Doc No: WG21 N3876
Date: 2014-01-19
Reply to: Nicolai Josuttis (nico@josuttis.de)
Subgroup: LEWG
Prev. Version: none

Convenience Functions to Combine Hash Values

Currently, there is not support in C++ to define hash functions for user-defined keys. Instead, the user has to implement an appropriate function. Implementing a hash function is not trivial. This proposal has the goal to make the definition of hash functions for user-defined types easier by providing a convenience function to combine multiple hash values.

The goal is not to provide a perfect hash function here, but to provide an easy-to-use interface to help application programmers to provide an pretty useful hash function so that they can use hash containers with their types. Note that this does not disable the ability to still provide better user-defined hash tables. It just helps to avoid that a user-defined hash function is better than a naive approach.

Note also that this paper doesn't provide an easier approach to define new hash functions (as discussed in N333).

Motivation

Consider the following example:

```
class Customer {
  public:
    …
    std::string getFirstname() const;
    std::string getLastname() const;
    int getAge() const;
};
bool operator== (const Customer&, const Customer&);
```

Now guess, you want to create an unordered container using Customers as elements. Currently, no default hash function is provided:

```
std::unordered_set<Customer> coll;   // ERROR
```

So, the user is required to provide a hash function, which without deep knowledge easily might result in providing a poor naive hash function (wither completely implemented himself/herself or poorly combining predefined hash functions). For example, my first approach was to create a hash value by adding the different values/members that should be taken into account:

```
class CustomerHash
{
  public:
    std::size_t operator() (const Customer& c) const {
        return std::hash<std::string>()(c.getFirstname()) +
               std::hash<std::string>()(c.getLastname()) +
               std::hash<std::string>()(c.getAge());
    }
};

std::unordered_set<Customer,CustomerHash> coll;   // OK, but bad
```

I learned from feedback that this is a bad hash function. But if even I (being an experienced programmer) make such a mistake, how can we require to provide good hash functions by ordinary application programmers? Thus, we should provide an easy-to-use approach to provide useful hash functions. In fact, using hash functions for basic types, we should provide the ability to create combined hash values out of different objects. Then, the application programmer could just define something like:

```
class CustomerHash
{
  public:
    std::size_t operator() (const Customer& c) const {
        return hash_val(c.getFirstname(),
                        c.getLastName(),
                        c.getAge());
    }
};

std::unordered_set<Customer,CustomerHash> coll;   // OK
```

That's easy to teach and easy to use.

There are papers about algorithms that combine hash values.
For example boost defines in
http://www.boost.org/doc/libs/1_35_0/doc/html/hash/combine.html:

```
template <typename T>
inline void hash_combine (std::size_t& seed, const T& val)
{
    seed ^= std::hash<T>()(val) + 0x9e3779b9
             + (seed<<6) + (seed>>2);
}
```

based of the following paper: http://www.cs.rmit.edu.au/~jz/fulltext/jasist-tch.pdf.

Thus, the following code provides the complete functionality:

```
template <typename T>
void hash_combine (std::size_t& seed, const T& val)
{
    seed ^= std::hash<T>()(val) + 0x9e3779b9
             + (seed<<6) + (seed>>2);
}
```

```
// auxiliary generic functions to create a hash value using a seed
template <typename T, typename... Types>
void hash_combine (std::size_t& seed,
                   const T& val, const Types&... args)
{
    hash_combine(seed,val);
    hash_combine(seed,args...);
}

// optional auxiliary generic functions to support hash_val() without arguments
void hash_combine (std::size_t& seed)
{
}

// generic function to create a hash value out of a heterogeneous list of arguments
template <typename... Types>
std::size_t hash_val (const Types&... args)
{
    std::size_t seed = 0;
    hash_combine (seed, args...);
    return seed;
}
```

Probably, we should not define the exact algorithm in the standard. But we should provide a corresponding interface. That's formally proposed here in this paper.

Relation to N333 and Comments Covering both

Note that N3333 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3333.html) provides a similar but broader approach for general hash functions. There, we have a different naming scheme. They propose:

- `hash_value(`*single_val*`)` as ADL supported convenience function, which you can overload for user-defined types (similar to `swap()`), and
- `hash_combine(`*val1*`,`*val2*`,`*val3*`)` to compute a hash value from multiple values.

Note however, that `hash_combine()` as proposed in this paper is used by boost for years now. The name hash_val() was chosen mainly because unlike make_pair() or make_shared() we don't create a hash object. We might name it hash_value() instead, though.

Jeffrey Jasskin commented on Jan 18, 2014:

1. It would be good if users could write the same code to expose their struct to the hash-table hasher as to the consistent-output hashers. Similarly, if the user wants to use something collision-resistant like SipHash (defense against http://emboss.github.io/blog/2012/12/14/breaking-murmur-hash-flooding-dos-reloaded/) for a particular hash table, even though they use a faster function for most tables, it would be nice if they could just drop it into the template argument, rather than having to rewrite all of their custom hashers. N3333 didn't solve this either.

2. Reducing the hash state to size_t at each step tends to run slower compared to keeping more state during the computation. http://burtleburtle.net/bob/hash/doobs.html (from 2006) mentions "One theoretical insight was that the last mix doesn't need to do well in reverse (though it has to affect all output bits). And the middle mixing steps don't have to affect all output bits (affecting some 32 bits is enough), though it does have to do well in reverse. So it uses different mixes for those two cases." Your paper's interface forbids this optimization.
3. Chandler reports that when he implemented N3333 for LLVM, he discovered that feeding one value at a time into the hasher made it hard for the optimizer to produce ideal code. He needed to take advantage of the variadic nature of hash_combine(). He couldn't give a concise explanation of why this was essential to make the optimizer do the right thing, but he hopes to put an example together before the actual meeting. Your hash_val() is variadic, so its implementation might be able to do this, but the requirement that it's equivalent to a series of binary hash_combine() calls worries me a bit.

Open Issues

- Should we allow to call hash_val() without arguments (provide hash_combine() with the seed argument only)? Currently, this would return 0. Is that OK?
- Should we provide a conditional noexcept declaration for hash_combine() (declaring to throw no exception if the hash function doesn't throw)?
- Should we make the type of the seed implementation defined?
- Should we even make all hash_combine() functions make internal and not part of the standard library? This which might allow improvements such as Jeffreys comment in list item "2." above.
- Do we need statements about the quality of hash_val() and/or hash_combine(), such as:
    - from N333: *If two calls to `hash_val()` aren't defined to return equal values, then their return values must be completely different with high probability. See http://en.wikipedia.org/wiki/Avalanche_effect for a possible (but not the only possible) meaning of "completely different with high probability". Situations that don't require equal hash_combine results include:*
        - *changing a single bit in any of hash_combine()'s arguments. (This includes changing hash_combine(false) to hash_combine(true).)*
        - *calling hash_combine in a different execution of the same binary.*
        - *Replacing hash_combine(arg1, arg2, arg3) with hash_combine(hash_combine(arg1, arg2), arg3)*
    - Complexity statements?
- Should hash_val() be overload-able for (easier) user-provided hash functions or should it be a pure library functionality
    - I prefer to distinguish and separate this proposal (for defining new hash values by combining existing ones) from a proposal that makes it easier to define new hash functions. But this should then look for an appropriate naming scheme.

- Bike shed: hash_val() or hash_value()?
- Should we also provide a version to combine hash values from range elements (hash values from tuples, …)?
    - Note: Such a function would need a different name and could be added later.

Acknowledgements

Thanks to Matt Austern, Jeffrey Jasskin, Mikael Kilpeläinen, Geoff Pike, Zhihao Yuan for providing feedback and taking time for discussions.

Proposed Resolution

In 2**0.8 Function objects [function.objects] §2, in the synopsis of** <functional> **after** all declarations of hash<> add:

// 20.8.13, hash convenience templates:

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

template <typename T, typename... Types>
void hash_combine (std::size_t& seed,
                   const T& val, const Types&... args);

void hash_combine (std::size_t& seed);

template <typename... Types>
std::size_t hash_val (const Types&... args);
```

After 20.8.12 Class template hash [unord.hash] add the following new paragraphs:

20.8.13 Hash convenience templates [hash.conv]

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

Effects: modifies seed so that the resulting value is a hash value combined out of the initial value of seed and hash<T>()(val) .
Throws: nothing in case hash<T>()(val) throws nothing.

```
template <typename T, typename... Types>
void hash_combine (std::size_t& seed,
                   const T& val, const Types&... args);
```

Effects: equivalent to

```
hash_combine(seed,val);
hash_combine(seed,args...);
```

```
void hash_combine (std::size_t& seed);
```

Effects: None

```
template <typename... Types>
std::size_t hash_val (const Types&... args);
```

Effects: equivalent to

```
std::size_t seed = 0;
hash_combine (seed, args...);
return seed;
```