Project: ISO JTC1/SC22/WG21: Programming Language C++
Doc No: WG21 **P0599R1**
Date: 2017-03-02
Reply to: Nicolai Josuttis (nico@josuttis.de)
Audience: LWG
Prev. Version: P0599R0

noexcept for Hash Functions

For C++17; US 140 requests:

> Specializations of std::hash for arithmetic, pointer, and standard library types should not be allowed to throw. The constructors, assignment operators, and function call operator should all be marked as noexcept. It might be reasonable to consider making this a binding requirement on user specializations of the hash template as well (in p1) but that may be big a change to make at this stage.

Discussing it informally in LWG in Kona 2017 seems to result in the following conclusion:

hash type	**should be noexcept?**	**Remark**
hash<error_code>	yes	
hash<optional<T>>	**no**	same hash as with underlying type (might throw!)
hash<variant<Types...>>	**no**	no defined behavior of hash function with respect to current value
hash<monostate>	yes	
hash<bitset<N>>	yes	
hash<unique_ptr<T, D>>	no	same hash as for underlying raw pointer, but might be fancy pointer
hash<shared_ptr<T>>	yes	same hash as for underlying raw pointer (no fancy pointer)
hash<NUMERIC>	yes	for all integral types (incl. bool and char) and floating-point types
hash<T*>	yes	(uses the address (can't look at the value because it might change))
hash<type_index>	yes	same as hash_code() of passed index
hash<string>	yes	
hash<u16string>	yes	
hash<u32string>	yes	
hash<wstring>	yes	
hash<string_view>	yes	no guarantee to match string hash value
hash<u16string_view>	yes	no guarantee to match u16string hash value
hash<u32string_view>	yes	no guarantee to match u32string hash value
hash<wstring_view>	yes	no guarantee to match wstring hash value
hash<vector<bool, Allocator>>	yes	
hash<thread::id>	yes	

Discussion result in Kona was to add a blanket statement and special remarks for the “no” cases, which are:

- optional and variant, because they use the hash function of the wrapped type(s), which might throw (and no conditional noexcept should be used)
- unique_ptr, because the hash value depends on the underlying raw pointer, which might be a fancy pointer
  - Note: shared_ptr may not have fancy pointers as raw pointer so we require noexcept here

Proposed Wording

(All against N4618)

20.6.10 Hash support [optional.hash]:

§1 (for optional<>):

The specialization hash<optional<T>> is enabled (20.14.14) if and only if hash<remove_const_t<T>> is enabled. When enabled, for an object o of type optional<T>, if bool(o) == true, then hash<optional<T>>()(o) shall evaluate to the same value as hash<remove_const_t<T>>()(*o); otherwise it evaluates to an unspecified value. The member functions are not guaranteed to be noexcept.

20.7.11 Hash support [variant.hash]:

§1 (for variant<>):

The specialization hash<variant<Types...>> is enabled (20.14.14) if and only if every specialization in hash<remove_const_t<Types>>... is enabled. The member functions are not guaranteed to be noexcept.

20.11.2.7 Smart pointer hash support [util.smartptr.hash]:

§1 (for unique_ptr<>):

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. When enabled, for an object p of type UP, hash<UP>()(p) shall evaluate to the same value as hash<typename UP::pointer>()(p.get()). The member functions are not guaranteed to be noexcept.

20.14.14 Class template hash [unord.hash]:

Split and modify §2 as follows:

Start a new paragraph with the current last sentence extended:

<new paragraph>
If the library provides an explicit or partial specialization of hash<Key>, that specialization is enabled except as noted otherwise, and its member functions are noexcept except as noted otherwise..