Document number:	P0199R0
Date:	2016-02-11
Project:	ISO/IEC JTC1 SC22 WG21 Programming Language C++
Audience:	Evolution Working Group
Reply-to:	Vicente J. Botet Escriba <vicente.botet@wanadoo.fr>

Default Hash

Abstract

Defining `hash_value` or specializing `is_uniquely_represented` as defined in P0029R0 for simple classes is tedious, repetitive, slightly error-prone, and easily automated.

I propose to (implicitly) supply default version of this operation and trait, if needed. The meaning of `hash_value` is to combine the members using `hash_combine`.

Table of Contents

Introduction

Defining `hash_value` or specializing `is_uniquely_represented` as defined in P0029R0 for simple classes is tedious, repetitive, slightly error-prone, and easily automated.

I propose to (implicitly) supply default version of this operation and trait, if needed. The meaning of `hash_value` is to combine the members using `hash_combine`.

If the simple defaults are unsuitable for a class, a programmer can, as ever, define more suitable ones or suppress the defaults. The proposal is to add the operations as an integral part of C++ (like =), rather than as a library feature.

The proposal follows the same approach as Default comparison as in N4475, that is, that having default generated code for these basic operations only when needed and possible would make the language simpler.

This paper contains no proposed wording. This is a discussion paper to determine EWG interest in the feature, and if there is interest to get direction for a follow-up paper with wording.

Motivation

Some standard algorithms require that an argument type supply a valid `hash` instantiation. Writing such types can be tedious (and all tedious tasks are error prone).

For example

```
class Foo {
  int i;
  string str;
  bool b;
  //...
  friend bool operator==(const Foo& lhs, const Foo& rhs) {
    return lhs.i == rhs.i && lhs.str == rhs.str && lhs.b == rhs.b;
  }

  template <class H>
  friend H hash_value(H h, const Foo& foo) {
    return hash_combine(std::move(h), foo.i, foo.str, foo.b);
  }
};
```

If Default comparison N4475 is adopted, the `==` operator will be not needed anymore as it could be generated by default as `=` operator is already.

Proposal

I propose to generate default versions for `hash_value` for simple classes when needed. If those defaults are unsuitable for a type, `=delete` them. If non-default of those operations are needed, define them (as always). If an operation is already declared, a default is not generated for it. This is exactly the way assignment and constructors work today and as comparison operators would work is N4475 is adopted.

Note that if `==` operator is defined by the user, `hash_value` default generation couldn't reflect the user decisions, and so it seems reasonable to not provide such a generation.

The same rationale given in N4475 applies `hash_value`.

This paper uses the last hash proposal P0029R0 as it pretends to unify previous proposals as N3980. This paper should be adapted to the final proposal. For the time being, we will use P0029R0.

It could also be great if the `hash_value` function could be generated by the compiler following the philosophy and criteria as defined in N4475.

What about `is_uniquely_represented`?

When `==` operator is generated by the compiler, it is not difficult to specialize `is_uniquely_represented` when the type of the concerned members (those used to define `==` operator), let call them `Ti` satisfy `is_uniquely_represented<Ti>` and there are no other data members nor padding.

Here it is how it could be specialized for `std::pair`.

```
template <class T, class U>
struct is_uniquely_represented<std::pair<T, U>>
    : public std::bool_constant<is_uniquely_represented<T>::value &&
                                is_uniquely_represented<U>::value &&
                                sizeof(T) + sizeof(U) == sizeof(std::pair<T, U>)>
{
};
```

If `is_uniquely_represented<C>{}==true`, there is no need to overload `hash_value` as `hash_value` is already defined for those types.

When `is_uniquely_represented` could specialized?

There would be no need to specialize it when the result will be false. However we must do it for templates as we don't know at compile time the result. So the specialization will be always generated.

In addition to the common restrictions defined above, the following restricts the generation of `is_uniquely_represented` specialization

- `is_uniquely_represented` has been already specialized before the first need for `is_uniquely_represented`, or
- the `==` operator is user defined or cannot be generated,

[Note: If the there is a specialization of `is_uniquely_represented` after its first need or in another translation unit, the program would be already ill-formed as there is a violation of the ODR. -- end]

What is the definition of `hash_value`?

The natural definition which combines, using `hash_combine`, each one of the data members seem to be a good candidate for the default.

When `hash_value` could be applied?

The following restricts the generation of `hash_value` for a class C

- has that operation defined or deleted before the first need for `hash_value` or
- the `==` operator is user defined or cannot be generated, or
- `is_uniquely_represented<C>{}==true`, or
- has a user-defined or deleted copy or move operation, or
- has a virtual base, or
- has a virtual function, or
- has a pointer member.

The generated implementation for `hash_value` is not considered a function so it cannot have its address taken [Note: like the default `=` operator].

Mutable members are ignored for the generated implementations.

[Note: If the overload of `hash_value` appears after its first need or in another translation unit, the program would be already ill-formed as there is a violation of the ODR. -- end]

[Note: Identifying this violation would require link-time checking. -- end].

Design Rationale

Why require that the default generation of `operator==` is applied?

The members use in the default generation for `hash_value` or the specialization of `is_uniquely_represented` must be the same than the ones that are taken in account for the definition of the `==` operator. When the user defines the `==` operator, there is no evident way to ensure this constrain. This is why the default generation for `hash_value` and `is_uniquely_represented` is applied only when the default generation of `==` is also applied.

`is_uniquely_represented` specialization

We could let the user the responsibility to specialize, however the author suspect that the user could forget to do it and the proposed specialization is to the author knowledge always safe.

Working paper wording

This wording is very "drafty" and has not gone through expert review. It is intended to reflect the design decisions described above.

The author was not aware of the new wording for Default comparison in N4532. There are a lot of there that should inspire the wording for this function.

The wording that follows is based on the wording of the current standard in particular N4527 and in initial wording in N4475.

Add a "Hash expression" section in 5

Hash expression [expr.hash_value]

A *hash expression* is a particular case of a *function call expression* when the function name is `hash_value`.

If an operand is of class type and no suitable function is found in the class namespace, the implicitly-declared `hash_value` non-member operation as described in over.generate*hash*value is used.

Add a "Special non-member hash_value operation" section after 13.6

Special non-member `hash_value` operation [over.generate*hash*value]

Implicitly-declared `hash_value` non-member operation

If no user-defined `hash_value` operation is provided for a class type `T` (`struct`, `class` but not `union`), and all of the following is true:

- there are no user-declared `operator==`;
- `is_uniquely_represented<C>{}==false`;

then the compiler will declare a `friend` `hash_value` operation with the signature

```
template <class H>
friend void hash_value(H, T&) noexcept(see below);
```

The generated implementation is not considered a function so it cannot have its address taken [Note: like the = operator.].

Explicitly defaulted `hash_value` non-member operation

The user may still force the generation of the implicitly declared `hash_value` operation declaring it aa a friend operation with the keyword `default`.

The generated implementation is not considered a function so it cannot have its address taken [Note: like the = operator.].

Deleted implicitly-declared `hash_value` non-member operation

The implicitly-declared or defaulted `hash_value` operation for class `T` is defined as deleted in any of the following is true:

- `T` has non-static data members that don't supports `hash_value`;
- `T` has direct or virtual base class that don't supports `hash_value`;
- `T` implicit-generated `operator==` is deleted;

The deleted implicitly-declared `hash_value` operation is ignored by overload resolution.

Implicitly-defined `hash_value` non-member operation

If the implicitly-declared `hash_value` operation is not deleted, it is defined (that is, a function's body is generated and compiled) by the compiler if odr-used. The `hash_value` non-member operation `hash_combine` each concerned non-static data member of the object, in their initialization order.

Alternative solutions

Based on a future reflection library e.g. N4428 or N4451, we could define the `hash_value` function instead of generating it. However, to the author knowledge, this would need to declare a friend function, which is much more intrusive than the compiler generated solution.

Next follows un incomplete implementation

```
namespace std {
namespace experimental {namespace reflect { inline namespace v1 {

    template <class C>
    struct is_hash_value_generation_enabled;

}}}

template <class H, class C>
enable_if<reflect::is_hash_value_generation_enabled_t<C>{}, H>
hash_value(H h, C & x)
{
    // std::hash_combine the non-static non-mutable data-member
    return std::hash_combine(h, ...)
}

} // namespace std
```

Note that the `hash_value` overload would need to be declared as friend on the class.

```
namespace MyNS {

class MyC {

    template <class H, class C>
        friend
        enable_if<
            std::experimental::reflect::is_hash_value_generation_enabled_t<C>{}, H>
        hash_value(H h, C & x);
    // ...
};

} //namespace
```

This would be almost a showstopper and one of the reasons, that even with reflection, a compiler generated version is a better and less intrusive choice.

Implementability

This proposal needs some compiler magic, either by generating directly the `hash_value` function or by providing the reflection traits as e.g. in N4428 or N4451.

Open Questions

The authors would like to have an answer to the following points if there is any interest at all in this proposal:

- Do we want a default or a reflection solution?
- Do we want implicit instantiation by the compiler of `is_uniquely_represented<C>` ?

Summary

Defaulting `hash_value` operation is simple, removes a common annoyance. It is completely compatible. In particular, the existing facilities for defining and suppressing those operations are untouched.

Acknowledgments

Thanks to Bjarne Strustrup for its clear identification of the types that are subject to this kind of default generation in N4475. Many thanks to Howard Hinnant, his comments in the ML, that allowed me to better understand how the user customization must be used and think about adding `is_uniquely_represented` specialization by the compiler.

Thanks to all those that have commented the idea on the std-proposals ML helping to better identify the constraints and improve the proposal in general.

References

- N3980 Types Don't Know #

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3980.html

- N4428 Type Property Queries (rev 4)

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4428.pdf

- N4451 Static reflection

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4451.pdf

- N4475 Default comparisons (R2)

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4475.pdf

- N4527 Working Draft, Standard for Programming Language C++

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4527.pdf

- N4532 Proposed wording for default comparisons

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4532.html

- P0017R0 Extension to aggregate initialization

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0017r0.html

- P0029R0 A Unified Proposal for Composable Hashing

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0029r0.html