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:

```cpp
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:

```cpp
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;
}

Proposed Wording:
Available in both <unordered_set> and <unordered_map>
add the following new function template:

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

with the following definition:

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

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

Effects: Calls hash<Ti>()(argsi) for all i and combines the resulting hash value with the following constraints:
• All return values are equal with the same input for a given execution of the program.
• For two different values t1 and t2, the probability that hash_combine(t1,...) and hash_combine(t2,...) compare equal should be very small, approaching 1.0 / numeric_limits<size_t>::max().

[Note: hash_combine(args1,args2) may differ from hash_combine(args2,args1) and
hash_combine(args1,args2,args3) may differ from
hash_combine(hash_combine(args1, args2), args3).]

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