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 §4 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 exemplifies 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_copy_assignable_v      <hash<T>>
       and is_move_assignable_v      <hash<T>>
       and is_destructible_v         <hash<T>>
       and is_swappable_v            <hash<T>>
       and is_callable_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 { };

template< typename H >
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":

```
template< typename T >  // exposition only
using uncvref_t = remove_cv_t< remove_reference_t<T> >;

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

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

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

---

[3]See §4 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:

```
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,[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:

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

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

3 Proposed wording[7]

3.1 Insert into the synopsis in [function.objects] (23.14) as shown.

---

[5]*Ibid.*

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

[7]All proposed additions (there are no ~~deletions~~) are relative to the post-Kona Working Draft [N4659]. Editorial notes are displayed against a gray background.

```
namespace std {
  ...
  // 23.14.15, hash function primary template and adjuncts
  template <class T> struct hash;
  template <class H> struct is_enabled_hash;
  template <class T> using hash_for = hash<see below>;
  template <class T> using is_hashable = is_enabled_hash< hash_for<T> >;
  template <class T> constexpr bool is_hashable_v = is_hashable<T>::value;
  template <class T> size_t hash_value(T&& t) noexcept(see below);
  template<class T> constexpr bool is_nothrow_hashable_v = is_hashable_v<T>
    and noexcept(noexcept(hash_value(declval<T>())));
  template<class T>
    using is_nothrow_hashable = bool_constant< is_nothrow_hashable_v >;
  ...
}
```

3.2 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.)

23.14.15 Class template **hash** and adjuncts   [unord.hash]

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

   template <class H> struct is_enabled_hash;

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

   template <class T> using hash_for = hash<*see below*>;

6 *Remarks:* The template argument to **hash** shall correspond to **remove_cv_t<remove_reference_t<T>>**.

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

7 The expression inside **noexcept** is equivalent to:
noexcept(hash_for<T>{}(std::forward<T>(t))).

8 *Requires:* Participates in overload resolution only if **is_hashable_v<T>** is **true**.

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

10 *Remarks:* The behavior of a program that adds specializations for this template is undefined.

4   Alternatives

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 . . . .

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

1. In particular, 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,[8] 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.

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

[N4659]   Richard Smith: "Working Draft, Standard for Programming Language C++." ISO/IEC JTC1/SC22/WG21 document N4659 (post-Kona mailing), 2017–03–21. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4659.pdf.

[P0513R0] Lisa Lippincott: "Poisoning the Hash." ISO/IEC JTC1/SC22/WG21 document P0513R0 (post-Issaquah mailing), 2016–11–10. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0513r0.pdf.

[P0599R1] Nicolai Josuttis: "**noexcept** for Hash Functions." ISO/IEC JTC1/SC22/WG21 document P0599R1 (post-Kona mailing), 2017–03–02. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0599R1.pdf.

7 Document history

Version	Date	Changes
0	2017-02-01	• Published as P0549R0.
1	2017-06-11	• Added **is_nothrow_hashable** (§2.4, etc.). • Updated relative to the post-Kona Working Draft [N4659]. • Minor editorial improvements. • Published as P0549R1.

---

[8]This can be implemented via a judiciously-placed **static_assert**, for example.