Document number: N3983
Date: 2014-05-07
Project: Programming Language C++, Library Evolution Working Group
Reply-to: Geoffrey Romer <gromer@google.com>

Hashing tuple-like types

Introduction

This paper proposes adding `std::hash` specializations for `std::pair`, `std::tuple`, and `std::array` to the standard library.

Motivation and Scope

It is often useful to have a map whose key is composed from multiple values, and an obvious and natural way to do this is to use a `pair` or `tuple` as the key. This works perfectly for associative containers, but currently fails for unordered associative containers, unless the user supplies a custom hash functor, because `std::hash` is not specialized for tuples or pairs, and user code is not permitted to provide such specializations unless one of the element types is (or depends on) a user-defined type. This needless obstacle substantially reduces the value of pair and tuple as vocabulary types.

As an example of existing practice, I've surveyed a very large C++ codebase that uses pre-C++11 vendor hash containers, with a hash specialization for pair in a widely-available header (specializations for `tuple` and `array` also exist, but were added too recently to provide meaningful data). In this codebase, `pair<string, string>` is the 11th most common key type for hash containers, exceeded only by a handful of integral and string types, and `pair<int, int>` is not far behind at #14. The overwhelming majority of `pair` key types do not depend on user-defined types, and so user-provided specializations are not an option even in principle.

This paper proposes to add `std::hash` specializations for the "tuple-like" types in the standard library, namely `std::pair`, `std::tuple`, and `std::array`.

Impact on the Standard

This is an extension to the standard library; it has no effect on the core language at all, and does not alter any existing library functionality. It introduces new partial specializations for `std::hash`, which user code is also permitted to specialize, and that raises the possibility that it may change the behavior of existing code, or even break it. However, the new partial specializations are all on standard library types, so [namespace.std]/p1 appears to ensure that when an instantiation of `std::hash` in a conformant program matches both a user-provided specialization and a specialization introduced by this proposal, the user-declared specialization will be more specialized, and so no ambiguity will be introduced, and the behavior of the code will remain the same.

Design Decisions

The major design decision made by this paper is its scope. I have opted to keep this proposal simple, minimizing impact on the standard while addressing the demonstrated user need and maintaining interface consistency. Consequently, this proposal does not introduce any new interfaces, but simply applies an existing extension mechanism (specialization of `std::hash`) to an existing pseudo-concept ("tuple-like types", q.v. [tuple.creation]/p13) which narrowly encompasses the use cases of interest. Various further extensions are possible, e.g. to support hashing of containers, but in the absence of motivating use cases, they are left as future work.

This proposal implicitly requires implementations to have some sort of hash combining mechanism, in order to produce a composite hash from the elements of a tuple. There is significant user demand for a public hash combining API, but this is already being addressed by proposals such as N3333, N3898, and N3876. This proposal leaves the combining mechanism unspecified, and therefore complements those proposals without depending on them.

One possible benefit of providing a `hash` specialization for `tuple` would be to provide a convenient mechanism for defining hash functions for user-defined types:

```
struct my_type {
  string s;
  int i;
  std::unique_ptr<Foo> f;
};

namespace std {
template<> struct hash<my_type> {
  size_t operator()(const my_type& v) {
    auto tup = std::tie(v.s, v.i, v.f);  // or forward_as_tuple?
    return hash<decltype(tup)>()(tup);
  }
}
```

However, in order to avoid copying the elements of the struct, code like this must hash a tuple of references, not ordinary values. Thus, to enable this type of usage, we would need to either provide special rules for reference elements in the `tuple` specialization of `hash`, or specialize `hash` on arbitrary reference types. The former option is decidedly ad-hoc, and would greatly complicate both the standard wording and the implementation. The latter looks quite promising, but is beyond the scope of this proposal. The aforementioned papers address custom hash definition directly, so there is little need complicate this proposal with a more limited solution to the same issue.

This proposal requires `hash<pair<T1, T2>>` to provide behavior equivalent to `hash<tuple<T1, T2>>`. This is intended to maximize interoperability between the two types, so that `pair<T1, T2>` can almost be thought of as a template alias for `tuple<T1, T2>`. In the absence of this requirement, users would face the potentially surprising situation that if p has type `pair<T1, T2>`, `hash<pair<T1, T2>>()(p)` and `hash<tuple<T1, T2>>()(p)` are both valid expressions (thanks to `pair`'s implicit convertibility to `tuple`), but can produce different results. However, good type hygiene should prevent such surprises, and we certainly can't guarantee in general that implicit conversions preserve hash values, so this requirement is not essential.

Along the same lines, it might be considered desirable for `hash<array<T, N>>` to behave equivalently to `hash<tuple<Ts...>>`, where `Ts...` consists of `N` repetitions of `T`. However, in this case there is a countervailing pressure to make `hash<array<T, N>>` behave equivalently to `hash<vector<T>>` for a size-`N` `vector`, in the not-unlikely event that such a specialization is ever added to the standard. `hash<array<T, N>>` could in principle comply with both requirements, but this could impose a nontrivial implementation burden, so in the absence of a stronger motivation, this proposal takes the conservative option of imposing no such requirements.

Proposed Wording

Wording is relative to N3797.

In [utility], add the following to the `<utility>` header synopsis, immediately above `struct integer_sequence`:

```
// [pair.hash] pair hash support
template <class T> struct hash;
template <class T1, class T2> struct hash<pair<T1, T2> >;
```

Following [pair.piecewise], add a new subsection of [pairs]:

Pair hash support [pair.hash]

```
template <class T> struct hash;
template <class T1, class T2> struct hash<pair<T1, T2> >;
```

The class template shall meet the requirements of class template `hash` [unord.hash]. For an object `p` of type `pair<T1, T2>`, `hash<pair<T1, T2> >()(p)` shall evaluate to the same value as `hash<tuple<T1, T2> >()(p)` [tuple.hash].

In [tuple.general], add the following to the `<tuple>` header synopsis, immediately below `swap`:

```
// [tuple.hash], hash support
template <class T> struct hash;
template <class... Types> struct hash<tuple<Types...>>;
```

Following [tuple.special], add a new subsection of [tuple]:

Tuple hash support [tuple.hash]

`template <class T> struct hash;`
`template <class... Types> struct hash<tuple<Types...>>;`
1. The class template shall meet the requirements of class template `hash` [unord.hash].
2. Requires: For every `Type` in `Types`, the specialization `hash<Type>` shall be well-formed and well-defined, and shall meet the requirements of class template `hash` [unord.hash].

In [sequences.general], add the following to the <array> header synopsis, immediately below the overloads of `get`:

```
template <class T> struct hash;
template <class T, size_t N> struct hash<array<T, N>>;
```

Following [array.tuple], add a new subsection of [array]:

Array hash support [array.hash]

`template <class T> struct hash;`
`template <class T, size_t N> struct hash<array<T, N>>;`
1. The class template shall meet the requirements of class template `hash` [unord.hash].
2. Requires: The specialization `hash<T>` shall be well-formed and well-defined, and shall meet the requirements of class template `hash` [unord.hash].