Boost.Functional/Hash

Daniel James

Table of Contents

Introduction 1
Tutorial 1
Extending boost::hash for a custom data type 2
Combining hash values 3
Portability 5
Reference 6
Links 20
Acknowledgements 20

Introduction

`boost::hash` is an implementation of the hash function object specified by the Technical Report. It is intended for use as the default hash function for unordered associative containers, and the Boost Multi-Index Containers Library's hash indexes.

As it is compliant with the Technical Report, 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, this adds support for:

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

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<std::vector<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/hash/hash.hpp>

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

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

If you wish to make use of the extensions, you will need to include the appropriate header (see the reference documentation for the full list).

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

int main()
{
    boost::hash<std::pair<int, int> > pair_hash;

    std::size_t h = pair_hash(std::make_pair(1, 2));
}
```

Or alternatively, include `<boost/hash.hpp>` for the full library.

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;
}
```

Extending boost::hash for a custom data type

`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 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.

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 the should generate different hash values. In this object equality was based just on the id, if it was based on the objects name and author 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:

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

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

When using __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`:

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

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 customization. On those compilers custom overloads for hash_value need 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
{
    struct custom_type
    {
        int value;

        friend inline 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.

So on these compilers define a member function:

```
#ifndef BOOST_NO_ARGUMENT_DEPENDENT_LOOKUP
        friend inline std::size_t hash_value(custom_type x)
        {
            boost::hash<int> hasher;
            return hasher(x.value);
        }
#else
        std::size_t hash() const
        {
            boost::hash<int> hasher;
            return hasher(value);
        }
#endif
```

which will be called from the `boost` namespace:

```
#ifdef BOOST_NO_ARGUMENT_DEPENDENT_LOOKUP
namespace boost
{
    std::size_t hash_value(foo::custom_type x)
    {
```

```
        return x.hash();
    }
}
#endif
```

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

Other Issues

On Visual C++ versions 6.5 and 7.0, `hash_value` isn't overloaded for built in arrays. `boost::hash`, `boost::hash_combine` and `boost::hash_range` all use a workaround to support built in arrays so this shouldn't be a problem in most cases.

On Visual C++ versions 6.5 and 7.0, function pointers aren't currently supported.

`boost::hash_value(long double)` on GCC on Solaris appears to treat `long double`s as doubles - so the hash function doesn't take into account the full range of values.

Reference

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.

Header <boost/functional/hash.hpp>

Includes all the following headers.

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

Defines `boost::hash`, the implementation for built in types and `std::string` and customisation functions.

```
namespace boost {
  template<typename T> struct hash;
  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);
  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(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&);
}
```

Struct template hash

Struct template hash -- An STL compliant hash function object.

```
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(val)`

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

Throws Only throws if `hash_value(T)` throws.

Function template hash_combine

Function template hash_combine -- Called repeatedly to incrementally create a hash value from several variables.

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

Description

Effects

```
seed ^= hash_value(v) + 0x9e3779b9 + (seed << 6) + (seed >> 2);
```

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

This is an extension to TR1

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

Function hash_range

Function hash_range -- Calculate the combined hash value of the elements of an iterator range.

```
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);
```

Description

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:

```
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

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

Function hash_value

Function hash_value -- Implementation of a hash function for integers.

```
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);
```

Description

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

Notes Overloads for other types supplied in other headers.

This is an extension to TR1

Returns `val`

Function hash_value

Function hash_value -- Implementation of a hash function for floating point values.

```
std::size_t hash_value(float val);
std::size_t hash_value(double val);
std::size_t hash_value(long double val);
```

Description

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

Notes — Overloads for other types supplied in other headers.

This is an extension to TR1

Returns — An unspecified value, except that equal arguments shall yield the same result

Function hash_value

Function hash_value -- Implementation of a hash function for pointers.

```
template<typename T> std::size_t hash_value(T* const& val);
```

Description

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

Notes — Overloads for other types supplied in other headers.

This is an extension to TR1

Returns — An unspecified value, except that equal arguments shall yield the same result

Function hash_value

Function hash_value -- Implementation of a hash function for built in arrays.

```
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] );
```

Description

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

Notes — Overloads for other types supplied in other headers.

This is an extension to TR1

Returns — `hash_range(val, val+N)`

Function hash_value

Function hash_value -- Implementation of a hash function for `std::basic_string`.

```
template<typename Ch, typename A>
  std::size_t hash_value(std::basic_string<Ch, std::char_traits<Ch>, A> const& val
```

Description

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

Notes — Overloads for other types supplied in other headers.

This is an extension to TR1

Returns — `hash_range(val.begin(), val.end())`

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

Hash implementation for `std::pair`.

```
namespace boost {
  template<typename A, typename B>
    std::size_t hash_value(std::pair<A, B> const &);
}
```

Function hash_value

Function hash_value --

```
template<typename A, typename B>
  std::size_t hash_value(std::pair<A, B> const & val);
```

Description

Effects

```
size_t seed = 0;
hash_combine(seed, val.first);
hash_combine(seed, val.second);
return seed;
```

Throws Only throws if `hash_value(A)` or `hash_value(B)` throws.

Notes This is an extension to TR1

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

Hash implementation for `std::vector`.

```
namespace boost {
  template<typename T, typename A>
    std::size_t hash_value(std::vector<T, A> const &);
}
```

Function hash_value

Function hash_value --

```
template<typename T, typename A>
  std::size_t hash_value(std::vector<T, A> const & val);
```

Description

Returns	`hash_range(val.begin(), val.end());`
Throws	Only throws if `hash_value(T)` throws.
Notes	This is an extension to TR1

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

Hash implementation for `std::list`.

```
namespace boost {
  template<typename T, typename A>
    std::size_t hash_value(std::list<T, A> const &);
}
```

Function hash_value

Function hash_value --

```
template<typename T, typename A>
  std::size_t hash_value(std::list<T, A> const & val);
```

Description

Returns	`hash_range(val.begin(), val.end());`
Throws	Only throws if `hash_value(T)` throws.
Notes	This is an extension to TR1

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

Hash implementation for `std::deque`.

```
namespace boost {
  template<typename T, typename A>
    std::size_t hash_value(std::deque<T, A> const &);
}
```

Function hash_value

Function hash_value --

```
template<typename T, typename A>
  std::size_t hash_value(std::deque<T, A> const & val);
```

Description

Returns `hash_range(val.begin(), val.end());`

Throws Only throws if `hash_value(T)` throws.

Notes This is an extension to TR1

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

Hash implementation for `std::set` and `std::multiset`.

```
namespace boost {
  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 &);
}
```

Function hash_value

Function hash_value --

```
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);
```

Description

Returns `hash_range(val.begin(), val.end());`

Throws Only throws if `hash_value(T)` throws.

Notes This is an extension to TR1

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

Hash implementation for `std::map` and `std::multimap`.

```
namespace boost {
  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 &);
}
```

Function hash_value

Function hash_value --

```
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);
```

Description

Returns	`hash_range(val.begin(), val.end());`
Throws	Only throws if `hash_value(std::pair<K const, T>)` throws.
Notes	This is an extension to TR1

Links

A Proposal to Add Hash Tables to the Standard Library
http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2005/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/n1745.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/n1756.pdf
The library implements the extension described in Issue 6.18.

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.

Acknowledgements

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

The 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 and Rob Stewart.

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.

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