C++ unordered_map using a custom class type as the key

I am trying to use a custom class as key for unordered_map, like the following,

```
#include <iostream>
#include <algorithm>
#include <unordered_map>
//#include <map>

using namespace std;

class node;
class Solution;

class Node {
    public:
        int a;
        int b;
        int c;
        Node(){}
        Node(vector<int> v) {
            sort(v.begin(), v.end());
            a = v[0];
            b = v[1];
            c = v[2];
        }

        bool operator==(Node i) {
            if ( i.a==this->a && i.b==this->b &&i.c==this->c ) {
                return true;
            } else {
                return false;
            }
        }
};


int main() {

    unordered_map<Node, int> m;

    vector<int> v;
    v.push_back(3);
    v.push_back(8);
    v.push_back(9);
    Node n(v);

    m[n] = 0;

    return 0;
}
```

I guess I need the tell C++ how to hash class Node, however, I am not quite sure how to do it. Is there any example for this kind of tasks?

Thanks a lot!

The following is the error from g++:

```
In file included from /usr/include/c++/4.6/string:50:0,
                 from /usr/include/c++/4.6/bits/locale_classes.h:42,
                 from /usr/include/c++/4.6/bits/ios_base.h:43,
                 from /usr/include/c++/4.6/ios:43,
                 from /usr/include/c++/4.6/ostream:40,
                 from /usr/include/c++/4.6/iostream:40,
                 from 3sum.cpp:4:
/usr/include/c++/4.6/bits/stl_function.h: In member function ‘bool
std::equal_to<_Tp>::operator()(const _Tp&, const _Tp&) const [with _Tp = Node]’:
/usr/include/c++/4.6/bits/hashtable_policy.h:768:48:   instantiated from ‘bool
std::__detail::_Hash_code_base<_Key, _Value, _ExtractKey, _Equal, _H1, _H2,
std::__detail::_Default_ranged_hash, false>::_M_compare(const _Key&,
std::__detail::_Hash_code_base<_Key, _Value, _ExtractKey, _Equal, _H1, _H2,
std::__detail::_Default_ranged_hash, false>::_Hash_code_type,
std::__detail::_Hash_node<_Value, false>*) const [with _Key = Node, _Value =
std::pair<const Node, int>, _ExtractKey = std::_Select1st<std::pair<const Node,
int> >, _Equal = std::equal_to<Node>, _H1 = std::hash<Node>, _H2 =
std::__detail::_Mod_range_hashing, std::__detail::_Hash_code_base<_Key, _Value,
_ExtractKey, _Equal, _H1, _H2, std::__detail::_Default_ranged_hash,
false>::_Hash_code_type = long unsigned int]’
/usr/include/c++/4.6/bits/hashtable.h:897:2:   instantiated from
```

```
‘std::_Hashtable<_Key, _Value, _Allocator, _ExtractKey, _Equal, _H1, _H2, _Hash,
_RehashPolicy, __cache_hash_code, __constant_iterators, __unique_keys>::_Node*
std::_Hashtable<_Key, _Value, _Allocator, _ExtractKey, _Equal, _H1, _H2, _Hash,
_RehashPolicy, __cache_hash_code, __constant_iterators,
__unique_keys>::_M_find_node(std::_Hashtable<_Key, _Value, _Allocator, _ExtractKey,
_Equal, _H1, _H2, _Hash, _RehashPolicy, __cache_hash_code, __constant_iterators,
__unique_keys>::_Node*, const key_type&, typename std::_Hashtable<_Key, _Value,
_Allocator, _ExtractKey, _Equal, _H1, _H2, _Hash, _RehashPolicy, __cache_hash_code,
__constant_iterators, __unique_keys>::_Hash_code_type) const [with _Key = Node,
_Value = std::pair<const Node, int>, _Allocator = std::allocator<std::pair<const
Node, int> >, _ExtractKey = std::_Select1st<std::pair<const Node, int> >, _Equal =
std::equal_to<Node>, _H1 = std::hash<Node>, _H2 =
std::__detail::_Mod_range_hashing, _Hash = std::__detail::_Default_ranged_hash,
_RehashPolicy = std::__detail::_Prime_rehash_policy, bool __cache_hash_code =
false, bool __constant_iterators = false, bool __unique_keys = true,
std::_Hashtable<_Key, _Value, _Allocator, _ExtractKey, _Equal, _H1, _H2, _Hash,
_RehashPolicy, __cache_hash_code, __constant_iterators, __unique_keys>::_Node =
std::__detail::_Hash_node<std::pair<const Node, int>, false>, std::_Hashtable<_Key,
_Value, _Allocator, _ExtractKey, _Equal, _H1, _H2, _Hash, _RehashPolicy,
__cache_hash_code, __constant_iterators, __unique_keys>::key_type = Node, typename
std::_Hashtable<_Key, _Value, _Allocator, _ExtractKey, _Equal, _H1, _H2, _Hash,
_RehashPolicy, __cache_hash_code, __constant_iterators,
__unique_keys>::_Hash_code_type = long unsigned int]’
/usr/include/c++/4.6/bits/hashtable_policy.h:546:53:   instantiated from
‘std::__detail::_Map_base<_Key, _Pair, std::_Select1st<_Pair>, true,
_Hashtable>::mapped_type& std::__detail::_Map_base<_Key, _Pair,
std::_Select1st<_Pair>, true, _Hashtable>::operator[](const _Key&) [with _Key =
Node, _Pair = std::pair<const Node, int>, _Hashtable = std::_Hashtable<Node,
std::pair<const Node, int>, std::allocator<std::pair<const Node, int> >,
std::_Select1st<std::pair<const Node, int> >, std::equal_to<Node>, std::hash<Node>,
std::__detail::_Mod_range_hashing, std::__detail::_Default_ranged_hash,
std::__detail::_Prime_rehash_policy, false, false, true>,
std::__detail::_Map_base<_Key, _Pair, std::_Select1st<_Pair>, true,
_Hashtable>::mapped_type = int]’
3sum.cpp:149:5:   instantiated from here
/usr/include/c++/4.6/bits/stl_function.h:209:23: error: passing ‘const Node’ as
‘this’ argument of ‘bool Node::operator==(Node)’ discards qualifiers [-fpermissive]
make: *** [threeSum] Error 1
```

c++ hash g++ unordered-map hashtree

asked Jun 10 '13 at 2:34

Alfred Zhong

1,138 • 5 • 22 • 34

1 The third template argument is the hash function you need to supply. – chrisaycock Jun 10 '13 at 2:38

1 cppreference has a simple and practical example of how to do this: en.cppreference.com/w/cpp/container/unordered_map/unordered_map – jogojapan Jun 10 '13 at 2:53

1 Answer

To be able to use `std::unordered_map` (or one of the other unordered associative containers) with a user-defined key-type, you need to define two things:

1. A **hash function**; this must be a class that overrides `operator()` and calculates the hash value given an object of the key-type. One particularly straight-forward way of doing this is to specialize the `std::hash` template for your key-type.
2. A **comparison function for equality**; this is required because the hash cannot rely on the fact that the hash function will always provide a unique hash value for every distinct key (i.e., it needs to be able to deal with collisions), so it needs a way to compare two given keys for an exact match. You can implement this either as a class that overrides `operator()`, or as a specialization of `std::equal`, or – easiest of all – by overloading `operator==()` for your key type (as you did already).

The difficulty with the hash function is that if your key type consists of several members, you will usually have the hash function calculate hash values for the individual members, and then somehow combine them into one hash value for the entire object. For good performance (i.e., few collisions) you should think carefully about how to combine the individual hash values to ensure you avoid getting the same output for different objects too often.

A fairly good starting point for a hash function is one that uses bit shifting and bitwise XOR to combine the individual hash values. For example, assuming a key-type like this:

```
struct Key
{
  std::string first;
  std::string second;
  int         third;

  bool operator==(const Key &other) const
  { return (first == other.first
            && second == other.second
            && third == other.third);
  }
};
```

Here is a simple hash function (adapted from the one used in the cppreference example for user-defined hash functions):

```
namespace std {

  template <>
  struct hash<Key>
  {
    std::size_t operator()(const Key& k) const
    {
      using std::size_t;
      using std::hash;
      using std::string;

      // Compute individual hash values for first,
      // second and third and combine them using XOR
      // and bit shifting:

      return ((hash<string>()(k.first)
               ^ (hash<string>()(k.second) << 1)) >> 1)
               ^ (hash<int>()(k.third) << 1);
    }
  };

}
```

With this in place, you can instantiate a `std::unordered_map` for the key-type:

```
int main()
{
  std::unordered_map<Key,std::string> m6 = {
    { {"John", "Doe", 12}, "example"},
    { {"Mary", "Sue", 21}, "another"}
  };
}
```

It will automatically use `std::hash<Key>` as defined above for the hash value calculations, and the `operator==` defined as member function of `Key` for equality checks.

If you don't want to specialize template inside the `std` namespace (although it's perfectly legal in this case), you can define the hash function as a separate class and add it to the template argument list for the map:

```
struct KeyHasher
{
  std::size_t operator()(const Key& k) const
  {
    using std::size_t;
    using std::hash;
    using std::string;

    return ((hash<string>()(k.first)
             ^ (hash<string>()(k.second) << 1)) >> 1)
             ^ (hash<int>()(k.third) << 1);
  }
};

int main()
{
  std::unordered_map<Key,std::string,KeyHasher> m6 = {
    { {"John", "Doe", 12}, "example"},
    { {"Mary", "Sue", 21}, "another"}
  };
}
```

How to define a better hash function? As said above, defining a good hash function is important to avoid collisions and get good performance. For a real good one you need to take into account the distribution of possible values of all fields and define a hash function that projects that distribution to a space of possible results as wide and evenly distributed as possible.

This can be difficult; the XOR/bit-shifting method above is probably not a bad start. For a slightly better start, you may use the `hash_value` and `hash_combine` function template from the Boost library. The former acts in a similar way as `std::hash` for standard types (recently also including tuples and other useful standard types); the latter helps you combine individual hash values into one. Here is a rewrite of the hash function that uses the Boost helper functions:

```
#include <boost/functional/hash.hpp>

struct KeyHasher
{
  std::size_t operator()(const Key& k) const
  {
      using boost::hash_value;
      using boost::hash_combine;

      // Start with a hash value of 0    .
      std::size_t seed = 0;

      // Modify 'seed' by XORing and bit-shifting in
      // one member of 'Key' after the other:
      hash_combine(seed,hash_value(k.first));
```

```
      hash_combine(seed,hash_value(k.second));
      hash_combine(seed,hash_value(k.third));

      // Return the result.
      return seed;
  }
};
```

answered Jun 10 '13 at 5:18

jogojapan
32.7k • 5 • 50 • 72

3 Can you please explain why it is necessary to shift the bits in `KeyHasher` ? – Wildling Aug 12 '14 at 17:39

8 If you didn't shift the bits and two strings were the same, the xor would cause them to cancel each other out. So hash("a","a",1) would be the same as hash("b","b",1). Also order wouldn't matter, so hash("a","b",1) would be the same as hash("b","a",1). – Buge Aug 29 '14 at 15:40

1 I am just learning C++ and one thing I always struggle with is: Where to put the code? I have written a specialize `std::hash` method for my key as you have done. I put this at the bottom of my Key.cpp file but I am getting the following error: `Error 57 error C2440: 'type cast' : cannot convert from 'const Key' to 'size_t' c:\program files (x86)\microsoft visual studio 10.0\vc\include\xfunctional` . I am guessing that the compiler is not finding my hash method? Should I be adding anything to my Key.h file? – Ben Feb 7 at 17:13

1 @Ben Putting it into the .h file is correct. `std::hash` is not actually a struct, but a *template (specialization) for a struct*. So it isn't an implementation -- it will be turned into an implementation when the compiler needs it. Templates should always go into header files. See also stackoverflow.com/questions/495021/… – jogojapan Feb 8 at 1:02

2 @nightfury `find()` returns an iterator, and that iterator points to an "entry" of the map. An entry is a `std::pair` consisting of key and value. So if you do `auto iter = m6.find({"John","Doe",12});` , you'll get the key in `iter->first` and the value (i.e. the string `"example"`) in `iter->second` . If you want the string directly, you can either use `m6.at({"John","Doe",12})` (that will throw an exception if the key doesn't exits), or `m6[{"John","Doe",12}]` (that will create an empty value if the key doesn't exist). – jogojapan Apr 21 at 8:12