std::hash

Defined in header <functional>

```
template< class Key >
struct hash;
```
(since C++11)

Each specialization of this template is either *enabled* ("untainted") or *disabled* ("poisoned").

The *enabled* specializations of the hash template defines a function object that implements a hash function . Instances of this function object satisfy *Hash*. In particular, they define an `operator() const` that:

1. Accepts a single parameter of type Key.
2. Returns a value of type `std::size_t` that represents the hash value of the parameter.
3. Does not throw exceptions when called.
4. For two parameters k1 and k2 that are equal, `std::hash<Key>()(k1) == std::hash<Key>()(k2)`.
5. For two different parameters k1 and k2 that are not equal, the probability that `std::hash<Key>()(k1) == std::hash<Key>()(k2)` should be very small, approaching `1.0/std::numeric_limits<std::size_t>::max()`.

All explicit and partial specializations of hash provided by the standard library are *DefaultConstructible*, *CopyAssignable*, *Swappable* and *Destructible*. User-provided specializations of hash also must meet those requirements.

The unordered associative containers std::unordered_set, std::unordered_multiset, std::unordered_map, std::unordered_multimap use specializations of the template **std::hash** as the default hash function.

For every type Key for which neither the library nor the user provides an enabled specialization std::hash<Key>, that specialization exists and is disabled. Disabled specializations do not satisfy *Hash*, do not satisfy *FunctionObject*, and following values are all `false`:

- `std::is_default_constructible<std::hash<Key>>::value`
- `std::is_copy_constructible<std::hash<Key>>::value`
- `std::is_move_constructible<std::hash<Key>>::value`
- `std::is_copy_assignable<std::hash<Key>>::value`
- `std::is_move_assignable<std::hash<Key>>::value`

In other words, they exist, but cannot be used.

Notes

The actual hash functions are implementation-dependent and are not required to fulfill any other quality criteria except those specified above. Notably, some implementations use trivial (identity) hash functions which map an integer to itself. In other words, these hash functions are designed to work with unordered associative containers, but not as cryptographic hashes, for example.

Hash functions are only required to produce the same result for the same input within a single execution of a program; this allows salted hashes that prevent collision denial-of-service attacks.

There is no specialization for C strings. `std::hash<const char*>` produces a hash of the value of the pointer (the memory address), it does not examine the contents of any character array.

Member types

(until C++20)

Member type	Definition
argument_type(deprecated in C++17)	Key
result_type(deprecated in C++17)	std::size_t

Member functions

(constructor)	constructs a hash function object (public member function)
operator()	calculates the hash of the argument (public member function)

Standard specializations for basic types

Defined in header <functional>

```
template<> struct hash<bool>;
template<> struct hash<char>;
template<> struct hash<signed char>;
template<> struct hash<unsigned char>;
```

```
template<> struct hash<char8_t>;         // C++20
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<> struct hash<std::nullptr_t>;
template< class T > struct hash<T*>;
```

In addition to the above, the standard library provides specializations for all (scoped and unscoped) enumeration types. These may be (but are not required to be) implemented as `std::hash<std::underlying_type<Enum>::type>`.

The standard library provides enabled specializations of `std::hash` for `std::nullptr_t` and all cv-unqualified arithmetic types (including any extended integer types), all enumeration types, and all pointer types.

Each standard library header that declares the template `std::hash` provides all enabled specializations described above. These headers include `<string>`, `<system_error>`, `<bitset>`, `<memory>`, `<typeindex>`, `<vector>`, `<thread>`, `<optional>`, `<variant>`, `<string_view>` (since C++17), `<coroutine>` (since C++20), `<stacktrace>` (since C++23).

All member functions of all standard library specializations of this template are `noexcept` except for the member functions of `std::hash<std::optional>`, `std::hash<std::variant>`, and `std::hash<std::unique_ptr>`. (since C++17)

Standard specializations for library types

std::hash<std::coroutine_handle> (C++20)	hash support for std::coroutine_handle (class template specialization)
std::hash<std::error_code> (C++11)	hash support for std::error_code (class template specialization)
std::hash<std::error_condition> (C++17)	hash support for std::error_condition (class template specialization)
std::hash<std::stacktrace_entry> (C++23)	hash support for std::stacktrace_entry (class template specialization)
std::hash<std::basic_stacktrace> (C++23)	hash support for std::basic_stacktrace (class template specialization)
std::hash<std::optional> (C++17)	specializes the **std::hash** algorithm (class template specialization)
std::hash<std::variant> (C++17)	specializes the **std::hash** algorithm (class template specialization)
std::hash<std::monostate> (C++17)	hash support for std::monostate (class template specialization)
std::hash<std::bitset> (C++11)	hash support for std::bitset (class template specialization)
std::hash<std::unique_ptr> (C++11)	hash support for std::unique_ptr (class template specialization)
std::hash<std::shared_ptr> (C++11)	hash support for std::shared_ptr (class template specialization)
std::hash<std::type_index> (C++11)	hash support for std::type_index (class template specialization)
std::hash<std::string> (C++11) **std::hash**<std::u8string> (C++20) **std::hash**<std::u16string> (C++11) **std::hash**<std::u32string> (C++11) **std::hash**<std::wstring> (C++11) **std::hash**<std::pmr::string> (C++17) **std::hash**<std::pmr::u8string> (C++20) **std::hash**<std::pmr::u16string> (C++17) **std::hash**<std::pmr::u32string> (C++17) **std::hash**<std::pmr::wstring> (C++17)	hash support for strings (class template specialization)
std::hash<std::string_view> (C++17) **std::hash**<std::wstring_view> (C++17) **std::hash**<std::u8string_view> (C++20) **std::hash**<std::u16string_view> (C++17) **std::hash**<std::u32string_view> (C++17)	hash support for string views (class template specialization)
std::hash<std::vector<bool>> (C++11)	hash support for `std::vector<bool>` (class template specialization)
std::hash<std::filesystem::path> (C++17)	hash support for std::filesystem::path (class template specialization)
std::hash<std::thread::id> (C++11)	hash support for std::thread::id (class template specialization)

Note: additional specializations for `std::pair` and the standard container types, as well as utility functions to compose hashes are available in `boost::hash` (http://www.boost.org/doc/libs/release/doc/html/hash/reference.html) .

Example

Run this code

```
#include <iostream>
#include <iomanip>
#include <functional>
#include <string>
#include <unordered_set>

struct S {
    std::string first_name;
    std::string last_name;
};
bool operator==(const S& lhs, const S& rhs) {
    return lhs.first_name == rhs.first_name && lhs.last_name == rhs.last_name;
}

// custom hash can be a standalone function object:
struct MyHash
{
    std::size_t operator()(S const& s) const noexcept
    {
        std::size_t h1 = std::hash<std::string>{}(s.first_name);
        std::size_t h2 = std::hash<std::string>{}(s.last_name);
        return h1 ^ (h2 << 1); // or use boost::hash_combine
    }
};

// custom specialization of std::hash can be injected in namespace std
template<>
struct std::hash<S>
{
    std::size_t operator()(S const& s) const noexcept
    {
        std::size_t h1 = std::hash<std::string>{}(s.first_name);
        std::size_t h2 = std::hash<std::string>{}(s.last_name);
        return h1 ^ (h2 << 1); // or use boost::hash_combine
    }
};

int main()
{
    std::string str = "Meet the new boss...";
    std::size_t str_hash = std::hash<std::string>{}(str);
    std::cout << "hash(" << std::quoted(str) << ") = " << str_hash << '\n';

    S obj = { "Hubert", "Farnsworth" };
    // using the standalone function object
    std::cout << "hash(" << std::quoted(obj.first_name) << ", "
              << std::quoted(obj.last_name) << ") = "
              << MyHash{}(obj) << " (using MyHash)\n" << std::setw(31) << "or "
              << std::hash<S>{}(obj) << " (using injected std::hash<S> specialization)\n";

    // custom hash makes it possible to use custom types in unordered containers
    // The example will use the injected std::hash<S> specialization above,
    // to use MyHash instead, pass it as a second template argument
    std::unordered_set<S> names = {obj, {"Bender", "Rodriguez"}, {"Turanga", "Leela"} };
    for(auto& s: names)
        std::cout << std::quoted(s.first_name) << ' ' << std::quoted(s.last_name) << '\n';
}
```

Possible output:

```
hash("Meet the new boss...") = 1861821886482076440
hash("Hubert", "Farnsworth") = 17622465712001802105 (using MyHash)
                          or 17622465712001802105 (using injected std::hash<S> specializat
"Turanga" "Leela"
"Bender" "Rodriguez"
"Hubert" "Farnsworth"
```

Defect reports

The following behavior-changing defect reports were applied retroactively to previously published C++ standards.

DR	Applied to	Behavior as published	Correct behavior
LWG 2148 (https://cplusplus.github.io/LWG/issue2148)	C++11	specializations for enumerations were missing	provided
LWG 2543 (https://cplusplus.github.io/LWG/issue2543)	C++11	`hash` might not be SFINAE-friendly	made SFINAE-friendly via disabled specializations
LWG 2817 (https://cplusplus.github.io/LWG/issue2817)	C++11	specialization for `nullptr_t` was missing	provided