# Boost.Functional/Hash

Daniel James

Copyright © 2005-2008 Daniel James

Distributed under the Boost Software License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)

## Table of Contents

Introduction ........ 2
Tutorial ........ 3
Extending boost::hash for a custom data type ........ 4
Combining hash values ........ 6
Portability ........ 8
Disabling The Extensions ........ 10
Change Log ........ 11
Rationale ........ 15
Reference ........ 16
Links ........ 30
Acknowledgements ........ 31

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

# Introduction

`boost::hash` is an implementation of the hash function object specified by the Draft Technical Report on C++ Library Extensions (TR1). It is the default hash function for Boost.Unordered, Boost.Intrusive's unordered associative containers, and Boost.MultiIndex's hash indicies and Boost.Bimap's `unordered_set_of`.

As it is compliant with TR1, it will work with:

- integers
- floats
- pointers
- strings

It also implements the extension proposed by Peter Dimov in issue 6.18 of the Library Extension Technical Report Issues List (page 63), this adds support for:

- arrays
- `std::pair`
- the standard containers.
- extending `boost::hash` for custom types.

## Note

This hash function is designed to be used in containers based on the STL and is not suitable as a general purpose hash function. For more details see the rationale.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

# Tutorial

When using a hash index with Boost.MultiIndex, you don't need to do anything to use `boost::hash` as it uses it by default. To find out how to use a user-defined type, read the section on extending boost::hash for a custom data type.

If your standard library supplies its own implementation of the unordered associative containers and you wish to use `boost::hash`, just use an extra template parameter:

```
std::unordered_multiset<int, boost::hash<int> >
        set_of_ints;

std::unordered_set<std::pair<int, int>, boost::hash<std::pair<int, int> >
        set_of_pairs;

std::unordered_map<int, std::string, boost::hash<int> > map_int_to_string;
```

To use `boost::hash` directly, create an instance and call it as a function:

```
#include <boost/functional/hash.hpp>

int main()
{
    boost::hash<std::string> string_hash;

    std::size_t h = string_hash("Hash me");
}
```

For an example of generic use, here is a function to generate a vector containing the hashes of the elements of a container:

```
template <class Container>
std::vector<std::size_t> get_hashes(Container const& x)
{
    std::vector<std::size_t> hashes;
    std::transform(x.begin(), x.end(), std::insert_iterator(hashes),
        boost::hash<typename Container::value_type>());

    return hashes;
}
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

# Extending boost::hash for a custom data type

`boost::hash` is implemented by calling the function `hash_value` [19]. The namespace isn't specified so that it can detect overloads via argument dependant lookup. So if there is a free function `hash_value` in the same namespace as a custom type, it will get called.

If you have a structure `library::book`, where each `book` is uniquely defined by it's member `id`:

```
namespace library
{
    struct book
    {
        int id;
        std::string author;
        std::string title;

        // ....
    };

    bool operator==(book const& a, book const& b)
    {
        return a.id == b.id;
    }
}
```

Then all you would need to do is write the function `library::hash_value`:

```
namespace library
{
    std::size_t hash_value(book const& b)
    {
        boost::hash<int> hasher;
        return hasher(b.id);
    }
}
```

And you can now use `boost::hash` with book:

```
library::book knife(3458, "Zane Grey", "The Hash Knife Outfit");
library::book dandelion(1354, "Paul J. Shanley",
    "Hash & Dandelion Greens");

boost::hash<library::book> book_hasher;
std::size_t knife_hash_value = book_hasher(knife);

// If std::unordered_set is available:
std::unordered_set<library::book, boost::hash<library::book> > books;
books.insert(knife);
books.insert(library::book(2443, "Lindgren, Torgny", "Hash"));
books.insert(library::book(1953, "Snyder, Bernadette M.",
    "Heavenly Hash: A Tasty Mix of a Mother's Meditations"));

assert(books.find(knife) != books.end());
assert(books.find(dandelion) == books.end());
```

The full example can be found in: /libs/functional/hash/examples/books.hpp and /libs/functional/hash/examples/books.cpp.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

## Tip

When writing a hash function, first look at how the equality function works. Objects that are equal must generate the same hash value. When objects are not equal they should generate different hash values. In this object equality was based just on the id so the hash function only hashes the id. If it was based on the object's name and author then the hash function should take them into account (how to do this is discussed in the next section).

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

# Combining hash values

Say you have a point class, representing a two dimensional location:

```
class point
{
    int x;
    int y;
public:
    point() : x(0), y(0) {}
    point(int x, int y) : x(x), y(y) {}

    bool operator==(point const& other) const
    {
        return x == other.x && y == other.y;
    }
};
```

and you wish to use it as the key for an `unordered_map`. You need to customise the hash for this structure. To do this we need to combine the hash values for `x` and `y`. The function `boost::hash_combine` is supplied for this purpose:

```
class point
{
    ...

    friend std::size_t hash_value(point const& p)
    {
        std::size_t seed = 0;
        boost::hash_combine(seed, p.x);
        boost::hash_combine(seed, p.y);

        return seed;
    }

    ...
};
```

Calls to hash_combine incrementally build the hash from the different members of point, it can be repeatedly called for any number of elements. It calls `hash_value` `[19]` on the supplied element, and combines it with the seed.

Full code for this example is at /libs/functional/hash/examples/point.cpp.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

## Note

When using `boost::hash_combine` the order of the calls matters.

```
    std::size_t seed = 0;
    boost::hash_combine(seed, 1);
    boost::hash_combine(seed, 2);
```

results in a different seed to:

```
    std::size_t seed = 0;
    boost::hash_combine(seed, 2);
    boost::hash_combine(seed, 1);
```

If you are calculating a hash value for data where the order of the data doesn't matter in comparisons (e.g. a set) you will have to ensure that the data is always supplied in the same order.

To calculate the hash of an iterator range you can use `boost::hash_range [17]`:

```
std::vector<std::string> some_strings;
std::size_t hash = boost::hash_range [17](some_strings.begin(), some_strings.end());
```

Note that when writing template classes, you might not want to include the main hash header as it's quite an expensive include that brings in a lot of other headers, so instead you can include the `<boost/functional/hash_fwd.hpp>` header which forward declares `boost::hash`, `boost::hash_range [17]` and `boost::hash_combine`. You'll need to include the main header before instantiating `boost::hash`. When using a container that uses `boost::hash` it should do that for you, so your type will work fine with the boost hash containers. There's an example of this in template.hpp and template.cpp.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

# Portability

`boost::hash` is written to be as portable as possible, but unfortunately, several older compilers don't support argument dependent lookup (ADL) - the mechanism used for customisation. On those compilers custom overloads for `hash_value` needs to be declared in the boost namespace.

On a strictly standards compliant compiler, an overload defined in the boost namespace won't be found when `boost::hash` is instantiated, so for these compilers the overload should only be declared in the same namespace as the class.

Let's say we have a simple custom type:

```
namespace foo
{
    template <class T>
    class custom_type
    {
        T value;
    public:
        custom_type(T x) : value(x) {}

        friend std::size_t hash_value(custom_type x)
        {
            boost::hash<int> hasher;
            return hasher(x.value);
        }
    };
}
```

On a compliant compiler, when `hash_value` is called for this type, it will look at the namespace inside the type and find `hash_value` but on a compiler which doesn't support ADL `hash_value` won't be found. To make things worse, some compilers which do support ADL won't find a friend class defined inside the class.

So first move the member function out of the class:

```
namespace foo
{
    template <class T>
    class custom_type
    {
        T value;
    public:
        custom_type(T x) : value(x) {}

        std::size_t hash(custom_type x)
        {
            boost::hash<T> hasher;
            return hasher(value);
        }
    };

    template <class T>
    inline std::size_t hash_value(custom_type<T> x)
    {
        return x.hash();
    }
}
```

Unfortunately, I couldn't declare hash_value as a friend, as some compilers don't support template friends, so instead I declared a member function to calculate the hash, and called it from hash_value.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

For compilers which don't support ADL, hash_value needs to be defined in the boost namespace:

```
#ifdef BOOST_NO_ARGUMENT_DEPENDENT_LOOKUP
namespace boost
#else
namespace foo
#endif
{
    template <class T>
    std::size_t hash_value(foo::custom_type<T> x)
    {
        return x.hash();
    }
}
```

Full code for this example is at /libs/functional/hash/examples/portable.cpp.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

# Disabling The Extensions

While `boost::hash`'s extensions are generally useful, you might want to turn them of in order to check that your code will work with other implementations of TR1. To do this define the macro `BOOST_HASH_NO_EXTENSIONS`. When this macro is defined, only the specialisations detailed in TR1 will be declared. But, if you later undefine the macro and include <`boost/functional/hash.hpp`> then the non-specialised form will be defined - activating the extensions.

It is strongly recommended that you never undefine the macro - and only define it so that it applies to the complete translation unit, either by defining it at the beginning of the main source file or, preferably, by using a compiler switch or preference. And you really should never define it in header files.

If you are writing a library which has code in the header which requires the extensions, then the best action is to tell users not to define the macro. Their code won't *require* the macro.

Translation units that are compiled with the macro defined will link with units that were compiled without it. This feature has been designed to avoid ODR violations.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

# Change Log

## Boost 1.33.0

- Initial Release

## Boost 1.33.1

- Fixed the points example, as pointed out by .

## Boost 1.34.0

- Use declarations for standard classes, so that the library doesn't need to include all of their headers
- Deprecated the `<boost/functional/hash/*.hpp>` headers. Now a single header, <`boost/functional/hash.hpp`> is used.
- Add support for the `BOOST_HASH_NO_EXTENSIONS` macro, which disables the extensions to TR1.
- Minor improvements to the hash functions for floating point numbers.
- Update the portable example to hopefully be more generally portable.

## Boost 1.34.1

- Ticket 952: Suppress incorrect 64-bit warning on Visual C++.

## Boost 1.35.0

- Support for `long long`, `std::complex`.
- Improved algorithm for hashing floating point numbers:
    - Improved portablity, as described by Daniel Krügler in a post to the boost users list.
    - Fits more information into each combine loop, which can reduce the the number of times combine is called and hopefully give a better quality hash function.
    - Improved the algorithm for hashing floating point numbers.
    - On Cygwin use a binary hash function for floating point numbers, as Cygwin doesn't have decent floating point functions for `long double`.
    - Never uses `fpclass` which doesn't support `long double`.
    - Ticket 1064: Removed unnecessary use of `errno`.
- Explicitly overload for more built in types.
- Minor improvements to the documentation.
- A few bug and warning fixes:
    - Ticket 1509: Suppress another Visual C++ warning.
    - Some workarounds for the Sun compilers.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

# Boost 1.36.0

- Stop using OpenBSD's dodgy `std::numeric_limits`.
- Using the boost typedefs for `long long` and `unsigned long long`.
- Move the extensions into their own header.

# Boost 1.37.0

- Ticket 2264: In Visual C++, always use C99 float functions for `long double` and `float` as the C++ overloads aren't always availables.

# Boost 1.38.0

- Changed the warnings in the deprecated headers from 1.34.0 to errors. These will be removed in a future version of Boost.
- Moved detail headers out of `boost/functional/detail`, since they are part of functional/hash, not functional. `boost/functional/detail/container_fwd.hpp` has been moved to `boost/detail/container_fwd.hpp` as it's used outside of this library, the others have been moved to `boost/functional/hash/detail`.

# Boost 1.39.0

- Move the hash_fwd.hpp implementation into the hash subdirectory, leaving a forwarding header in the old location. You should still use the old location, the new location is mainly for implementation and possible modularization.
- Ticket 2412: Removed deprecated headers.
- Ticket 2957: Fix configuration for vxworks.

# Boost 1.40.0

- Automatically configure the float functions using template metaprogramming instead of trying to configure every possibility manually.
- Workaround for when STLport doesn't support long double.

# Boost 1.42.0

- Reduce the number of warnings for Visual C++ warning level 4.
- Some code formatting changes to fit lines into 80 characters.
- Rename an internal namespace.

# Boost 1.43.0

- Ticket 3866: Don't foward declare containers when using gcc's parallel library, allow user to stop forward declaration by defining the `BOOST_DETAIL_NO_CONTAINER_FWD` macro.
- Ticket 4038: Avoid hashing 0.5 and 0 to the same number.
- Stop using deprecated `BOOST_HAS_*` macros.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

# Boost 1.44.0

- Add option to prevent implicit conversions when calling `hash_value` by defining `BOOST_HASH_NO_IMPLICIT_CASTS`. When using `boost::hash` for a type that does not have `hash_value` declared but does have an implicit conversion to a type that does, it would use that implicit conversion to hash it. Which can sometimes go very wrong, e.g. using a conversion to bool and only hashing to 2 possible values. Since fixing this is a breaking change and was only approached quite late in the release cycle with little discussion it's opt-in for now. This, or something like it, will become the default in a future version.

# Boost 1.46.0

- Avoid warning due with gcc's `-Wconversion` flag.

# Boost 1.50.0

- Ticket 6771: Avoid gcc's `-Wfloat-equal` warning.
- Ticket 6806: Support `std::array` and `std::tuple` when available.
- Add deprecation warning to the long deprecated `boost/functional/detail/container_fwd.hpp`.

# Boost 1.51.0

- Support the standard smart pointers.
- `hash_value` now implemented using SFINAE to avoid implicit casts to built in types when calling it.
- Updated to use the new config macros.

# Boost 1.52.0

- Restore `enum` support, which was accidentally removed in the last version.
- New floating point hasher - will hash the binary representation on more platforms, which should be faster.

# Boost 1.53.0

- Add support for `boost::int128_type` and `boost::uint128_type` where available - currently only `__int128` and `unsigned __int128` on some versions of gcc.
- On platforms that are known to have the standard floating point functions, don't use automatic detection - which can break if there are ambiguous overloads.
- Fix undefined behaviour when using the binary float hash (Thomas Heller).

# Boost 1.54.0

- Ticket 7957: Fixed a typo.

# Boost 1.55.0

- Simplify a SFINAE check so that it will hopefully work on Sun 5.9 (#8822).
- Suppress Visual C++ infinite loop warning (#8568).

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

# Boost 1.56.0

- Removed some Visual C++ 6 workarounds.
- Ongoing work on improving `hash_combine`. This changes the combine function which was previously defined in the reference documentation.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

# Rationale

The rationale can be found in the original design [1].

## Quality of the hash function

Many hash functions strive to have little correlation between the input and output values. They attempt to uniformally distribute the output values for very similar inputs. This hash function makes no such attempt. In fact, for integers, the result of the hash function is often just the input value. So similar but different input values will often result in similar but different output values. This means that it is not appropriate as a general hash function. For example, a hash table may discard bits from the hash function resulting in likely collisions, or might have poor collision resolution when hash values are clustered together. In such cases this hash function will preform poorly.

But the standard has no such requirement for the hash function, it just requires that the hashes of two different values are unlikely to collide. Containers or algorithms designed to work with the standard hash function will have to be implemented to work well when the hash function's output is correlated to its input. Since they are paying that cost a higher quality hash function would be wasteful.

For other use cases, if you do need a higher quality hash function, then neither the standard hash function or `boost::hash` are appropriate. There are several options available. One is to use a second hash on the output of this hash function, such as Thomas Wang's hash function. This this may not work as well as a hash algorithm tailored for the input.

For strings there are several fast, high quality hash functions available (for example MurmurHash3 and Google's CityHash), although they tend to be more machine specific. These may also be appropriate for hashing a binary representation of your data - providing that all equal values have an equal representation, which is not always the case (e.g. for floating point values).

---

[1] issue 6.18 of the Library Extension Technical Report Issues List (page 63)

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

# Reference

For the full specification, see section 6.3 of the C++ Standard Library Technical Report and issue 6.18 of the Library Extension Technical Report Issues List (page 63).

## Header <boost/functional/hash.hpp>

Defines `boost::hash`, and helper functions.

```
namespace boost {
  template<typename T> struct hash;

  template<> struct hash<bool>;
  template<> struct hash<char>;
  template<> struct hash<signed char>;
  template<> struct hash<unsigned char>;
  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<> struct hash<std::string>;
  template<> struct hash<std::wstring>;
  template<typename T> struct hash<T*>;
  template<> struct hash<std::type_index>;

  // Support functions (Boost extension).
  template<typename T> void hash_combine(size_t &, T const&);
  template<typename It> std::size_t hash_range(It, It);
  template<typename It> void hash_range(std::size_t&, It, It);

  // Overloadable hash implementation (Boost extension).
  std::size_t hash_value(bool);
  std::size_t hash_value(char);
  std::size_t hash_value(signed char);
  std::size_t hash_value(unsigned char);
  std::size_t hash_value(wchar_t);
  std::size_t hash_value(short);
  std::size_t hash_value(unsigned short);
  std::size_t hash_value(int);
  std::size_t hash_value(unsigned int);
  std::size_t hash_value(long);
  std::size_t hash_value(unsigned long);
  std::size_t hash_value(long long);
  std::size_t hash_value(unsigned long long);
  std::size_t hash_value(float);
  std::size_t hash_value(double);
  std::size_t hash_value(long double);
  template<typename T> std::size_t hash_value(T* const&);
  template<typename T, unsigned N> std::size_t hash_value(T (&val)[N]);
  template<typename T, unsigned N> std::size_t hash_value(const T (&val)[N]);
  template<typename Ch, typename A>
    std::size_t hash_value(std::basic_string<Ch, std::char_traits<Ch>, A> const&);
  template<typename A, typename B>
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
    std::size_t hash_value(std::pair<A, B> const&);
  template<typename T, typename A>
    std::size_t hash_value(std::vector<T, A> const&);
  template<typename T, typename A>
    std::size_t hash_value(std::list<T, A> const&);
  template<typename T, typename A>
    std::size_t hash_value(std::deque<T, A> const&);
  template<typename K, typename C, typename A>
    std::size_t hash_value(std::set<K, C, A> const&);
  template<typename K, typename C, typename A>
    std::size_t hash_value(std::multiset<K, C, A> const&);
  template<typename K, typename T, typename C, typename A>
    std::size_t hash_value(std::map<K, T, C, A> const&);
  template<typename K, typename T, typename C, typename A>
    std::size_t hash_value(std::multimap<K, T, C, A> const&);
  template<typename T> std::size_t hash_value(std::complex<T> const&);
  std::size_t hash_value(std::type_index);
  template<typename T, std::size_t N>
    std::size_t hash_value(std::array<T, N> const&);
  template<typename... T> std::size_t hash_value(std::tuple<T...>);
}
```

## Support functions (Boost extension).

1. 
```
template<typename T> void hash_combine(size_t & seed, T const& v);
```

Called repeatedly to incrementally create a hash value from several variables.

Effects: Updates `seed` with a new hash value generated by combining it with the result of `hash_value [19](v)`. Will always produce the same result for the same combination of `seed` and `hash_value [19](v)` during the single run of a program.

Notes: hash_value [19] is called without qualification, so that overloads can be found via ADL.

This is an extension to TR1

Forward declared in `<boost/functional/hash_fwd.hpp>`

This hash function is not intended for general use, and isn't guaranteed to be equal during separate runs of a program - so please don't use it for any persistent storage or communication.

Throws: Only throws if hash_value [19](T) throws. Strong exception safety, as long as hash_value [19](T) also has strong exception safety.

2. 
```
template<typename It> std::size_t hash_range(It first, It last);
template<typename It> void hash_range(std::size_t& seed, It first, It last);
```

Calculate the combined hash value of the elements of an iterator range.

Effects: For the two argument overload:

```

size_t seed = 0;

for(; first != last; ++first)
{
    hash_combine(seed, *first);
}

return seed;
```

For the three arguments overload:

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
for(; first != last; ++first)
{
    hash_combine(seed, *first);
}
```

Notes: `hash_range` is sensitive to the order of the elements so it wouldn't be appropriate to use this with an unordered container.

This is an extension to TR1

Forward declared in `<boost/functional/hash_fwd.hpp>`

This hash function is not intended for general use, and isn't guaranteed to be equal during separate runs of a program - so please don't use it for any persistent storage or communication.

Throws: Only throws if `hash_value` [19]`(std::iterator_traits<It>::value_type)` throws. `hash_range(std::size_t&, It, It)` has basic exception safety as long as `hash_value` [19]`(std::iterator_traits<It>::value_type)` has basic exception safety.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

## Overloadable hash implementation (Boost extension).

1.

```
std::size_t hash_value(bool val);
std::size_t hash_value(char val);
std::size_t hash_value(signed char val);
std::size_t hash_value(unsigned char val);
std::size_t hash_value(wchar_t val);
std::size_t hash_value(short val);
std::size_t hash_value(unsigned short val);
std::size_t hash_value(int val);
std::size_t hash_value(unsigned int val);
std::size_t hash_value(long val);
std::size_t hash_value(unsigned long val);
std::size_t hash_value(long long val);
std::size_t hash_value(unsigned long long val);
std::size_t hash_value(float val);
std::size_t hash_value(double val);
std::size_t hash_value(long double val);
template<typename T> std::size_t hash_value(T* const& val);
template<typename T, unsigned N> std::size_t hash_value(T (&val)[N]);
template<typename T, unsigned N> std::size_t hash_value(const T (&val)[N]);
template<typename Ch, typename A>
  std::size_t hash_value(std::basic_string<Ch, std::char_traits<Ch>, A> const& val);
template<typename A, typename B>
  std::size_t hash_value(std::pair<A, B> const& val);
template<typename T, typename A>
  std::size_t hash_value(std::vector<T, A> const& val);
template<typename T, typename A>
  std::size_t hash_value(std::list<T, A> const& val);
template<typename T, typename A>
  std::size_t hash_value(std::deque<T, A> const& val);
template<typename K, typename C, typename A>
  std::size_t hash_value(std::set<K, C, A> const& val);
template<typename K, typename C, typename A>
  std::size_t hash_value(std::multiset<K, C, A> const& val);
template<typename K, typename T, typename C, typename A>
  std::size_t hash_value(std::map<K, T, C, A> const& val);
template<typename K, typename T, typename C, typename A>
  std::size_t hash_value(std::multimap<K, T, C, A> const& val);
template<typename T> std::size_t hash_value(std::complex<T> const& val);
std::size_t hash_value(std::type_index val);
template<typename T, std::size_t N>
  std::size_t hash_value(std::array<T, N> const& val);
template<typename... T> std::size_t hash_value(std::tuple<T...> val);
```

Implementation of the hash function.

Generally shouldn't be called directly by users, instead they should use `boost::hash`, `boost::hash_range [17]` or `boost::hash_combine` which call `hash_value` without namespace qualification so that overloads for custom types are found via ADL.

Notes: This is an extension to TR1

This hash function is not intended for general use, and isn't guaranteed to be equal during separate runs of a program - so please don't use it for any persistent storage or communication.

Throws: Only throws if a user supplied version of `hash_value [19]` throws for an element of a container, or one of the types stored in a pair.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Returns:

| **Types** | **Returns** |
|---|---|
| `bool`, `char`, `signed char`, `unsigned char`, `wchar_t`, `short`, `unsigned short`, `int`, `unsigned int`, `long`, `unsigned long` | `val` |
| `long long`, `unsigned long long` | `val when abs(val) <= std::numeric_limits<std::size_t>::max()`. |
| `float`, `double`, `long double` | An unspecified value, except that equal arguments shall yield the same result. |
| `T*` | An unspecified value, except that equal arguments shall yield the same result. |
| `T val[N]`, `const T val[N]` | `hash_range(val, val+N)` |
| `std::basic_string<Ch, std::char_traits<Ch>, A>`, `std::vector<T, A>`, `std::list<T, A>`, `std::deque<T, A>`, `std::set<K, C, A>`, `std::multiset<K, C, A>`, `std::map<K, T, C, A>`, `std::multimap<K, T, C, A>`, `std::array<T, N>` | `hash_range(val.begin(), val.end())` |
| `std::pair<A, B>` | `size_t seed = 0;`<br>`hash_combine(seed, val.first);`<br>`hash_combine(seed, val.second);`<br>`return seed;` |
| `std::tuple<T...>` | `size_t seed = 0;`<br>`hash_combine(seed, get<0>(val));`<br>`hash_combine(seed, get<1>(val));`<br>`// ....`<br>`return seed;` |
| `std::complex<T>` | When `T` is a built in type and `val.imag() == 0`, the result is equal to `hash_value(val.real())`. Otherwise an unspecified value, except that equal arguments shall yield the same result. |
| `std::type_index` | `val.hash_code()` |

## Struct template hash

boost::hash — A TR1 compliant hash function object.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

# Synopsis

```
// In header: <boost/functional/hash.hpp>

template<typename T>
struct hash : public std::unary_function<T, std::size_t> {
  std::size_t operator()(T const&) const;
};
```

## Description

```
std::size_t operator()(T const& val) const;
```

Returns:

```
hash_value [19](val)
```

Notes: The call to `hash_value [19]` is unqualified, so that custom overloads can be found via argument dependent lookup.

This is not defined when the macro `BOOST_HASH_NO_EXTENSIONS` is defined. The specializations are still defined, so only the specializations required by TR1 are defined.

Forward declared in `<boost/functional/hash_fwd.hpp>`

This hash function is not intended for general use, and isn't guaranteed to be equal during separate runs of a program - so please don't use it for any persistent storage or communication.

Throws: Only throws if `hash_value [19](T)` throws.

## Struct hash<bool>

boost::hash<bool>

# Synopsis

```
// In header: <boost/functional/hash.hpp>


struct hash<bool> {
  std::size_t operator()(bool) const;
};
```

## Description

```
std::size_t operator()(bool val) const;
```

Returns: Unspecified in TR1, except that equal arguments yield the same result.

hash_value [19](val) in Boost.

Throws: Doesn't throw

## Struct hash<char>

boost::hash<char>

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

# Synopsis

```
// In header: <boost/functional/hash.hpp>


struct hash<char> {
  std::size_t operator()(char) const;
};
```

## Description

```
std::size_t operator()(char val) const;
```

Returns: Unspecified in TR1, except that equal arguments yield the same result.

hash_value [19](val) in Boost.

Throws: Doesn't throw

## Struct hash<signed char>

boost::hash<signed char>

# Synopsis

```
// In header: <boost/functional/hash.hpp>


struct hash<signed char> {
  std::size_t operator()(signed char) const;
};
```

## Description

```
std::size_t operator()(signed char val) const;
```

Returns: Unspecified in TR1, except that equal arguments yield the same result.

hash_value [19](val) in Boost.

Throws: Doesn't throw

## Struct hash<unsigned char>

boost::hash<unsigned char>

# Synopsis

```
// In header: <boost/functional/hash.hpp>


struct hash<unsigned char> {
  std::size_t operator()(unsigned char) const;
};
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

### Description

```
std::size_t operator()(unsigned char val) const;
```

Returns: Unspecified in TR1, except that equal arguments yield the same result.

hash_value [19](val) in Boost.

Throws: Doesn't throw

## Struct hash<wchar_t>

boost::hash<wchar_t>

# Synopsis

```
// In header: <boost/functional/hash.hpp>


struct hash<wchar_t> {
  std::size_t operator()(wchar_t) const;
};
```

### Description

```
std::size_t operator()(wchar_t val) const;
```

Returns: Unspecified in TR1, except that equal arguments yield the same result.

hash_value [19](val) in Boost.

Throws: Doesn't throw

## Struct hash<short>

boost::hash<short>

# Synopsis

```
// In header: <boost/functional/hash.hpp>


struct hash<short> {
  std::size_t operator()(short) const;
};
```

### Description

```
std::size_t operator()(short val) const;
```

Returns: Unspecified in TR1, except that equal arguments yield the same result.

hash_value [19](val) in Boost.

Throws: Doesn't throw

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

## Struct hash<unsigned short>

boost::hash<unsigned short>

# Synopsis

```
// In header: <boost/functional/hash.hpp>


struct hash<unsigned short> {
  std::size_t operator()(unsigned short) const;
};
```

### Description

```
std::size_t operator()(unsigned short val) const;
```

| | |
|---|---|
| Returns: | Unspecified in TR1, except that equal arguments yield the same result.<br>hash_value [19](val) in Boost. |
| Throws: | Doesn't throw |

## Struct hash<int>

boost::hash<int>

# Synopsis

```
// In header: <boost/functional/hash.hpp>


struct hash<int> {
  std::size_t operator()(int) const;
};
```

### Description

```
std::size_t operator()(int val) const;
```

| | |
|---|---|
| Returns: | Unspecified in TR1, except that equal arguments yield the same result.<br>hash_value [19](val) in Boost. |
| Throws: | Doesn't throw |

## Struct hash<unsigned int>

boost::hash<unsigned int>

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

# Synopsis

```
// In header: <boost/functional/hash.hpp>


struct hash<unsigned int> {
  std::size_t operator()(unsigned int) const;
};
```

### Description

```
std::size_t operator()(unsigned int val) const;
```

| | |
|---|---|
| Returns: | Unspecified in TR1, except that equal arguments yield the same result.<br>hash_value [19](val) in Boost. |
| Throws: | Doesn't throw |

## Struct hash<long>

boost::hash<long>

# Synopsis

```
// In header: <boost/functional/hash.hpp>


struct hash<long> {
  std::size_t operator()(long) const;
};
```

### Description

```
std::size_t operator()(long val) const;
```

| | |
|---|---|
| Returns: | Unspecified in TR1, except that equal arguments yield the same result.<br>hash_value [19](val) in Boost. |
| Throws: | Doesn't throw |

## Struct hash<unsigned long>

boost::hash<unsigned long>

# Synopsis

```
// In header: <boost/functional/hash.hpp>


struct hash<unsigned long> {
  std::size_t operator()(unsigned long) const;
};
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

### Description

```
std::size_t operator()(unsigned long val) const;
```

Returns: Unspecified in TR1, except that equal arguments yield the same result.

hash_value [19](val) in Boost.

Throws: Doesn't throw

## Struct hash<long long>

boost::hash<long long>

# Synopsis

```
// In header: <boost/functional/hash.hpp>


struct hash<long long> {
  std::size_t operator()(long long) const;
};
```

### Description

```
std::size_t operator()(long long val) const;
```

Returns: Unspecified in TR1, except that equal arguments yield the same result.

hash_value [19](val) in Boost.

Throws: Doesn't throw

## Struct hash<unsigned long long>

boost::hash<unsigned long long>

# Synopsis

```
// In header: <boost/functional/hash.hpp>


struct hash<unsigned long long> {
  std::size_t operator()(unsigned long long) const;
};
```

### Description

```
std::size_t operator()(unsigned long long val) const;
```

Returns: Unspecified in TR1, except that equal arguments yield the same result.

hash_value [19](val) in Boost.

Throws: Doesn't throw

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

## Struct hash<float>

boost::hash<float>

# Synopsis

```
// In header: <boost/functional/hash.hpp>


struct hash<float> {
  std::size_t operator()(float) const;
};
```

### Description

```
std::size_t operator()(float val) const;
```

Returns: Unspecified in TR1, except that equal arguments yield the same result.

hash_value [19](val) in Boost.

Throws: Doesn't throw

## Struct hash<double>

boost::hash<double>

# Synopsis

```
// In header: <boost/functional/hash.hpp>


struct hash<double> {
  std::size_t operator()(double) const;
};
```

### Description

```
std::size_t operator()(double val) const;
```

Returns: Unspecified in TR1, except that equal arguments yield the same result.

hash_value [19](val) in Boost.

Throws: Doesn't throw

## Struct hash<long double>

boost::hash<long double>

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

# Synopsis

```
// In header: <boost/functional/hash.hpp>


struct hash<long double> {
  std::size_t operator()(long double) const;
};
```

### Description

```
std::size_t operator()(long double val) const;
```

Returns: Unspecified in TR1, except that equal arguments yield the same result.

hash_value [19](val) in Boost.

Throws: Doesn't throw

## Struct hash<std::string>

boost::hash<std::string>

# Synopsis

```
// In header: <boost/functional/hash.hpp>


struct hash<std::string> {
  std::size_t operator()(std::string const&) const;
};
```

### Description

```
std::size_t operator()(std::string const& val) const;
```

Returns: Unspecified in TR1, except that equal arguments yield the same result.

hash_value [19](val) in Boost.

Throws: Doesn't throw

## Struct hash<std::wstring>

boost::hash<std::wstring>

# Synopsis

```
// In header: <boost/functional/hash.hpp>


struct hash<std::wstring> {
  std::size_t operator()(std::wstring const&) const;
};
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

### Description

```
std::size_t operator()(std::wstring const& val) const;
```

Returns: Unspecified in TR1, except that equal arguments yield the same result.

hash_value [19](val) in Boost.

Throws: Doesn't throw

## Struct template hash<T*>

boost::hash<T*>

# Synopsis

```
// In header: <boost/functional/hash.hpp>

template<typename T>
struct hash<T*> {
  std::size_t operator()(T*) const;
};
```

### Description

```
std::size_t operator()(T* val) const;
```

Returns: Unspecified in TR1, except that equal arguments yield the same result.

Throws: Doesn't throw

## Struct hash<std::type_index>

boost::hash<std::type_index>

# Synopsis

```
// In header: <boost/functional/hash.hpp>


struct hash<std::type_index> {
  std::size_t operator()(std::type_index) const;
};
```

### Description

```
std::size_t operator()(std::type_index val) const;
```

Returns: `val.hash_code()`

Throws: Doesn't throw

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

# Links

**A Proposal to Add Hash Tables to the Standard Library** http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2003/n1456.html The hash table proposal explains much of the design. The hash function object is discussed in Section D.

**The C++ Standard Library Technical Report.** http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1836.pdf Contains the hash function specification in section 6.3.2.

**Library Extension Technical Report Issues List.** http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1837.pdf The library implements the extension described in Issue 6.18, pages 63-67.

**Methods for Identifying Versioned and Plagiarised Documents** Timothy C. Hoad, Justin Zobel http://www.cs.rmit.edu.au/~jz/fulltext/jasist-tch.pdf Contains the hash function that `boost::hash_combine` is based on.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

# Acknowledgements

This library is based on the design by Peter Dimov. During the initial development Joaquín M López Muñoz made many useful suggestions and contributed fixes.

The formal review was managed by Thorsten Ottosen, and the library reviewed by: David Abrahams, Alberto Barbati, Topher Cooper, Caleb Epstein, Dave Harris, Chris Jefferson, Bronek Kozicki, John Maddock, Tobias Swinger, Jaap Suter, Rob Stewart and Pavel Vozenilek. Since then, further constructive criticism has been made by Daniel Krügler, Alexander Nasonov and .

The implementation of the hash function for pointers is based on suggestions made by Alberto Barbati and Dave Harris. Dave Harris also suggested an important improvement to `boost::hash_combine` that was taken up.

Some useful improvements to the floating point hash algorithm were suggested by Daniel Krügler.

The original implementation came from Jeremy B. Maitin-Shepard's hash table library, although this is a complete rewrite.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/