hashable;

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

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

9 *Constraints:* **is_hashable_v<T>** is **true**.

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

11 *Remarks:* The expression inside **noexcept** is equivalent to: **noexcept(hash_for<T>{}(t)).**

template<class T> struct is_nothrow_hashable;

12 *Remarks:* Each specialization of this template shall meet the *Cpp17UnaryTypeTrait* requirements ([meta.rqmts]) with a BaseCharacteristic of **true_type** if **is_hashable_v<T> && noexcept(hashed_value(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

6 Bibliography

[N4659]   Richard Smith: "Working Draft, Standard for Programming Language C++." ISO/IEC JTC1/ SC22/WG21 document N4659 (post-Kona mailing), 2017–03–21. http://wg21.link/n4659.

[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.

[N4713]   Richard Smith: "Working Draft, Standard for Programming Language C++." ISO/IEC JTC1/ SC22/WG21 document N4713 (post-Albuquerque mailing), 2017–11–27. http://wg21.link/ n4713.

[N4762]   Richard Smith: "Working Draft, Standard for Programming Language C++." ISO/IEC JTC1/ SC22/WG21 document N4762 (corrected post-Rappersville mailing), 2018–07–07. http://wg21. link/n4762.

[N4791]   Richard Smith: "Working Draft, Standard for Programming Language C++." ISO/IEC JTC1/ SC22/WG21 document N4791 (post-San Diego mailing), 2018–12–07. https://wg21.link/ n4791.

[P0513R0] Lisa Lippincott: "Poisoning the Hash." ISO/IEC JTC1/SC22/WG21 document P0513R0 (post-Issaquah mailing), 2016–11–10. http://wg21.link/p0513r0.

[P0599R1] Nicolai Josuttis: "**noexcept** for Hash Functions." ISO/IEC JTC1/SC22/WG21 document P0599R1 (post-Kona mailing), 2017–03–02. http://wg21.link/p0599R1.

7 Document history

Version	Date	Changes
0	2017–02–01	• Published as P0549R0, pre-Kona.
1	2017–06–11	• 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.
2	2017–10–10	• 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.
3	2018–02–03	• Updated relative to the post-Albuquerque Working Draft [N4713]. • Added feature-test macro recommendation. • Published as P0549R3, pre-Jacksonville.
4	2018–10–07	• 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.
5	2019–01–20	• 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.