Doc No: N2944=09-0134

Date: 2009-09-27

Authors: John Lakos
Bloomberg LP
jlakos@bloomberg.net

Pablo Halpern

Intel

phalpern@halpernwightsoftware.com

Equality Comparison for Unordered Containers

Contents

Background

Apart from unordered collections, all of the container types in the current WP appear to be consistent with what N2479 refers to as having *value semantics*. In particular, each container type defines default construction, copy construction, copy assignment and the two (homogeneous, free) equality comparison operators, `operator==` and `operator!=`, with the truth of `operator==` being a postcondition of both copy construction and assignment (23.1.1 [container.requirements.general], Paragraph 5, Table 90 [Container requirements]). By contrast, unordered containers are currently explicitly exempt (23.1.5 [unord.req], Paragraph 2, Table 96 [Container requirements that are not required for unordered associative containers], and Paragraph 10) from having to implement the equality comparison operators, yet in all other respects are consistent with having value semantics as defined by N2479.

This manifest inconsistency has been discussed in the past (circa 2004) and documented in N1837 (search for 6.2). The fundamental problem was that the equality function described in the container requirements (23.1.1 [container.requirements.general], Paragraph 5, Table 90 [container.requirements.general]) – i.e., that == is an equivalence relation and

```
    a.size() == b.size() && equal(a.begin(), a.end(), b.begin())
```

and that the behavior is linear – "makes no sense for hash tables" because the order of iteration is not considered a *salient attribute* (i.e., one that contributes to value) of the container (see N2479) and separately because the (pathological) worst-case behavior of equality comparison is necessarily quadratic. The alternatives considered included, "close as NAD, put in a caveat saying we don't quite satisfy the container requirements, put in the `operator==` defined in terms of `std::equal`, or put in Howard's (more useful) `operator==`." By a vote of 0-6-0-3 straw vote the second was chosen.

Status Quo: Inconsistent and Incomplete

The unordered containers are, as defined today, inconsistent in that they are inherently value-semantic types that implement all of the value-semantic operations except for the equality comparison operations (== and !=). Moreover, the attempt to document the omission (23.1.5 [unord.req], Paragraph 2, Table 96 [Container requirements that are not required for unordered associative containers], and Paragraph 10) introduced additional inconsistencies, which (along with numerous other inaccuracies and inconsistencies) must be repaired, regardless. These unordered containers are also incomplete in that, unlike the ordered containers, they do not make explicit what abstract ("mathematical") type they are approximating or what abstract ("externalizable") values they are attempting to represent. Furthermore, these unordered collections omit an important, explicitly documented (23.1.1 [container.requirements.general], Paragraph 5, Table 90 [Container requirements]) postcondition of all STL containers (and arguably all C++ *value-semantic*) types – i.e., that, after copy construction or copy assignment, the (*values* of the) source and destination must compare equal. By defining equality comparison for unordered collections, we complete the set of *value-semantic* operations on them and, in so doing, force all of these *value-semantic* operations to be both self-consistent and also mutually consistent with their counterparts on the corresponding ordered collections.

Document Conventions

All section names and numbers are relative to the August 2008 working draft, N2723.

> Existing working paper text is indented and shown in dark blue. Edits to the working paper are shown with ~~red strikeouts for deleted text~~ and green underlining for inserted text within the indented blue original text.

Comments and rationale mixed in with the proposed wording appears as shaded text.

Requests for LWG opinions and guidance appear with light (yellow) shading. It is expected that changes resulting from such guidance will be minor and will not delay acceptance of this proposal in the same meeting at which it is presented.

Discussion

Based on private discussions with various members of the LWG (as well as those documented in N1837), the impediments to defining homogeneous `operator==` and `operator!=` for unordered collections invariably involve one or more of the following concerns:

1. It's not clear we need them (let's not implement something that won't be used).

2. We don't know exactly how to define them (hence, we might get them wrong and have to live with another less-than-well-thought-out feature).

3. Any reasonably definition would come with a concomitant runtime cost that would be prohibitively expensive at runtime (any use would therefore be subtly dangerous).

We will now address each of these concerns in turn.

Why do we need operator== for unordered collections?

Regarding the first concern (lack of utility), there are at least four separate reasons why we need equality comparison for unordered collections. The first and dominant reason is utility: The concept of value for unordered containers is meaningful and useful in practical applications. It is arguably more common to want to know if two sets have the same elements independent of order than it is to know if two sets have the same elements in the same order. For example, if we give a group of children bags of candy that allegedly *compare equal*, what's important is that, ultimately, the items in the bags respectively compare equal, not the order in which these items are initially discovered.

As a second example, consider implementing a (Hyper) Graph of N nodes in terms of STL containers. In standard matrix notation of a graph, a directed edge from Node i to j is present if there is a non-zero value in the (i, j)th element of the matrix. Two graphs have the same value if they have they have the same standard matrix representation. There are many ways we might consider using STL to model such an object. For example, we could use a `multimap<int,int>` of size E (number of edges) to encode each edge. The problem with such an approach is that it fails to implement the notion of value described in the specification, as the relative order of the edges emanating from a single node would be treated as a silent attribute of value. A better alternative would be to use an `unordered_multimap<int,int>`. Now the relative order of edges emanating from a node will not be considered when comparing the two graphs (as required by the specification). Another correct approach would be to create a vector<unordered_multiset<int> > of size N to represent the (non-unique) set of adjacent edges at each O(1) randomly-accessible node. Yet another approach might be to create a `vector<multiset<int> >` of size N where the adjacent edges are kept ordered. In this case, the semantics for value happen to be correct because each member of each group of the equivalent keys in the (ordered) multiset represents the identical value so the order in which they were added is not observable.

As a third and final example, consider implementing a TelephoneBook type whose value is defined solely by the data it contains and not by the order in which it was added. Our initial choice might be a map`<string,int>.` This choice is fine assuming that the names are unique, but suppose there are occasionally duplicate names with distinct phone numbers (e.g., different people, or perhaps the same person with multiple numbers). In that case, we might consider using a `multimap<string,int>`, but now the order in which duplicate keys are added would necessarily affect value. If the order of entries with duplicate key values should not affect the overall value of a `TelephoneBook` object, then the data structure of choice is `unordered_multimap<string,int>.`

The second reason why we need equality comparison defined for unordered containers is consistency: There is simply no valid reason (see below) why unordered containers should be the only container types in the STL that do not support all of the essential *value-semantic* operations. Gratuitous inconsistency needlessly complicates understanding and creates barriers to effective use.

The third reason is to establish a common vocabulary: We want to provide a single uniform definition of value upon which everyone can rely. Without such a definition, developers will "roll their own" and these interpretations will invariably differ (e.g., perhaps some will conclude, based solely on efficient implementation, that the order of equivalent keys should be considered part of the value).

The fourth reason is testability. Implementing this important, explicitly stated, postcondition of both copy construction and copy assignment allows developers who use such containers to unit-test their software using the standard "***y == f(x)***" test paradigm – i.e., generate an initial (unordered collection) value, ***x***, apply an application subroutine, ***f***, to ***x***, separately generate an expected (unordered collection) value, ***y***, and finally use the provided (unordered collection) equality-comparison operator, **==**, to assert that **y** has the same value as ***f(x)***.

How should we define operator== for unordered containers?

Regarding the second concern (suboptimal specification), our extensive understanding of (and experience with) *value-semantic* types (as described in N2479) gives us a large body of knowledge including essential properties of *value semantics* that helps guide us in providing a useful, efficient, and consistent definition of *value* for unordered collections, based on *salient attributes* – i.e., those aspects of an object's instance state that contribute to its overall *value*. The salient attributes of an unordered container are simply the values of the elements in the container, regardless of iteration order. Thus, too unordered containers have the same value if one is a permutation of the other.

In general, computing permutations is a quadratic operation. However, given two unordered containers that use the same hash and key-equivalence functions, the elements will be partitioned into key-equivalence groups that make comparison much more efficient. Thus, we define the result of `operator==` for unordered collections as follows: two unordered collections, **a** and **b**, have the same *value* if (1) `a.size() == b.size()`, and (2) for each contiguous group (ai_1, ai_2) of equivalent keys — as defined by `a.eq_key(k1, k2)` and returned by `a.equal_range(k1)` — in **a,** there exists a (necessarily unique) corresponding contiguous group (bi_1, bi_2) of equivalent keys — as defined by `b.eq_key(k1, k2)` and returned by `b.equal_range(k1)` — in **b** such that (i) `distance(ai1, ai2) == distance(bi1, bi2)` and there exists a reordering ($bi_{1'}$, $bi_{2'}$) of the elements in (bi_1, bi_2) such that `equal(ai1, ai2, bi1')` would return true. Reference implementations consistent with both the above definition and the formal wording can be found in the appendix at the end of this document.

Wouldn't operator== for unordered containers be slow?

Regarding the third concern (prohibitive runtime cost), the useful and intuitive definition of equality comparison for unordered containers proposed here has a runtime cost that is linear in the average case and quadratic only in the pathological worst case. For the unordered containers that permit equivalent keys, the complexity of `operator==` is proportional to ΣE_i^2 in the average case and N^2 in the worse case, where N is `a.size()`, and E_i is the size of the ith equivalent-key group in **a**. Note that if the number of duplicate keys (or even the maximum number of duplicates in any contiguous group of equivalent keys) is known to be bounded by a constant, then the overall (worst-case) cost of equality comparison is O(`a.size()`).

More generally, it should be understood that the expected behavior of nearly all of the operations on unordered containers (including copy construction and assignment, element insertion, and lookup) have an expected runtime behavior that is fast, but is a factor of N slower in the worst case; equality comparison is no different and therefore deserves no special consideration in that regard. For unordered containers that support equivalent keys, allowing the container to be populated with largely duplicated keys, although likely to result in poor runtime performance, is not so much a fault of the data structure itself, but rather of its use by unenlightened clients (who would be well-advised to consider other strategies).

In conclusion, we have established that there is a need for equality comparison between unordered containers of the same type, that a consistent and intuitive definition of such a homogeneous equality-comparison operation exists, and that its expected runtime is highly efficient in all but pathological cases – consistent with the vast majority of other operations on unordered containers. We therefore provide formal language to introduce the notion of value for unordered containers and, at the same time, clean up the numerous inaccuracies and inconsistencies we have found in section 23.1 [container requirements], section 23.3 [associative containers], and especially section 23.4 [unordered associative containers]. (Note that this paper subsumes all of the defects and repairs noted in open issue #861.) Finally, an additional, general-purpose algorithm, `is_permutation` (analogous to `equal`), taking 3 iterators (with and without an optional binary predicate) is added to section 25.3 [alg.nonmodifying] in order to facilitate both the documentation and implementation of equality-comparison operators for unordered associative containers supporting non-unique keys.

Summary and Scope of Changes

In a nutshell, we propose to make the following changes to the WP:

- Add a new algorithm, `is_permutation`, that is analogous to `equal`.
- Define `operator==` and `operator!=` for unordered associative containers.
- Update container requirements tables.
- Repair defects in affected sections along the way as appropriate.

Proposed Wording

23.1.1 General Container Requirements [container.requirements.general]

Change Table 90 [Container equirements] as indicated:

`X u;`			post: ~~`u.size() == 0`~~`u.empty() != false`	constant
`X()`			~~`X.size() == 0`~~`X().empty() != false`	constant

...

`a == b`	`convertible to bool`	`==` is an equivalence relation. ~~`a.size()== b.size()`~~ `distance(a.begin(),a.end()) == distance(b.begin(),b.end()) && equal(a.begin(),a.end(),b.begin())`	`T` is EqualityComparable	linear

...

`a.size()`	`size_type`	~~`a.end() - a.begin()`~~`distance(a.begin(),a.end())`		(Note A)
`a.max_size()`	`size_type`	size~~()~~ of the largest possible container		(Note A)
`a.empty()`	`convertible to bool`	~~`a.size() == 0`~~`a.begin() == a.end()`		constant

These changes are to accommodate `std::forward_list`, which supports homogeneous equality comparison operators operator== and operator!= , but does not define a size() method (since it cannot be implemented in constant time). We considered adding an extra row for unordered container operator== to this table, but could not find a way to express the semantics in a way that was concise enough for the table format.

Change Table 92 [Allocator-Aware Container Requirements] as indicated:

`X()`
`X u;`
		Requires: `A` is DefaultConstructible. post: `u.size() == 0, get_allocator() == A()`	constant

...

`a == b`	`convertible to bool`	`==` is an equivalence relation. ~~`a.size()== b.size()`~~ `distance(a.begin(),a.end()) == distance(b.begin(),b.end()) && equal(a.begin(),a.end(),b.begin())`	linear

23.1.3 General Container Requirements [sequence.reqmts]

Change Table 93 [Sequence container requirements (in addition to container)] as indicated:

a. Change the text in the Assertion/note column in the row for "`X(n, t) / X a(n, t)`" as follows:

[..] `post:` ~~u.size()~~ distance(begin(), end()) `== n` [..]

b. Change the Assertion/note column in the row for "`X(i, j) / X a(i, j)`" as follows:

[..] `post:` ~~u.size()~~ distance(begin(), end()) `== distance(i, j)` [..]

c. Change the text in the Assertion/note column in the row for "`a.clear()`" as follows:

`a.erase(a.begin(), a.end()) post: a.`~~size() == 0~~empty() == true

23.1.4 Associative Containers [associative.reqmts]

Change Table 95 [Associative container requirements (in addition to container)] as indicated:

Not every occurrence of `size()` was replaced, because all current associative containers have a `size`. The following changes ensure consistency regarding the semantics of "`erase`" for all tables and add some missing objects.

:

a) Change the text in the Complexity column in the row for "`a.insert(i, j)`" as follows:

`N log(a.size() + N)` ~~(N is the distance from i to j)~~where N == distance(i, j)

b) Change the text in the Complexity column in the row for "`a.erase(k)`" as follows:

`log(a.size()) + a.count(k)`

c) Change the text in the Complexity column in the row for "`a.erase(q1, q2)`" as follows:

`log(a.size()) + N` ~~where N is the distance from q1 to q2~~ == distance(q1, q2).

d) Change the text in the Assertion/note column in the row for "`a.clear()`" as follows:

`a.erase(a.begin(),a.end()) post:` ~~size() == 0~~a.empty() == true

e) Change the text in the Complexity column in the row for "`a.clear()`" as follows:

`linear in a.size()`

f) Change the text in the Complexity column in the row for "`a.count(k)`" as follows:

`log(a.size()) + a.count(k)`

23.1.5 Associative Containers [unord.reqmts]

In section 23.1.5 [unord.req], change paragraph 2 and table 96 as follows:

2 Unordered associative containers conform to the requirements for Containers (23.1), except that the expressions `a == b` and `a != b` have different semantics than for the other container types and the expressions in table 96 are not required to be valid, where `a` and `b` denote values of a type `X`, and `X` is an unordered associative container class:

Container requirements that are not required for unordered associative containers

~~`a == b`~~
~~`a != b`~~
`a < b`
`a > b`
`a <= b`
`a >= b`

Change Paragraphs 5 and 6 as indicated:

5 Two values k1 and k2 of type Key are considered ~~equal~~equivalent if the container's ~~equality~~key_eq function object returns true when passed those values. If k1 and k2 are ~~equal~~equivalent, the hash function shall return the same value for both.

6 An unordered associative container supports *unique keys* if it may contain at most one element for each key. Otherwise, it supports *equivalent keys*. `unordered_set` and `unordered_map` support unique keys. `unordered_multiset` and `unordered_multimap` support equivalent keys. In containers that support equivalent keys, elements with equivalent keys are adjacent to each other in the iteration order of the container. Thus, although the absolute order of elements in an unordered container is not specified, the elements are grouped into *equivalent-key groups*, such that all elements of each group have equivalent keys. Mutating operations on unordered containers preserve the relative order of elements within each equivalent-key group, unless otherwise specified. ~~For unordered_multiset and unordered_multimap, insert and erase preserve the relative ordering of equivalent elements.~~

Change Table 97 [Unordered associative container requirements (in addition to container)] as indicated:

The same rationale as for Table 95 applies here plus we also correct the complexity of insert and the postcondition for max_load_factor.

a. Change the text in the Complexity column in the row for "`a.insert(i,j)`" as follows

Average case O(N), where N is `distance(i,j)`. Worst case O(N * (`a.size()` + N)).

b. Change the text in the Assertion/note column in the row for "`a.clear()`" as follows:

[..] Post: a.~~size() == 0~~empty() == true

c. Change the text in the Assertion/note column in the row for "a.max_load_factor(x)" as follows:

Pre: z shall be positive. ~~Changes~~May change the container's maximum load factor using z as a hint.

Change Paragraph 10 as indicated:

10 ~~Unordered associative containers are not required to support the expressions a == b or a != b. [Note: This is because the container requirements define operator equality in terms of equality of ranges. Since the elements of an unordered associative container appear in an arbitrary order, range equality is not a useful operation. — end note]~~

10 Two unordered containers, a and b compare equal if a.size() == b.size() and, for every equivalent-key group, [Ea1,Ea2), in a, there exists an equivalent-key group, [Eb1,Eb2), in b such that distance(Ea1, Ea2) == distance(Eb1,Eb2) and is_permutation(Ea1, Ea2, Eb1) returns true. For unordered_set and unordered_map, the complexity of operator== (i.e., the number of calls to operator==(value_type, value_type), key_eq::operator() and hasher::operator()) is proportional to N in the average case, and to N^2 in the worst case, where N is a.size(). For unordered_multiset and unordered_multimap, the complexity of operator== is proportional to $\sum Ei^2$ in the average case, and to N^2 in the worse case, where N is a.size(), and Ei is the size of the ith equivalent-key group in a. However, if the respective elements of each corresponding pair of equivalent-key groups, Eai and Ebi are arranged in the same order (as is commonly the case, e.g., if a and b are unmodified copies of the same container), then the average-case complexity for unordered_multiset and unordered_multimap becomes proportional to N (but worst-case complexity remains $O(N^2)$ -- i.e., for a pathologically bad hash function). The behavior of operators == and != is undefined for unordered containers unless (1) the Hash and Pred function objects respectively have the same behavior for both containers ,and (2) operator==(Key,Key) is a refinement of the partition into equivalent-key groups produced by Pred.

The meaning of value for both ordered and unordered containers is based on the definition of value for the contained elements themselves, and not on that implied by key_equal as defined by the pred functor supplied to the container as a template interface policy. We have therefore elected to make explicit here the requirement (for overall container equality comparisons) that equality comparison be defined for contained elements.

We have made three additional explicit (compile-time) pre-conditions on the valid use of equality comparison for unordered containers, based on semantics, performance, and an absence of practical need to support the restricted behavior. The first restriction is that both containers have the same interpretation of equivalent keys. If they do not, then it is possible for two unique-key containers of the same C++ type that currently have the same value to be acted upon in the same way (e.g., via an insert of the same key value) and subsequently not have the same value, thus violating the fundamental property of value-semantic types. Although it is theoretically possible to retain proper value semantics and still allow unequal key comparators for non-unique unordered associative containers, the required algorithm would necessarily be quadratic in the average case.

The second restriction is that the hashers must have the same behavior. Allowing the hashers to be different would mean that we would give up an important optimization for both unique and non-unique unordered containers: When the number of buckets in the two unordered containers is the same, we can avoid having to hash each iterated element (or maintain its hash value along with its value) if, on lookup, we can assume that value hashes to the same bucket, which can be significant when the cost of hashing compared to that of equality comparison on elements is significant.

Finally, the requirement that equality comparison for keys be a refinement on key equivalence defined for unordered associative containers is also motivated by efficient implementation: Without this property, we cannot exploit the contiguous equivalent keys in non-unique associative containers, and will again have to resort to an algorithm that is quadratic even in the average case. While this restriction is necessary only for the non-unique associative containers, allowing it only for the unique ones seems to have insufficient practical benefit that would justify documenting the distinction.

Removing this third restriction for unordered containers with unique keys only is easy to do at the cost of a little more complexity in the description of the preconditions. Do people feel that such functionality would be useful?

23.4 Unordered associative containers [unord]

Add the following to the synopsis for <unordered_map>:

```
template <class Key, class T, class Hash, class Pred, class Alloc>
  bool operator==(unordered_map<Key, T, Hash, Pred, Alloc>& a,
                  unordered_map<Key, T, Hash, Pred, Alloc>& b);
template <class Key, class T, class Hash, class Pred, class Alloc>
  bool operator!=(unordered_map<Key, T, Hash, Pred, Alloc>& a,
                  unordered_map<Key, T, Hash, Pred, Alloc>& b);
template <class Key, class T, class Hash, class Pred, class Alloc>
  bool operator==(unordered_multimap<Key, T, Hash, Pred, Alloc>& a,
                  unordered_multimap<Key, T, Hash, Pred, Alloc>& b);
template <class Key, class T, class Hash, class Pred, class Alloc>
  bool operator!=(unordered_multimap<Key, T, Hash, Pred, Alloc>& a,
                  unordered_multimap<Key, T, Hash, Pred, Alloc>& b);
```

and add the following to the synopsis for <unordered_set>:

```
template <class Key, class T, class Hash, class Pred, class Alloc>
  bool operator==(unordered_set<Key, T, Hash, Pred, Alloc>& a,
                  unordered_set<Key, T, Hash, Pred, Alloc>& b);
template <class Key, class T, class Hash, class Pred, class Alloc>
  bool operator!=(unordered_set<Key, T, Hash, Pred, Alloc>& a,
                  unordered_set<Key, T, Hash, Pred, Alloc>& b);
template <class Key, class T, class Hash, class Pred, class Alloc>
  bool operator==(unordered_multiset<Key, T, Hash, Pred, Alloc>& a,
                  unordered_multiset<Key, T, Hash, Pred, Alloc>& b);
template <class Key, class T, class Hash, class Pred, class Alloc>
```

```
   bool operator!=(unordered_multiset<Key, T, Hash, Pred, Alloc>& a,
                   unordered_multiset<Key, T, Hash, Pred, Alloc>& b);
```

25 Algorithms Library [algorithms]

Add the following two declarations after those for `equal` but before those for `search`:

```
template<class ForwardIterator1, class ForwardIterator2>
  bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
                      ForwardIterator2 first2);
template <class ForwardIterator1, class ForwardIterator2,
          class BinaryPredicate>
  bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
                      ForwardIterator2 first2, BinaryPredicate pred);
```

Add the following paragraph 12 at the end of this section:

12 For algorithms that operate on ranges where the end iterator, last2, of the second range is not specified, it is required that `distance(first1, last1) <= distance(first2, last2).`

25.1.12 Search [alg.search]

Insert a new section 25.1.12 [alg.Is_permutation] after 25.1.11 [alg.equal] making Search section 25.1.13:

25.1.12 Is permutation [alg.is_permutation]

```
template<class ForwardIterator1, class ForwardIterator2>
  bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
                      ForwardIterator2 first2);

template<class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
                      ForwardIterator2 first2, BinaryPredicate pred);
```

1 Returns: true if there exists a reordering, first2', of the sequence beginning with first2 such that equal(first1, last1, first2'), equal(first1, last1, first2', pred) != false. Otherwise, returns false.

2 Complexity: Exactly (last1 – first1) of the corresponding predicate if `equal(first1,last1,first2),equal(first1,last1,first2,pred)` would return true; otherwise at O((distance(first1,last1)^2) applications of the corresponding predicate.

References

N2479: Normative Language to Describe Value Copy Semantics (http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2007/n2479.pdf)

N1837: Library Extension Technical Report – Issues List (http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2005/n1837.pdf)

Appendix: Reference Implementations

The five code examples in this section illustrate defined behavior along with some useful optimizations.

Equality Comparison: unordered_set

```
template <class KEY>
bool operator==(const unordered_set<KEY>& a,
                const unordered_set<KEY>& b)
{
    typedef typename unordered_set<KEY>::const_iterator ConstIter;

    if (a.size() != b.size()) {
        return false;
    }

    for (ConstIter itr1 = a.begin(); itr1 != a.end(); ++itr1) {
         ConstIter itr2 = b.find(*itr1);
        if(itr2 == b.end() || *itr1 != *itr2) {
            return false;
        }
    }
    return true;
}
```

Equality Comparison: unordered_map

```
template <class KEY, class VALUE>
bool operator==(const unordered_map<KEY, VALUE>& a,
                const unordered_map<KEY, VALUE>& b)
{
    typedef typename unordered_map<KEY, VALUE>::const_iterator ConstIter;

    if (a.size() != b.size()) {
        return false;
    }

    for (ConstIter itr1 = a.begin(); itr1 != a.end(); ++itr1) {
         ConstIter itr2 = b.find(itr1->first);
        if(itr2 == b.end() || *itr1 != *itr2) {
            return false;
        }
    }
    return true;
}
```

Equality Comparison: unordered_multiset

```
template <class KEY>
bool operator==(const unordered_multiset<KEY>& a,
                const unordered_multiset<KEY>& b)
{
    typedef typename unordered_multiset<KEY>::const_iterator ConstIter;
    if (a.size() != b.size()) {
        return false;
    }

    for (ConstIter itr1 = a.begin();
         itr1 != a.end();
         /* Increment within the loop */) {

        // First check the two equivalent-key groups have the same size.
        pair<ConstIter, ConstIter> aRange = a.equal_range(*itr1);
        pair<ConstIter, ConstIter> bRange = b.equal_range(*itr1);

        if (distance(aRange.first, aRange.second)
         != distance(bRange.first, bRange.second)) {
            return false;
        }

        // Then check whether the two equivalent-key groups are permutations
        // of each other.
        if(!is_permutation(aRange.first,
                           aRange.second,
                           bRange.first)) {
            return false;
        }

        // Increment the iterator to the next equivalent-key group.
        itr1 = aRange.second;
    }
    return true;
}
```

Equality Comparison: unordered_multimap

```
template <class KEY, class VALUE>
bool operator==(const unordered_multimap<KEY, VALUE>& a,
                const unordered_multimap<KEY, VALUE>& b)
{
    typedef typename unordered_multimap<KEY, VALUE>::const_iterator ConstIter;

    if (a.size() != b.size()) {
        return false;
    }

    for (ConstIter itr1 = a.begin();
         itr1 != a.end();
         /* Increment within the loop */) {

        // First check the two equivalent-key groups have the same size.
        pair<ConstIter, ConstIter> aRange = a.equal_range(itr1->first);
        pair<ConstIter, ConstIter> bRange = b.equal_range(itr1->first);

        if (distance(aRange.first, aRange.second)
         != distance(bRange.first, bRange.second)) {
            return false;
        }
        // Then check whether the two equivalent-key groups are permutations
        // of each other.
        if(!is_permutation(aRange.first,
                           aRange.second,
                           bRange.first)) {
            return false;
        }

        // Increment the iterator to the next equivalent-key group.
        itr1 = aRange.second;
    }
    return true;
}
```

Is_permutation

```
template <class ForwardIterator1, class ForwardIterator2>
bool
is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
               ForwardIterator2 first2)
{
    typedef typename
        iterator_traits<ForwardIterator2>::difference_type
        distance_type;

    // Efficiently compare identical prefixes: O(N) if sequences
    // have the same elements in the same order.

    for ( ; first1 != last1; ++first1, ++first2) {
        if (! (*first1 == *first2))
            break;
    }
    if (first1 == last1)
        return true;

    // Establish last2 assuming equal ranges by iterating over the
    // rest of the list.

    ForwardIterator2 last2 = first2;
    advance(last2, distance(first1, last1));

    for (ForwardIterator1 scan = first1; scan != last1; ++scan) {
        if (scan != find(first1, scan, *scan))
            continue;  // We've seen this one before
        distance_type matches = count(first2, last2, *scan);
        if (0 == matches || count(scan, last1, *scan) != matches)
            return false;
    }

    return true;
}
```