Project: ISO JTC1/SC22/WG21: Programming Language C++
Doc No: WG21 **P0814R2**
Date: 2018-02-12
Reply to: Nicolai Josuttis (nico@josuttis.de)
Audience: LEWG, LWG
Prev. Version: P0814R1

Revisions:

R2: Integrated all proposed improvements from LEWG in Albuquerque in 2017.

- Make the feature available in <functional>.
- No requirement to be associative and no requirement to be non-associative.
- Provide a range overload.

Regarding the name there was no clear recommendation by LEWG. Note that any future change of hash_combine() will probably break binary compatibility, so this is an issue to consider. We can

- Provide a more specific name (e.g. `hash_combine_size_t()`)
- Or leave it as it is because for a new approach we should anyway introduce a new name because `hash_combine()` with `size_t` is already established by boost.
  This opinion has slightly more consensus (a new name already mentioned in LEWG for an improved hashing was `hash_append()`).

The latter arguments were slightly more compelling; so we keep `hash_combine()`.

R1: Added a few key design questions and fixed the proposed wording to allow that the same arguments in the same order yield the same result if nested called (thanks to Geoffrey Romer).

hash_combine() Again

C++11 came out with hash containers but poor support to implement hash functions.
A few proposals tried to fix that:

- **In 2012**, N3333 "Hashing User-Defined Types in C++1y"
- **In 2014**, N3976 "Convenience Functions to Combine Hash Values"

N3976 was more or less rejected with the promise that N333 will solve it better soon. But now, 3-5 years later in C++17, we still don't have support to help application programmers to use unordered containers for their own types. Proving:

> The perfect is the enemy of the good

This paper proposed a minimal solution that still gives freedom to future standard to make it better.

The proposal is roughly taken from the following requirement, which both papers saw as a valid and common request and were more or less proposing the same solution:

- Application programmers should have a convenience function to compute a combined hash value from the hash values of types for which std::hash<> is supported.

Thus, for example, to use a class Customer in a hash container the programmer simply should be able to program:

```
struct MyCustomerHash {
  std::size_t operator() (const Customer& c) const {
    return hash_combine(c.getFirstname(),
                        c.getLastName(),
                        c.getAge());
  }
};
std::unordered_set<Customer,CustomerHash> coll;
```

With fold expression, `hash_combine()` is easy to implement. For example:

```
template<typename T>
void _hash_combine (size_t& seed, const T& val)
```

```
{
  seed ^= std::hash<T>()(val) + 0x9e3779b9 + (seed<<6) + (seed>>2);
}

template<typename... Types>
size_t hash_combine (const Types&... args)
{
  size_t seed = 0;
  (_hash_combine(seed,args) , ... );   // create hash value with seed over all args
  return seed;
}
```

However, the underlying hash combine function is not easy to implement (here, we use Boost's approach, see e.g., http://www.boost.org/doc/libs/1_35_0/doc/html/hash/combine.html).
Platform-specific aspects also might matter.
For this reason, making it part of the library is a useful step.

For future compatibility we suggest to make the return type of `hash_combine()` a template parameter with a default type:

```
template<typename RT = size_t, typename... Types>
RT hash_combine (const Types&... args)
{
  std::size_t seed = 0;
  (_hash_combine(seed,args) , ... );   // create hash value with seed over all args
  return seed;
}
```

Issues Discussed and Resolved in Albuquerque 2017

A key issue to decide was whether `hash_combine()` should be associative. That is, should `hash_combine(t0,t1,t2)` and `hash_combine(hash_combine(t0,t1),t2)` and `hash_combine(t0,hash_combine(t1,t2))` compare equal?
Why this helps to support using different borders of chunks of memory for the same hash value, the resulting hash value quality might become worse.
Thus, as recommended by LEWG, associativity is not required. But non-assiociativity is also not required (that is, it is implementation defined, whether `hash_combine(t0,t1,t2)` equals `hash_combine(t0, hash_combine(t1,t2))`.

Another question was, whether we should extend this API to take a pair of iterators.
This is useful, because having them avoids to have a loop at runtime.

Finally, this convenience function should be available for sets and maps. In which header should we define it? The proposed recommendation by LEWG was in <functional> (just as hash<>).

Proposed Wording:

In 23.14.1 Header <functional> synopsis [functional.syn]

After `template <class T> struct hash;` add the following new function template:

```
namespace std {
  template<typename RT = size_t, typename... T>
  RT hash_combine(const T&... args);

  template<typename RT = size_t, typename InputIterator>
  RT hash_combine(InputIterator first, InputIterator last);
}
```

with the following definitions:

```
template<typename RT = size_t, typename... T>
  RT hash_combine (const T&... args);
```

Requires: For any T_i the specialization of hash<T_i> is enabled (23.14.15).

Effects: Calls `hash<Ti>()(argsi)` for all i and combines the resulting hash values.
Returns: the combined hash value with the following constraints:

- All return values are equal with the same input for a given execution of the program.
- For three different values t0, t1 and t2, the probability that `hash_combine(t0,t1,…)`, `hash_combine(t1,t2,…)`, and `hash_combine(t0,t2,…)` compare equal should be very small, approaching 1.0 / numeric_limits<size_t>::max(),
  except that `hash_combine(t0,t1,t2,…)` and
  `hash_combine(hash_combine(t0,t1),t2,…)` and
  `hash_combine(t0,hash_combine(t1,t2),…)`, etc. may compare equal.

[Note: `hash_combine(arg1,arg2)` may differ from `hash_combine(arg2,arg1)` and `hash_combine(arg1,arg2,arg3)` may differ from `hash_combine(hash_combine(arg1, arg2), arg3).`]

```
template<typename RT = size_t, typename InputIterator>
  RT hash_combine (InputIterator first, InputIterator last);
```

Requires: first != last and the specialization of hash<InputIterator::value_type> is enabled (23.14.15).
Returns: an object whose type is RT and whose value is

- `hash_combine(*first)` if `next(first)==last` or
- `hash_combine(hash_combine(hash_combine(*i0, *i1), *i2), *i3, …)` for every iterator i in `[advance(first, 1), last)`, otherwise.

Acknowledgements

Thanks to all who incredibly helped me to prepare this paper, such as Geoffrey Romer and all people in the C++ library working group.