

Document: P0513R0
Date: 2016-11-10
Reply-To: Lisa Lippincott <lisa.e.lippincott@gmail.com>
Audience: Library Working Group

Poisoning the Hash

Lisa Lippincott

Problem: By introducing “poisoned” specializations of `hash`, the proposed resolution for issue LWG 2543 creates a situation where most specializations of `hash`, including expected specializations by users, do not satisfy the requirements placed on specializations of `hash`.

National body comment FI 15 draws `hash<optional<T>>` into the problem; GB 69 and LWG 2791 are also addressed incidentally here.

Solution: Clarify that each specialization must be either *disabled* (“poisoned”) or *enabled* (“untainted”) and provide separate specifications for each kind of specialization.

1 Changes to 20.14.14 [unord.hash]

The unordered associative containers defined in 23.5 use specializations of the class template `hash` as the default hash function.



Each specialization of `hash` is either enabled or disabled, as described below. [Note: Enabled specializations meet the requirements of `Hash`, and disabled specializations do not. —end note] Each header that declares the template `hash` provides enabled specializations of `hash` for `nullptr_t` and all cv-qualified arithmetic, enumeration, and pointer types. For any type `Key` for which neither the library nor the user provides an explicit or partial specialization of the class template `hash`, `hash<Key>` is disabled. If the library provides an explicit or partial specialization of `hash<Key>`, that specialization is enabled except as noted otherwise.

If `H` is a disabled specialization of `hash`, these values are false: `is_default_constructible_v<H>`, `is_copy_constructible_v<H>`, `is_move_constructible_v<H>`, `is_copy_assignable_v<H>`, and `is_move_assignable_v<H>`. Disabled specializations of `hash` are not function object types ([function.objects]). [Note: This means that the specialization of `hash` exists, but any attempts to use it as a `Hash` will be ill-formed. —end note]

For all object types `Key` for which there exists a specialization `hash<Key>`, and for all integral and enumeration types (7.2) `Key`, the instantiation `hash<Key>` shall:

An enabled specialization `hash<Key>` will:

- satisfy the `Hash` requirements (17.6.3.4), with `Key` as the function call argument type, the `DefaultConstructible` requirements (Table 20), the `CopyAssignable` requirements (Table 24),
- be swappable (17.6.3.2) for lvalues,
- satisfy the requirement that if `k1 == k2` is true, `h(k1) == h(k2)` is also true, where `h` is an object of type `hash<Key>` and `k1` and `k2` are objects of type `Key`;
- satisfy the requirement that the expression `h(k)`, where `h` is an object of type `hash<Key>` and `k` is an object of type `Key`, shall not throw an exception unless `hash<Key>` is a user-defined specialization that depends on at least one user-defined type.

```
template <> struct hash<bool>;
template <> struct hash<char>;
template <> struct hash<signed char>;
template <> struct hash<unsigned char>;
template <> struct hash<char16_t>;
template <> struct hash<char32_t>;
template <> struct hash<wchar_t>;
template <> struct hash<short>;
template <> struct hash<unsigned short>;
template <> struct hash<int>;
template <> struct hash<unsigned int>;
template <> struct hash<long>;
template <> struct hash<unsigned long>;
template <> struct hash<long long>;
template <> struct hash<unsigned long long>;
template <> struct hash<float>;
template <> struct hash<double>;
template <> struct hash<long double>;
template <class T> struct hash<T*>;
```

The template specializations shall meet the requirements of class template `hash` (20.14.14).

1.1 Rationale

The detailed description of poisoning roughly follows the proposed resolution of LWG 2543.

Specifying that explicit and partial specializations of `hash` in the library are enabled by default reduces the need to make changes elsewhere.

There's no need to require explicit specializations any more.

The final sentence was recursive.

2 Changes to 20.14 [function.objects] ¶2, portion corresponding to 20.14.14

```
// 20.14.14, hash function primary template:  
template <class T> struct hash;
```

```
// Hash function specializations  
template <> struct hash<bool>;  
template <> struct hash<char>;  
template <> struct hash<signed char>;  
template <> struct hash<unsigned char>;  
template <> struct hash<char16_t>;  
template <> struct hash<char32_t>;  
template <> struct hash<wchar_t>;  
template <> struct hash<short>;  
template <> struct hash<unsigned short>;  
template <> struct hash<int>;  
template <> struct hash<unsigned int>;  
template <> struct hash<long>;  
template <> struct hash<long long>;  
template <> struct hash<unsigned long>;  
template <> struct hash<unsigned long long>;  
template <> struct hash<float>;  
template <> struct hash<double>;  
template <> struct hash<long double>;  
template <class T> struct hash<T*>;
```

3 Changes to 20.6.10 [optional.hash], addressing FI 15

```
template <class T> struct hash<optional<T>>;
```

Requires: The template specialization `hash<T>` shall meet the requirements of class template `hash` (20.14.14). The template specialization `hash<optional<T>>` shall meet the requirements of class template `hash`.

The specialization `hash<optional<T>>` is enabled (20.14.14) if and only if `hash<remove_const_t<T>>` is enabled.

For

When enabled, for

an object `o` of type `optional<T>`, if `bool(o) == true`, `hash<optional<T>>()(o)` shall evaluate to the same value as

```
hash<T>()( *o );
```

```
hash<remove_const_t<T>>()( *o );
```

otherwise it evaluates to an unspecified value.

4 Changes to 20.7.11 [variant.hash], addressing LWG 2543

```
template <class... Types> struct hash<variant<Types...>>;
```

The template specialization `hash<T>` shall meet the requirements of class template `hash` (20.14.14) for all `T` in `Types...`. The template specialization `hash<variant<Types...>>` shall meet the requirements of class template `hash`.

The specialization `hash<variant<Types...>>` is enabled (20.14.14) if and only if every specialization in `hash<remove_const_t<Types>>...` is enabled.

5 Changes to 20.11.2.7 [util.smartptr.hash]

```
template <class T, class D> struct hash<unique_ptr<T, D>>;
```

The template specialization shall meet the requirements of class template `hash` (20.14.14).

Letting `UP` be `unique_ptr<T,D>`, the specialization `hash<UP>` is enabled (20.14.14) if and only if `hash<typename UP::pointer>` is enabled.

For

When enabled, for

an object `p` of type `UP`,

where `UP` is `unique_ptr<T, D>`,

`hash<UP>()(p)` shall evaluate to the same value as `hash<typename UP::pointer>()(p.get())`.

Requires: The specialization `hash<typename UP::pointer>` shall be well-formed and well-defined, and shall meet the requirements of class template `hash` (20.14.14).

```
template <class T> struct hash<shared_ptr<T>>;
```

The template specialization shall meet the requirements of class template `hash` (20.14.14).

For an object `p` of type `shared_ptr<T>`, `hash<shared_ptr<T>>()(p)` shall evaluate to the same value as `hash<T*>()(p.get())`.

6 Changes to D.8.2 [depr.func.adaptor.typedefs]

¶6

For all object types `Key` for which there exists a specialization `hash<Key>`, and for all enumeration types (7.2) `Key`, the instantiation `hash<Key>` shall provide

All enabled specializations `hash<Key>` of `hash` (20.14.14) provide

two nested types, `result_type` and `argument_type`, which shall be synonyms for `size_t` and `Key`, respectively.

7 Redundancy-reducing changes

These changes simply remove redundancies of the form “specializations of `hash` meet the requirements placed on specializations of `hash`.”

Where it appears, the new wording “is enabled” is unnecessary, as (20.14.14) makes enabled the default for all explicit or partial specializations in the library. The text is just there to keep the paragraph from being empty.

7.1 Changes to 19.5.6 [syserr.hash]

```
template <> struct hash<error_code>;
```

The template specialization shall meet the requirements of class template `hash` (20.14.14).

The specialization is enabled (20.14.14).

7.2 Changes to 20.9.3 [bitset.hash]

```
template <size_t N> struct hash<bitset<N>>;
```

The template specialization shall meet the requirements of class template hash (20.14.14).

The specialization is enabled (20.14.14).

7.3 Changes to 20.18.4 [type.index.hash]

```
template <> struct hash<type_index>;
```

The template specialization shall meet the requirements of class template hash (20.14.14).

For an object `index` of type `type_index`, `hash<type_index>()(index)` shall evaluate to the same result as `index.hash_code()`.

7.4 Changes to 21.3.4 [basic.string.hash]

```
template <> struct hash<string>;  
template <> struct hash<u16string>;  
template <> struct hash<u32string>;  
template <> struct hash<wstring>;
```

The template specializations shall meet the requirements of class template hash (20.14.14).

If `S` is one of these string types, `SV` is the corresponding string view type, and `s` is an object of type `S`, then `hash<S>()(s) == hash<SV>()(SV(s))`.

7.5 Changes to 21.4.5 [string.view.hash]

```
template <> struct hash<string_view>;  
template <> struct hash<u16string_view>;  
template <> struct hash<u32string_view>;  
template <> struct hash<wstring_view>;
```

The template specializations shall meet the requirements of class template hash (20.14.14).

The specializations are enabled (20.14.14).

7.6 Changes to 23.3.12 [vector.bool] ¶7

```
template <class Allocator> struct hash<vector<bool,Allocator>>;
```

The template specialization shall meet the requirements of class template hash (20.14.14).

The specialization is enabled (20.14.14).

7.7 Changes to 30.3.1.1 [thread.thread.id] ¶14

```
template <> struct hash<thread::id>;
```

The template specialization shall meet the requirements of class template hash (20.14.14).

The specialization is enabled (20.14.14).

8 Further remarks

There ought to be a standard trait `hash_enabled<T>`.

9 Acknowledgements

Thanks to Marshall Clow, Howard Hinnant, and Ville Voutilainen for assistance on this paper.