"...one of the most highly regarded and expertly designed C++ library projects in the world."
— Herb Sutter (https://herbsutter.com) and Andrei Alexandrescu (http://en.wikipedia.org/wiki/Andrei_Alexandrescu), C++ Coding Standards (https://books.google.com/books/about/C++_Coding_Standards.html?id=mmjVIC6WolgC)

Boost.ContainerHash

Table of Contents

Introduction

`boost::hash` is an enhanced implementation of the hash function (https://en.wikipedia.org/wiki/Hash_function) object specified by C++11 as `std::hash`. It is the default hash function for Boost.Unordered, Boost.Intrusive's unordered associative containers, Boost.MultiIndex's hash indices, and Boost.Bimap's `unordered_set_of`.

Out of the box, `boost::hash` supports

- standard integral types (integers, character types, and `bool`);
- standard floating point types (`float`, `double`, `long double`);
- pointers (to objects and to functions, but not pointers to members) and `nullptr`;
- enumeration types;
- C arrays;
- `std::complex`;
- tuple-like types, such as `std::pair`, `std::tuple`, and user-defined types that specialize `std::tuple_size` and provide `get<I>`;
- sequence-like types, both standard and user-defined (sequence-like types have `begin()` and `end()` member functions returning iterators);
- unordered sequences, standard or user-defined (sequences for which the hash value does not depend on the element order, such as `std::unordered_set` and `std::unordered_map`);
- described structs and classes — ones that have been annotated with the `BOOST_DESCRIBE_STRUCT` or `BOOST_DESCRIBE_CLASS` macros from Boost.Describe;
- `std::unique_ptr`, `std::shared_ptr`;
- `std::type_index`;
- `std::error_code`, `std::error_condition`;
- `std::optional`;
- `std::variant`, `std::monostate`.

`boost::hash` is extensible; it's possible for a user-defined type `X` to make iself hashable via `boost::hash<X>` by defining an appropriate overload of the function `hash_value`. Many, if not most, Boost types already contain the necessary support.

`boost::hash` meets the requirements for `std::hash` specified in the C++11 standard, namely, that for two different input values their corresponding hash values are either guaranteed to be distinct, or the probability of their being the same (a hash collision) is small. Standard unordered containers, and the hash-based Boost containers, are designed to work well with such hash functions.

`boost::hash` does not meet the stronger requirements often placed on hash functions in a more general context. In particular, the hash function is not cryptographic, is not collision-resistant against a determined adversary, and does not necessarily possess good "avalanche" properties; that is, small (single bit) perturbations in the input do not necessarily result in large (half bits changing) perturbations in the output.

In particular, `boost::hash` has traditionally been the identity function for all integral types that fit into `std::size_t`, because this guarantees lack of collisions and is as fast as possible.

Recent Changes

Boost 1.84.0

- C++03 is no longer supported.

Boost 1.82.0

- Added an overload of `hash_value` for `std::nullptr_t`.
- Added `is_tuple_like` and an overload of `hash_value` for tuple-like types.
- Changed string hashing to use `mulxp1_hash` (https://github.com/pdimov/mulxp_hash), improving both quality and speed. This changes the hash values for string-like types (ranges of `char`, `signed char`, `unsigned char`, `std::byte`, `char8_t`).

Boost 1.81.0

Major update.

- The specializations of `boost::hash` have been removed; it now always calls `hash_value`.
- Support for `BOOST_HASH_NO_EXTENSIONS` has been removed. The extensions are always enabled.
- All standard containers are now supported. This includes `std::forward_list` and the unordered associative containers.
- User-defined containers (types that have `begin()` and `end()` member functions that return iterators) are now supported out of the box.
- Described structs and classes (those annotated with `BOOST_DESCRIBE_STRUCT` or `BOOST_DESCRIBE_CLASS`) are now supported out of the box.
- `hash_combine` has been improved. This changes the hash values of composite (container and tuple) types and of scalar types bigger than `size_t`.
- The performance (and quality, as a result of the above change) of string hashing has been improved. `boost::hash` for strings now passes SMHasher in 64 bit mode.
- The documentation has been substantially revised to reflect the changes.

Boost 1.78.0

- Fixed `hash_combine` so that its behavior no longer depends on whether `size_t` is the exact same type as `boost::uint64_t` (which wasn't the case on macOS). This changes the hash values of composite (container and tuple) types on macOS.

Tutorial

When using a Boost container such as Boost.Unordered, you don't need to do anything to use `boost::hash` as it's the default. To find out how to use a user-defined type, read the section on extending boost::hash for user types.

If you wish to use `boost::hash` with the standard unordered associative containers, pass it as a template parameter:

```C++
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:

```C++
#include <boost/container_hash/hash.hpp>

int main()
{
    boost::hash<std::string> string_hash;
    std::size_t h = string_hash("Hash me");
}
```

or alternatively:

```C++
#include <boost/container_hash/hash.hpp>

int main()
{
    std::size_t h = boost::hash<std::string>()("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:

```C++
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::back_inserter(hashes),
        boost::hash<typename Container::value_type>());

    return hashes;
}
```

Extending boost::hash for User Types

`boost::hash` is implemented by calling the function `hash_value`. 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 user type, it will get called.

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

```C++
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`:

```C++
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:

```C++
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 examples/books.hpp and examples/books.cpp.

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 `id` so the hash function only hashes `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).

Combining Hash Values

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

```C++
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:

```C++
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` on the supplied element, and combines it with the seed.

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

Note that when using `boost::hash_combine` the order of the calls matters.

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

and

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

result in a different values in `seed`.

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

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

Since `hash_range` works by repeatedly invoking `hash_combine` on the elements of the range, the hash value will also be dependent on the element order.

If you are calculating a hash value for a range where the order of the data doesn't matter, such as `unordered_set`, you can use `boost::hash_unordered_range` instead.

```cpp
std::unordered_set<std::string> set;
std::size_t hash = boost::hash_unordered_range(set.begin(), set.end());
```

When writing template classes, you might not want to include the main `hash.hpp` header as it's quite an expensive include that brings in a lot of other headers, so instead you can include the `<boost/container_hash/hash_fwd.hpp>` header which forward declares `boost::hash`, `boost::hash_combine`, `boost::hash_range`, and `boost::hash_unordered_range`. 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 examples/template.hpp and examples/template.cpp.

To avoid including even `hash_fwd.hpp` - which still requires the contents of Boost.ContainerHash to be physically present - you are allowed to copy the declarations from `hash_fwd.hpp` (and only those) directly into your own header. This is a special exception guaranteed by the library; in general, you can't declare library functions, Boost or otherwise, without risk of breakage in a subsequent release.

Hashing User Types with Boost.Describe

Let's look at our `point` class again:

```cpp
class point
{
    int x;
    int y;

public:

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

If you're using C++14 or above, a much easier way to add support for `boost::hash` to `point` is by using Boost.Describe (and get an automatic definition of `operator==` for free):

```cpp
#include <boost/describe/class.hpp>
#include <boost/describe/operators.hpp>

class point
{
    int x;
    int y;

    BOOST_DESCRIBE_CLASS(point, (), (), (), (x, y))

public:

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

using boost::describe::operators::operator==;
using boost::describe::operators::operator!=;
```

(Full code for this example is at examples/point2.cpp.)

Since the `point` class has been annotated with `BOOST_DESCRIBE_CLASS`, the library can enumerate its members (and base classes) and automatically synthesize the appropriate `hash_value` overload for it, without us needing to do so.

Reference

<boost/container_hash/hash_fwd.hpp>

This header contains forward declarations for the library primitives. These declarations are guaranteed to be relatively stable, that is, best effort will be expended on their not changing from release to release, allowing their verbatim copy into user headers that do not wish to physically depend on Boost.ContainerHash.

```C++
namespace boost
{

namespace container_hash
{

template<class T> struct is_range;
template<class T> struct is_contiguous_range;
template<class T> struct is_unordered_range;
template<class T> struct is_described_class;
template<class T> struct is_tuple_like;

} // namespace container_hash

template<class T> struct hash;

template<class T> void hash_combine( std::size_t& seed, T const& v );

template<class It> void hash_range( std::size_t& seed, It first, It last );
template<class It> std::size_t hash_range( It first, It last );

template<class It> void hash_unordered_range( std::size_t& seed, It first, It last );
template<class It> std::size_t hash_unordered_range( It first, It last );

} // namespace boost
```

<boost/container_hash/hash.hpp>

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

```C++
namespace boost
{

template<class T> struct hash;

template<class T> void hash_combine( std::size_t& seed, T const& v );

template<class It> void hash_range( std::size_t& seed, It first, It last );
template<class It> std::size_t hash_range( It first, It last );

template<class It> void hash_unordered_range( std::size_t& seed, It first, It last );
template<class It> std::size_t hash_unordered_range( It first, It last );

// Enabled only when T is an integral type
template<class T>
  std::size_t hash_value( T v );

// Enabled only when T is an enumeration type
template<class T>
  std::size_t hash_value( T v );

// Enabled only when T is a floating point type
template<class T>
  std::size_t hash_value( T v );

template<class T>
  std::size_t hash_value( T* const& v );

template<class T, std::size_t N>
  std::size_t hash_value( T const (&v)[N] );

template<class T>
  std::size_t hash_value( std::complex<T> const& v );

template<class A, class B>
  std::size_t hash_value( std::pair<A, B> const& v );

// Enabled only when container_hash::is_tuple_like<T>::value is true
template<class T>
  std::size_t hash_value( T const& v );

// Enabled only when container_hash::is_range<T>::value is true
template<class T>
  std::size_t hash_value( T const& v );

// Enabled only when container_hash::is_contiguous_range<T>::value is true
template<class T>
  std::size_t hash_value( T const& v );

// Enabled only when container_hash::is_unordered_range<T>::value is true
template<class T>
  std::size_t hash_value( T const& v );

// Enabled only when container_hash::is_described_class<T>::value is true
template<class T>
  std::size_t hash_value( T const& v );

template<class T>
  std::size_t hash_value( std::shared_ptr<T> const& v );

template<class T, class D>
  std::size_t hash_value( std::unique_ptr<T, D> const& v );
```

```
std::size_t hash_value( std::type_index const& v );

std::size_t hash_value( std::error_code const& v );
std::size_t hash_value( std::error_condition const& v );

template<class T>
  std::size_t hash_value( std::optional<T> const& v );

std::size_t hash_value( std::monostate v );

template<class... T>
  std::size_t hash_value( std::variant<T...> const& v );

} // namespace boost
```

hash<T>

C++

```
template<class T> struct hash
{
    std::size_t operator()( T const& v ) const;
};
```

operator()

C++

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

Returns:

`hash_value(v)`.

Throws:

Only throws if `hash_value(v)` throws.

Remarks:

The call to `hash_value` is unqualified, so that user-supplied overloads will be found via argument dependent lookup.

hash_combine

C++

```
template<class T> void hash_combine( std::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 deterministically combining it with the result of `boost::hash<T>()(v)`.

Throws:

Only throws if `boost::hash<T>()(v)` throws. On exception, `seed` is not updated.

Remarks:

Equivalent to `seed = combine(seed, boost::hash<T>()(v))`, where `combine(s, v)` is a mixing function that takes two arguments of type `std::size_t` and returns `std::size_t`, with the following desirable properties:

1. For a constant `s`, when `v` takes all possible `size_t` values, `combine(s, v)` should also take all possible `size_t` values, producing a sequence that is close to random; that is, it should be a random permutation.

   This guarantees that for a given `seed`, `combine` does not introduce hash collisions when none were produced by `boost::hash<T>(v)`; that is, it does not lose information from the input. It also implies that `combine(s, v)`, as a function of `v`, has good avalanche properties; that is, small (e.g. single bit) perturbations in the input `v` lead to large perturbations in the return value (half of the output bits changing, on average).

2. For two different seeds `s1` and `s2`, `combine(s1, v)` and `combine(s2, v)`, treated as functions of `v`, should produce two different random permutations.

3. `combine(0, 0)` should not be 0. Since a common initial value of `seed` is zero, `combine(0, 0) == 0` would imply that applying `hash_combine` on any sequence of zeroes, regardless of length, will produce zero. This is undesirable, as it would lead to e.g. `std::vector<int>()` and `std::vector<int>(4)` to have the same hash value.

The current implementation uses the function `mix(s + 0x9e3779b9 + v)` as `combine(s, v)`, where `mix(x)` is a high quality mixing function that is a bijection over the `std::size_t` values, of the form

```
x ^= x >> k1;
x *= m1;
x ^= x >> k2;
x *= m2;
x ^= x >> k3;
```

where the constants `k1`, `k2`, `k3`, `m1`, `m2` are suitably chosen.

Note that `mix(0)` is 0. This is why we add the arbitrary constant `0x9e3779b9` to meet the third requirement above.

hash_range

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

Effects:

When `typename std::iterator_traits<It>::value_type` is not `char`, `signed char`, `unsigned char`, `std::byte`, or `char8_t`,

```
for( ; first != last; ++first )
{
    boost::hash_combine<typename std::iterator_traits<It>::value_type>( seed, *first );
}
```

Otherwise, bytes from `[first, last)` are coalesced and hashed in an unspecified manner. This is done in order to improve performance when hashing strings.

Remarks:

For chars, the current implementation uses `mulxp1_hash` (https://github.com/pdimov/mulxp_hash) when `std::size_t` is 64 bit, and `mulxp1_hash32` when it's 32 bit.

```
template<class It> std::size_t hash_range( It first, It last );
```

Effects:

```
size_t seed = 0;
boost::hash_range( seed, first, last );
return seed;
```

hash_unordered_range

```
template<class It> void hash_unordered_range( std::size_t& seed, It first, It last );
```

Effects:

Updates `seed` with the values of `boost::hash<typename std::iterator_traits<It>::value_type>()(*i)` for each `i` in `[first, last)`, such that the order of elements does not affect the final result.

```
template<class It> std::size_t hash_unordered_range( It first, It last );
```

Effects:

```
size_t seed = 0;
boost::hash_unordered_range( seed, first, last );
return seed;
```

hash_value

```
// Enabled only when T is an integral type
template<class T>
  std::size_t hash_value( T v );
```

Returns:

When the value of `v` fits into `std::size_t`, when `T` is an unsigned type, or into `ssize_t`, when `T` is a signed type, `static_cast<std::size_t>(v)`.

Otherwise, an unspecified value obtained by mixing the value bits of `v`.

```
// Enabled only when T is an enumeration type
template<class T>
  std::size_t hash_value( T v );
```

Returns:

`static_cast<std::size_t>(v)`.

Remarks:

`hash_value(std::to_underlying(v))` would be better, but C++03 compatibility mandates the current implementation.

```
// Enabled only when T is a floating point type
template<class T>
  std::size_t hash_value( T v );
```

Returns:

An unspecified value obtained by mixing the value bits of `v`.

Remarks:

When `sizeof(v) <= sizeof(std::size_t)`, the bits of `v` are returned as-is (except in the case of -0.0, which is treated as +0.0).

```C++
template<class T>
  std::size_t hash_value( T* const& v );
```

Returns:

An unspecified value derived from `reinterpret_cast<std::uintptr_t>(v)`.

```C++
template<class T, std::size_t N>
  std::size_t hash_value( T const (&v)[N] );
```

Returns:

`boost::hash_range( v, v + N )`.

```C++
template<class T>
  std::size_t hash_value( std::complex<T> const& v );
```

Returns:

An unspecified value derived from `boost::hash<T>()(v.real())` and `boost::hash<T>()(v.imag())` such that, if `v.imag() == 0`, the value is equal to `boost::hash<T>()(v.real())`.

Remarks:

A more straightforward implementation would just have used `hash_combine` on `v.real()` and `v.imag()`, but the historical guarantee that real-valued complex numbers should match the hash value of their real part precludes it.

This guarantee may be dropped in a future release, as it's of questionable utility.

```C++
template<class A, class B>
  std::size_t hash_value( std::pair<A, B> const& v );
```

Effects:

```C++
  std::size_t seed = 0;

  boost::hash_combine( seed, v.first );
  boost::hash_combine( seed, v.second );

  return seed;
```

```C++
// Enabled only when container_hash::is_tuple_like<T>::value is true
template<class T>
  std::size_t hash_value( T const& v );
```

Effects:

```C++
std::size_t seed = 0;

using std::get;

boost::hash_combine( seed, get<0>(v) );
boost::hash_combine( seed, get<1>(v) );
// ...
boost::hash_combine( seed, get<N-1>(v) );

return seed;
```

where `N` is `std::tuple_size<T>::value`.

Remarks:

This overload is only enabled when `container_hash::is_range<T>::value` is `false`.

```C++
// Enabled only when container_hash::is_range<T>::value is true
template<class T>
  std::size_t hash_value( T const& v );
```

Returns:

`boost::hash_range( v.begin(), v.end() )`.

Remarks:

This overload is only enabled when `container_hash::is_contiguous_range<T>::value` and `container_hash::is_unordered_range<T>::value` are both `false`.

It handles all standard containers that aren't contiguous or unordered, such as `std::deque`, `std::list`, `std::set`, `std::map`.

```C++
// Enabled only when container_hash::is_contiguous_range<T>::value is true
template<class T>
  std::size_t hash_value( T const& v );
```

Returns:

`boost::hash_range( v.data(), v.data() + v.size() )`.

Remarks:

This overload handles all standard contiguous containers, such as `std::string`, `std::vector`, `std::array`, `std::string_view`.

```C++
// Enabled only when container_hash::is_unordered_range<T>::value is true
template<class T>
  std::size_t hash_value( T const& v );
```

Returns:

`boost::hash_unordered_range( v.begin(), v.end() )`.

Remarks:

This overload handles the standard unordered containers, such as `std::unordered_set` and `std::unordered_map`.

```C++
// Enabled only when container_hash::is_described_class<T>::value is true
template<class T>
  std::size_t hash_value( T const& v );
```

Effects:

```C++
  std::size_t seed = 0;

  boost::hash_combine( seed, b1 );
  boost::hash_combine( seed, b2 );
  // ...
  boost::hash_combine( seed, bM );

  boost::hash_combine( seed, m1 );
  boost::hash_combine( seed, m2 );
  // ...
  boost::hash_combine( seed, mN );

  return seed;
```

where `bi` are the bases of `v` and `mi` are its members.

```C++
template<class T>
  std::size_t hash_value( std::shared_ptr<T> const& v );

template<class T, class D>
  std::size_t hash_value( std::unique_ptr<T, D> const& v );
```

Returns:

`boost::hash<T*>( v.get() )`.

```C++
std::size_t hash_value( std::type_index const& v );
```

Returns:

`v.hash_code()`.

```C++
std::size_t hash_value( std::error_code const& v );
std::size_t hash_value( std::error_condition const& v );
```

Effects:

```C++
  std::size_t seed = 0;

  boost::hash_combine( seed, v.value() );
  boost::hash_combine( seed, &v.category() );

  return seed;
```

```cpp
template<class T>
  std::size_t hash_value( std::optional<T> const& v );
```

Returns:

For a disengaged `v`, an unspecified constant value; otherwise, `boost::hash<T>()( *v )`.

```cpp
std::size_t hash_value( std::monostate v );
```

Returns:

An unspecified constant value.

```cpp
template<class... T>
  std::size_t hash_value( std::variant<T...> const& v );
```

Effects:

```cpp
  std::size_t seed = 0;

  boost::hash_combine( seed, v.index() );
  boost::hash_combine( seed, x );

  return seed;
```

where `x` is the currently contained value in `v`.

Throws:

`std::bad_variant_access` when `v.valueless_by_exception()` is `true`.

<boost/container_hash/is_range.hpp>

Defines the trait `boost::container_hash::is_range`.

```cpp
namespace boost
{

namespace container_hash
{

template<class T> struct is_range;

} // namespace container_hash

} // namespace boost
```

is_range<T>

```cpp
template<class T> struct is_range
{
    static constexpr bool value = /* see below */;
};
```

`is_range<T>::value` is `true` when, for a const value `x` of type `T`, `x.begin()` and `x.end()` return iterators of the same type `It` (such that `std::iterator_traits<It>` is a valid specialization.)

Users are allowed to specialize `is_range` for their types if the default behavior does not deduce the correct value.

<boost/container_hash/is_contiguous_range.hpp>

Defines the trait `boost::container_hash::is_contiguous_range`.

```
namespace boost
{

namespace container_hash
{

template<class T> struct is_contiguous_range;

} // namespace container_hash

} // namespace boost
```

is_contiguous_range<T>

```
template<class T> struct is_contiguous_range
{
    static constexpr bool value = /* see below */;
};
```

`is_contiguous_range<T>::value` is `true` when `is_range<T>::value` is `true` and when, for a const value `x` of type `T`, `x.data()` returns a pointer to a type that matches the `value_type` of the iterator returned by `x.begin()` and `x.end()`, and `x.size()` returns a value of an integral type.

Users are allowed to specialize `is_contiguous_range` for their types if the default behavior does not deduce the correct value.

<boost/container_hash/is_unordered_range.hpp>

Defines the trait `boost::container_hash::is_unordered_range`.

```
namespace boost
{

namespace container_hash
{

template<class T> struct is_unordered_range;

} // namespace container_hash

} // namespace boost
```

is_unordered_range<T>

```cpp
template<class T> struct is_unordered_range
{
    static constexpr bool value = /* see below */;
};
```

`is_unordered_range<T>::value` is `true` when `is_range<T>::value` is `true` and when `T::hasher` is a valid type.

Users are allowed to specialize `is_unordered_range` for their types if the default behavior does not deduce the correct value.

<boost/container_hash/is_described_class.hpp>

Defines the trait `boost::container_hash::is_described_class`.

```cpp
namespace boost
{

namespace container_hash
{

template<class T> struct is_described_class;

} // namespace container_hash

} // namespace boost
```

is_described_class<T>

```cpp
template<class T> struct is_described_class
{
    static constexpr bool value = /* see below */;
};
```

`is_described_class<T>::value` is `true` when `boost::describe::has_describe_bases<T>::value` is `true`, `boost::describe::has_describe_members<T>::value` is `true`, and `T` is not a union.

Users are allowed to specialize `is_described_class` for their types if the default behavior does not deduce the correct value.

<boost/container_hash/is_tuple_like.hpp>

Defines the trait `boost::container_hash::is_tuple_like`.

```cpp
namespace boost
{

namespace container_hash
{

template<class T> struct is_tuple_like;

} // namespace container_hash

} // namespace boost
```

is_tuple_like<T>

```C++
template<class T> struct is_tuple_like
{
    static constexpr bool value = /* see below */;
};
```

`is_tuple_like<T>::value` is `true` when `std::tuple_size<T>::value` is valid.

Users are allowed to specialize `is_tuple_like` for their types if the default behavior does not deduce the correct value.

Design and Implementation Notes

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 perform 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.

The hash_value Customization Point

The way one customizes the standard `std::hash` function object for user types is via a specialization. `boost::hash` chooses a different mechanism — an overload of a free function `hash_value` in the user namespace that is found via argument-dependent lookup.

Both approaches have their pros and cons. Specializing the function object is stricter in that it only applies to the exact type, and not to derived or convertible types. Defining a function, on the other hand, is easier and more convenient, as it can be done directly in the type definition as an `inline friend`.

The fact that overloads can be invoked via conversions did cause issues in an earlier iteration of the library that defined `hash_value` for all integral types separately, including `bool`. Especially under C++03, which doesn't have `explicit` conversion operators, some types were convertible to `bool` to allow their being tested in e.g. `if` statements, which caused them to hash to 0 or 1, rarely what one expects or wants.

This, however, was fixed by declaring the built-in `hash_value` overloads to be templates constrained on e.g. `std::is_integral` or its moral equivalent. This causes types convertible to an integral to no longer match, avoiding the problem.

Hash Value Stability

In general, the library does not promise that the hash values will stay the same from release to release (otherwise improvements would be impossible). However, historically values have been quite stable. Before release 1.81, the previous changes have been in 1.56 (a better `hash_combine`) and 1.78 (macOS-specific change to `hash_combine`.)

Code should generally not depend on specific hash values, but for those willing to take the risk of occasional breaks due to hash value changes, the library now has a test that checks hash values for a number of types against reference values (`test/hash_reference_values.cpp`), whose version history (https://github.com/boostorg/container_hash/commits/develop/test/hash_reference_values.cpp) can be used as a rough guide to when hash values have changed, and for what types.

hash_combine

The initial implementation of the library was based on Issue 6.18 of the Library Extension Technical Report Issues List (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1837.pdf) (pages 63-67) which proposed the following implementation of `hash_combine`:

```C++
template<class T>
void hash_combine(size_t & seed, T const & v)
{
    seed ^= hash_value(v) + (seed << 6) + (seed >> 2);
}
```

taken from the paper "Methods for Identifying Versioned and Plagiarised Documents (https://people.eng.unimelb.edu.au/jzobel/fulltext/jasist03thz.pdf)" by Timothy C. Hoad and Justin Zobel.

During the Boost formal review, Dave Harris pointed out that this suffers from the so-called "zero trap"; if `seed` is initially 0, and all the inputs are 0 (or hash to 0), `seed` remains 0 no matter how many input values are combined.

This is an undesirable property, because it causes containers of zeroes to have a zero hash value regardless of their sizes.

To fix this, the arbitrary constant `0x9e3779b9` (the golden ratio in a 32 bit fixed point representation) was added to the computation, yielding

```C++
template<class T>
void hash_combine(size_t & seed, T const & v)
{
    seed ^= hash_value(v) + 0x9e3779b9 + (seed << 6) + (seed >> 2);
}
```

This is what shipped in Boost 1.33, the first release containing the library.

This function was a reasonable compromise between quality and speed for its time, when the input consisted of `char`s, but it's less suitable for combining arbitrary `size_t` inputs.

In Boost 1.56, it was replaced by functions derived from Austin Appleby's MurmurHash2 hash function round (https://github.com/aappleby/smhasher/blob/61a0530f28277f2e850bfc39600ce61d02b518de/src/MurmurHash2.cpp#L57-L62).

In Boost 1.81, it was changed again — to the equivalent of `mix(seed + 0x9e3779b9 + hash_value(v))`, where `mix(x)` is a high quality mixing function that is a bijection over the `size_t` values, of the form

```C++
x ^= x >> k1;
x *= m1;
x ^= x >> k2;
x *= m2;
x ^= x >> k3;
```

This type of mixing function was originally devised by Austin Appleby as the "final mix" part of his MurmurHash3 hash function. He used

```C++
x ^= x >> 33;
x *= 0xff51afd7ed558ccd;
x ^= x >> 33;
x *= 0xc4ceb9fe1a85ec53;
x ^= x >> 33;
```

as the 64 bit function `fmix64` (https://github.com/aappleby/smhasher/blob/61a0530f28277f2e850bfc39600ce61d02b518de/src/MurmurHash2.cpp#L57-L62) and

```C++
x ^= x >> 16;
x *= 0x85ebca6b;
x ^= x >> 13;
x *= 0xc2b2ae35;
x ^= x >> 16;
```

as the 32 bit function `fmix32` (https://github.com/aappleby/smhasher/blob/61a0530f28277f2e850bfc39600ce61d02b518de/src/MurmurHash3.cpp#L68-L77).

Several improvements of the 64 bit function have been subsequently proposed, by David Stafford (https://zimbry.blogspot.com/2011/09/better-bit-mixing-improving-on.html), Pelle Evensen (https://mostlymangling.blogspot.com/2019/12/stronger-better-morer-moremur-better.html), and Jon Maiga (http://jonkagstrom.com/mx3/mx3_rev2.html). We currently use Jon Maiga's function

```C++
x ^= x >> 32;
x *= 0xe9846af9b1a615d;
x ^= x >> 32;
x *= 0xe9846af9b1a615d;
x ^= x >> 28;
```

Under 32 bit, we use a mixing function proposed by "TheIronBorn" in a Github issue (https://github.com/skeeto/hash-prospector/issues/19) in the repository (https://github.com/skeeto/hash-prospector) of Hash Prospector (https://nullprogram.com/blog/2018/07/31/) by Chris Wellons:

```C++
x ^= x >> 16;
x *= 0x21f0aaad;
x ^= x >> 15;
x *= 0x735a2d97;
x ^= x >> 15;
```

With this improved `hash_combine`, `boost::hash` for strings now passes the SMHasher test suite (https://github.com/aappleby/smhasher) by Austin Appleby (for a 64 bit `size_t`).

hash_range

The traditional implementation of `hash_range(seed, first, last)` has been

```C++
for( ; first != last; ++first )
{
    boost::hash_combine<typename std::iterator_traits<It>::value_type>( seed, *first );
}
```

(the explicit template parameter is needed to support iterators with proxy return types such as `std::vector<bool>::iterator`.)

This is logical, consistent and straightforward. In the common case where `typename std::iterator_traits<It>::value_type` is `char` — which it is in the common case of `boost::hash<std::string>` — this however leaves a lot of performance on the table, because processing each `char` individually is much less efficient than processing several in bulk.

In Boost 1.81, `hash_range` was changed to process elements of type `char`, `signed char`, `unsigned char`, `std::byte`, or `char8_t`, four of a time. A `uint32_t` is composed from `first[0]` to `first[3]`, and that `uint32_t` is fed to `hash_combine`.

In Boost 1.82, `hash_range` for these types was changed to use `mulxp1_hash` (https://github.com/pdimov/mulxp_hash). This improves both quality and speed of string hashing.

Note that `hash_range` has also traditionally guaranteed that the same element sequence yields the same hash value regardless of the iterator type. This property remains valid after the changes to `char` range hashing. `hash_range`, applied to the `char` sequence `{ 'a', 'b', 'c' }`, results in the same value whether the sequence comes from `char[3]`, `std::string`, `std::deque<char>`, or `std::list<char>`.

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
https://people.eng.unimelb.edu.au/jzobel/fulltext/jasist03thz.pdf

Contains the hash function that the initial implementation of `boost::hash_combine` was based on.

Performance in Practice of String Hashing Functions
M.V. Ramakrishna, J. Zobel
In Proc. Int. Conf. on Database Systems for Advanced Applications, pages 215-223, Melbourne, Australia, April 1997.
https://www.comp.nus.edu.sg/~lingtw/dasfaa_proceedings/DASFAA97/P215.pdf

Referenced in the above paper as the source of the hash function.

MurmurHash3 hash function source
Austin Appleby
https://github.com/aappleby/smhasher/blob/61a0530f28277f2e850bfc39600ce61d02b518de/src/MurmurHash3.cpp#L65-L90

Austin Appleby's 32 and 64 bit finalization mixing functions that introduced the "xmxmx" general form of a high quality bijective transformation that approximates a random permutation.

SMHasher hash function test suite
Austin Appleby
https://github.com/aappleby/smhasher

Contains a battery of tests for evaluating hash functions. The current 64 bit implementation of `boost::hash` for strings passes SMHasher. Previous iterations did not.

Better Bit Mixing - Improving on MurmurHash3's 64-bit Finalizer
David Stafford
https://zimbry.blogspot.com/2011/09/better-bit-mixing-improving-on.html

Describes the so-called "variant 13" mixing function, an improvement over `fmix64` from MurmurHash3, made famous by its adoption by the `splitmix64` random number generator (http://xorshift.di.unimi.it/splitmix64.c).

Stronger, better, morer, Moremur; a better Murmur3-type mixer
Pelle Evensen
https://mostlymangling.blogspot.com/2019/12/stronger-better-morer-moremur-better.html

Describes Moremur, an improvement over MurmurHash3 `fmix64` and Stafford "variant 13".

Improved mx3 and the RRC test
Jon Maiga
http://jonkagstrom.com/mx3/mx3_rev2.html

Contains another improvement over MurmurHash3 `fmix64` and "variant 13". This is what the current implementation of `boost::hash_combine` uses when `std::size_t` is 64 bits.

Prospecting for Hash Functions
Chris Wellons
https://nullprogram.com/blog/2018/07/31/

Describes Hash Prospector (https://github.com/skeeto/hash-prospector), a utility for discovering and evaluating mixing functions.

New best known functions
"TheIronBorn"
https://github.com/skeeto/hash-prospector/issues/19

Describes a good 32 bit mixing function, used by the current implementation of `boost::hash_combine` when `std::size_t` is 32 bits.

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 沈慧峰.

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.

The documentation was converted from Quickbook to AsciiDoc by Christian Mazakas.

Change Log

Boost 1.67.0

- Moved library into its own module, `container_hash`.
- Moved headers for new module name, now at: `<boost/container_hash/hash.hpp>`, `<boost/container_hash/hash_fwd.hpp>`, `<boost/container_hash/extensions.hpp>`.
- Added forwarding headers to support the old headers locations.
- Support `std::string_view`, `std::error_code`, `std::error_condition`, `std::optional`, `std::variant`, `std::monostate` where available.
- Update include paths from other Boost libraries.

- Manually write out tuple overloads, rather than using the preprocessor to generate them. Should improve usability, due to better error messages, and easier debugging.
- Fix tutorial example (#11017 (https://svn.boost.org/trac/boost/ticket/11017)).
- Quick fix for hashing `vector<bool>` when using libc++. Will try to introduce a more general fix in the next release.

Boost 1.66.0

- Avoid float comparison warning when using Clang - this workaround was already in place for GCC, and was used when Clang pretends to be GCC, but the warning was appearing when running Clang in other contexts.

Boost 1.65.0

- Support for `char16_t`, `char32_t`, `u16string`, `u32string`

Boost 1.64.0

- Fix for recent versions of Visual C++ which have removed `std::unary_function` and `std::binary_function` (#12353 (https://svn.boost.org/trac/boost/ticket/12353)).

Boost 1.63.0

- Fixed some warnings.
- Only define hash for `std::wstring` when we know we have a `wchar_t`. Otherwise there's a compile error as there's no overload for hashing the characters in wide strings (#8552 (https://svn.boost.org/trac/boost/ticket/8552)).

Boost 1.58.0

- Fixed strict aliasing violation (GitHub #3 (https://github.com/boostorg/container_hash/issues/3)).

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.

Boost 1.55.0

- Simplify a SFINAE check so that it will hopefully work on Sun 5.9 (#8822 (https://svn.boost.org/trac10/ticket/8822)).
- Suppress Visual C++ infinite loop warning (#8568 (https://svn.boost.org/trac10/ticket/8568)).

Boost 1.54.0

- Ticket 7957 (https://svn.boost.org/trac/boost/ticket/7957): Fixed a typo.

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 1.52.0

- 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 1.51.0

- 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 1.50.0

- Ticket 6771 (https://svn.boost.org/trac/boost/ticket/6771): Avoid gcc's `-Wfloat-equal` warning.
- Ticket 6806 (https://svn.boost.org/trac/boost/ticket/6806): Support `std::array` and `std::tuple` when available.
- Add deprecation warning to the long deprecated `boost/container_hash/detail/container_fwd.hpp`.

Boost 1.46.0

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

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 1.43.0

- Ticket 3866 (https://svn.boost.org/trac/boost/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 (https://svn.boost.org/trac/boost/ticket/4038): Avoid hashing `0.5` and `0` to the same number.
- Stop using deprecated `BOOST_HAS_*` macros.

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 1.40.0

- 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 1.39.0

- 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 (https://svn.boost.org/trac/boost/ticket/2412): Removed deprecated headers.
- Ticket 2957 (https://svn.boost.org/trac/boost/ticket/2957): Fix configuration for vxworks.

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/container_hash/detail`, since they are part of `functional/hash`, not `container_hash`. `boost/container_hash/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 1.37.0

- Ticket 2264 (http://svn.boost.org/trac/boost/ticket/2264): In Visual C++, always use C99 float functions for long double and float as the C++ overloads aren't always availables.

Boost 1.36.0

- 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 1.35.0

- 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 (http://lists.boost.org/boost-users/2005/08/13418.php).
  - 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 (http://svn.boost.org/trac/boost/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 (http://svn.boost.org/trac/boost/ticket/1509): Suppress another Visual C++ warning.
    - Some workarounds for the Sun compilers.

Boost 1.34.1

- Ticket 952 (https://svn.boost.org/trac10/ticket/952): Suppress incorrect 64-bit warning on Visual C++.

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 1.33.1

- Fixed the points example, as pointed out by 沈慧峰.

Boost 1.33.0

- Initial Release

Copyright and License