Matthew Austern <austern@apple.com>
9 Apr 2003
Doc number N1456=03-0039

A Proposal to Add Hash Tables to the Standard Library (revision 4)

I. Motivation

Unordered associative containers—hash tables—are one of the most frequently requested additions to the standard C++ library. Although hash tables have poorer worst-case performance than containers based on balanced trees, their performance is better in many real-world applications.

Hash tables are appropriate for this TR because they plug an obvious hole in the existing standard library. They are not intended for any one specific problem domain, style of programming, or community of programmers. I expect them to be used by a wide range of programmers.

There is extensive experience with hash tables implemented in C++ in the style of standard containers. Hash tables were proposed for the C++ standard in 1995; the proposal was rejected for reasons of timing. Three independently written libraries, SGI, Dinkumware, and Metrowerks, now provide hashed associative containers as an extension. (The GNU C++ library includes hash tables derived from SGI's.)

The three shipping hash table implementations are similar, but not identical; this proposal is not identical to any of them. Some of the differences will be discussed in section III.

II. Impact On the Standard

This proposal is a pure extension. It proposes a minor change to an existing header (a new function object in <functional>), but it does not require changes to any standard classes or functions and it does not require changes to any of the standard requirement tables. It does not require any changes in the core language, and it has been implemented in standard C++.

This proposal does not depend on any other library extensions. The initial implementation does use some nonstandard components, but they're part of the implementation rather than the interface and they aren't part of this proposal.

III. Design Decisions

A. Packaging Issues

The three implementations in current use, as well as the Barreiro/Fraley/Musser proposal, all used the names `hash_set`, `hash_map`, `hash_multiset`, `hash_multimap`. Existing practice suggests that these names should be retained.

Unfortunately, existing practice is also a strong argument for choosing a different name. Reusing a name that's already in common use, and applying it to a class with a similar but not identical interface, has too great a potential for confusion. Additionally, since several vendors have defined classes in namespace `std` with the `hash_*` names, defining a standard class with that name would introduce a nasty backward compatibility problem. Vendors would have to figure out a transition strategy for going from the pre-standard classes to the standard ones.

Accordingly, this proposal instead chooses the names `unordered_set`, `unordered_map`, `unordered_multiset`, and `unordered_multimap`. These names have the further advantage that they may alert users to the most important way in which (say) `unordered_set` differs from `set`: the former lacks the latter's ordering guarantee.

This naming choice was suggested by the Library Working Group. The working group considered and rejected several other potential solutions to the name collision problem:

- *The same general approach, but with slightly different names.* Two other names that were considered were `bag_set` and `unsorted_set`. The LWG believed that `unordered_set` was clearer than those two, because it comes closest to expressing the essential way in which these containers differ from the existing associative containers.
- *Minor changes to the earlier names*, such as `hashset` or `hashed_set`. The LWG rejected this approach because it was believed that having two almost-identical classes with almost-identical names would be too likely to cause user confusion.
- *An extra namespace*, so that the names might look like `std::hash::hash_set` or `std::hash_table::set`. The LWG rejected this approach for two reasons. First, there is no precedent in the existing Standard Library for using namespaces in this way. Second, since using directives are common and naive users often gloss over the question of which namespace a class is in, namespace segregation would probably not suffice to avoid user confusion.
- *Reusing the existing associative container names*, `set`, `map`, `multiset`, and `multimap`, and giving those classes extra policy parameters so users can select between the existing usage as sorted associative containers, and the new usage as hash tables. This is certainly implementable. However, the LWG believes that it would be even more confusing than the other two options: it's hard to know how we would document a class whose interface (member functions, nested types, and class invariants) would completely change depending on the

presence of a template parameter. We might equally well imagine having a single policy-based class, `container`, instead of the separate containers `vector`, `deque`, and `list`.

- *Keep the earlier names,* and find a way for implementers to make it work. One might imagine, for example, using metaprogramming techniques where `hash_set<>` would examine its template arguments and whether the user indended to use it with the old or the new interface. Again, while "do what I mean" machinery is certainly implementable, the LWG believed that it was likely to be fragile or confusing.

The distinction between the four unordered containers is the same as the distinction between the four standard associative containers. All four unordered containers allow lookup of elements by key. In `unordered_set` and `unordered_multiset` the elements are the keys; modification of elements is not allowed. In `unordered_map` and `unordered_multimap` the elements are of type `pair<const Key, Value>`. The key part can't be modified, but the value part can. In `unordered_set` and `unordered_map`, no two elements may have the same key; in `unordered_multiset` and `unordered_multimap` there may be any number of elements with the same key.

This proposal defines the `unordered_set` and `unordered_multiset` classes within a new `<unordered_set>` header, and `unordered_map` and `unordered_multimap` within `<unordered_map>`. It defines a default hash function, `hash<>`, within the standard header `<functional>`; see III.D for a discussion of the decision to define a `hash<>` within `<functional>`.

This proposal defines a set of Unordered Associative Container requirements, and then separately describes four classes that conform to those requirements. In that respect this proposal follows the lead of the standard and of the Barreiro/Fraley/Musser proposal. However, Barreiro, Fraley, and Musser proposed more extensive changes to the container requirements. They proposed two new requirements tables, not one: Sorted Associative Container, satisfied by the existing standard associative containers, and Hashed Associative Container. They then modified the existing Associative Container requirements (table 69 in the C++ standard) so that both sorted associative containers and hashed associative containers would satisfy the new Associative Container requirements. The difference is shown in Figure 1.

Figure 1: Container taxonomy as described by this proposal and by Barreiro, Fraley, and Musser

This proposal:

```
               /-- Sequence
              /
   Container ---- Associative Container
              \
               \-- Unordered Associative Container
```

Barreiro, Fraley, and Musser:

```
               /-- Sequence
              /
   Container                             /-- Sorted Associative
Container
              \                         /
               \-- Associative Container
                                        \
                                         \-- Hashed Associative
Container
```

I believe that the Barreiro/Fraley/Musser taxonomy is better: the generality of the name "Associative Container", and the specificity of table 69, aren't a good match. However, that proposal was made before the C++ standard was finalized. The superiority of the Barreiro/ Fraley/Musser taxonomy isn't so great as to justify changing an existing requirements table that users may be relying on.

The three hash table implementations in current use are not identical, but they are similar enough that for simple uses they are interchangeable. This proposal attempts to maintain a similar level of compatibility.

B. Chaining Versus Open Addressing

Knuth (section 6.4 of *The Art of Computer Programming*) distinguishes between two kinds of hash tables: "chaining", where a hash code is associated with the head of a linked list, and "open addressing", where a hash code is associated with an index into an array.

I'm not aware of any satisfactory implementation of open addressing in a generic framework. Open addressing presents a number of problems:

- It's necessary to distinguish between a vacant position and an occupied one.
- It's necessary either to restrict the hash table to types with a default constructor, and to construct every array element ahead of time, or else to maintain an array some of whose elements are objects and others of which are raw memory.
- Open addressing makes collision management difficult: if you're inserting an element whose hash code maps to an already-occupied location, you need a

policy that tells you where to try next. This is a solved problem, but the best known solutions are complicated.
- Collision management is especially complicated when erasing elements is allowed. (See Knuth for a discussion.) A container class for the standard library ought to allow erasure.
- Collision management schemes for open addressing tend to assume a fixed size array that can hold up to N elements. A container class for the standard library ought to be able to grow as necessary when new elements are inserted, up to the limit of available memory.

Solving these problems could be an interesting research project, but, in the absence of implementation experience in the context of C++, it would be inappropriate to standardize an open-addressing container class.

All further discussion will assume chaining. Each linked list within a hash table is called a "bucket". The average number of elements per bucket is called the "load factor", or z.

C. Lookup Within a Bucket

When looking up an item in a hash table by key k, the general strategy is to find the bucket that corresponds to k and then to perform a linear search within that bucket. The first step uses the hash function; the second step must use something else.

The most obvious technique is to use `std::find()` or the equivalent: look for an item whose key is equal to k. Naturally, it would be wrong for operator== to be hard-wired in; it should be possible for the user to provide a function object with semantics of equality. As an example where some predicate other than operator== is useful, suppose the user is storing C-style strings, i.e. pointers to null-terminated arrays of characters. In this case equality of keys k1 and k2 shouldn't mean pointer comparison; it should mean testing that strcmp(k1, k2) == 0.

This proposal takes such an approach. Unordered associative containers are parameterized by two function objects, a hash function and an equality function. Both have defaults.

An alternative technique is possible: instead of testing for equality, sort each bucket in ascending order by key. Linear search for a key k would mean searching for a key k' such that k < k' and k' < k are both false. Again, this shouldn't be taken to mean that operator< would literally be hard-wired in; users could provide their own comparison function, so long as that function has less-than semantics.

The performance characteristics of the two techniques are slightly different. The following table shows the average number of comparisons required for a search through a bucket of n elements:

	Using equality	Using less-than
Failed search	n	n/2
Successful search	n/2	n/2 + 1

The difference for a failed search is because with less-than you can tell that a search has failed as soon as you see a key that's larger than k; with equal-to you have to get to the end of the bucket. The difference for a successful search is because with equal-to you can tell that a search has succeeded as soon as you see a key that's equal to k; with less-than all you know when you find a key that's not less than k is that the search has terminated, and you need an extra comparison to tell whether it has terminated in success or failure.

I do not see a clear-cut performance advantage from either technique. Which technique is faster depends on usage pattern: the load factor, and the relative frequency of failed and successful searches. There are also performance implications for insertion, but I expect those differences to be smaller because in most cases I expect insertion to be dominated by the cost of memory allocation or element construction, not by the cost of lookup.

For users, it's sometimes important for a container to present its elements as a sorted range; sorting elements by inserting them into an `std::set`, for example, is a common idiom. However, I see no value (other than the performance issues discussed above) in sorting elements within a single bucket. If the hash function is well chosen, after all, elements will be distributed between buckets in a seemingly random way. I believe it is more helpful to tell users that they should use the existing associative containers (`set`, `map`, `multiset`, `multimap`) when they need useful guarantees on element ordering. The choice of names like `unordered_set` and `unordered_map` should help with that guidance.

From the point of view of user convenience, there isn't a huge difference between the two alternatives of equal-to and less-than. I view equality as slightly more convenient, since it's common to define data types that have equality operations but not less-than operations, and rather less common to do the reverse. There are some types where less-than is not a natural operation, and users would have to define a somewhat arbitrary less-than operation for no reason other than to put the objects in a hash table. One obvious example is `std::complex<>`.

Existing implementations differ. The SGI and Metrowerks implementations use equal-to, and the Dinkumware implementation uses less-than.

An aside: in principle, linear search isn't strictly necessary. A bucket doesn't have to be structured as a linked list; it could be structured as a binary tree, or as some other data structure. This proposal assumes linked lists, partly for reasons of existing practice (all of the C++ hash table implementations in widespread use are implemented in terms of linked lists) and partly because I believe that in practice a tree structure would hurt performance more often than it would help. Balanced trees have large per-node space overhead, and binary tree lookup is faster than linear search only when the number of elements is large. If the hash table's load factor is small and the hash function well chosen, trees have no advantage over linear lists.

D. The Hash Function Interface

Abstractly, a hash function is a function f(k) that takes an argument of type Key and returns an integer in the range [0, B), where B is the number of buckets in the hash table. A hash function must have the property that f(k1) == f(k2) when k1 and k2 are the same. A good hash function should also have the property that f(k1) and f(k2) are unlikely to be the same when k1 and k2 are different.

It is impossible to write a fully general hash function that's valid for all types. (You can't just convert an object to raw memory and hash the bytes; among other reasons, that idea fails because of padding.) Because of that, and also because a good hash function is only good in the context of a specific usage pattern, it's essential to allow users to provide their own hash functions.

There can be a default hash function for a selected set of types; ideally, it should include the most commonly used types.

There are two design decisions involving non-default hash functions:

1. How can a user-written function return an integer in the range [0, B) for arbitrary B, especially since B may vary at runtime?
2. Should the hash function be a standalone function object, or part of a larger package that controls other aspects of the hash policy?

In principle there are two possible answers to the first question. First, the hash function could take two arguments instead of one, where the second argument is B. The hash function would have the responsibility of returning some number in the range [0, B). Second, the hash function could return a number in a very large range, say [0, `std::numeric_limits<std::size_t>::max()`). The hash table class would be responsible for converting the hash code (the value returned by the hash function) into a bucket index in the range [0, B).

This proposal uses a single-argument hash function. The reasons are:

- Existing practice. The three shipping hash table implementations all use single-argument hash functions.
- The main advantage of a two-argument hash function is that a user-written hash function might make good use of the bucket count; it might, for example, involve multiplication modulo B. However, this advantage is weaker than it might seem, since a user-written hash function that took B as an argument would have to cope with arbitrary B. (B is chosen by the hash table class, not the user.) This is a significant restriction, because it means a user-written hash function couldn't rely on any special numerical properties of B.
- As discussed in section III.E, the bucket count won't always remain the same during the lifetime of a hash table. This means that, for a particular key k, the hash table class may need the bucket index for two different bucket counts B1 and B2. With a double-argument hash function, the hash table class would have to call the hash function twice. With a single-argument hash function, the hash table class only has to invoke the hash function once.

If the hash table should be packaged along with other aspects of hash policy, what should those aspects be? There are two obvious candidates. First, it could be packaged along with the the function object that tests for key equality, or perhaps, even more generally, with a function object that specifies a policy for linear search within a bucket. (See section III.C.) Second, it could be packaged along with the parameters that govern changing the bucket count. (See section III.E.)

This proposal uses a standalone hash function, rather than a hash function that's part of a policy package. This is mostly a consequence of other design decisions. First, bucket resizing is determined by floating-point parameters that can be changed at runtime (see III.E), so there is no advantage in putting them in a policy class. Second, linear search within a bucket uses equality (see III.C), and equality is such a common operation that in most cases I expect that a user-supplied equality predicate will have been written for some other purpose, and will be reused as a hash table template argument. Making equality part of a larger policy class would make such reuse harder.

This proposal includes a function object hash<>, with an operator() that takes a single argument and returns an std::size_t. The hash<> template is an incomplete type; it is specialized, and declared as a complete type, for a few common types. I've chosen all of the built-in integer types, all floating-point types, all pointer types, and std::basic_string<charT, traits, Allocator>. I believe that std::basic_string is especially important, because hash tables are often used for strings. Beyond that, the list is fairly arbitrary.

Some earlier hash table implementations gave char* special treatment: it specialized the default hash function to look at character array being pointed to, rather than the pointer itself. This proposal removes that special treatment. Special treatment makes it slightly easier to use hash tables for C string, but at the cost of removing uniformity and making it harder to write generic code. Since naive users would generally be expected to use std::basic_string instead of C strings, the cost of special treatment outweighs the benefit.

The hash<> function object is defined in the <functional> header. Another sensible alternative would have been to declare it in both <unordered_set> and <unordered_map>. Implementers would have to arrange for there to be only a single definition when both headers are used, but that's straightforward. The main reason I chose to put it in <functional> is that authors of user-defined types may need to specialize hash<> (which means they need the declaration) even if they have no need to use any of the hashed containers.

Trivial as it may seem, hash function packaging may be the most contentious part of this proposal. Existing implementations differ. The SGI and Metrowerks implementations use hash functions that aren't bundled with anything else, but the Dinkumware implementation uses a more general hash policy class. Since this is an interface issue, a decision is necessary.

E. Control of Hash Resizing

The time required for looking up an element by key k is $c_1 + c_2$ n, where c_1 and c_2 are constants, and where n is the number of elements in the bucket indexed by k's hash code. If the hash function is well chosen, and elements are evenly distributed between buckets, this is approximately $c_1 + c_2$ N/B, where N is the number of elements in the container and B is the bucket count. If the bucket count is taken as a constant, then the asymptotic complexity for element lookup is O(N).

To maintain average case complexity O(1) for lookup, the bucket count must grow as elements are added to the hash table; on average the bucket count must be proportional to N. Another way of putting this is that the load factor, N/B, must be approximately constant.

Two methods of maintaining a roughly constant load factor are in current use.

First is traditional rehashing. When the load factor becomes too large, choose a new and larger bucket count, B'. Then go through every element in the hash table, computing a new bucket index based on B'. This is an expensive operation. Since we want the amortized complexity of element insertion to be constant, we must use exponential growth; that is, $B' = \gamma B$, where the growth factor, γ, is larger than 1. In general this proportionality can only be approximate, since many hashing schemes require B to have special numerical properties—primality, for example.

Second is a newer technique, incremental hashing. (See [Plauger 1998].) Incremental rehashing structures the hash table in such a way that it is possible to add a single bucket at a time. When adding a bucket it is only necessary to examine the elements of a single old bucket, distributing some of them to the new one.

The advantage of incremental hashing is that insertion time becomes more predictable: there's no longer a large time difference between insertions that do trigger rehashing and insertions that don't. The disadvantage is that incremental hashing makes lookup slightly slower. The slowdown is for two reasons. First, the logic to determine a bucket index from a hash code is slightly more complicated: it requires one extra test. Second, incremental hashing results in a somewhat less uniform distribution of elements within buckets. It relies on a construction where there are conceptually B buckets, of which U are in current use; B is a power of 2, and $U > B/2$. We first find a bucket index i in the range [0, B), and then find a bucket index j in the range [0, U) by subtracting U from i if necessary. If the original hash codes are evenly distributed, a bucket in the range [0, B-U) will on average have twice the number of elements as a bucket in the range [B-U, U).

Because of this tradeoff, there is not a clear choice between incremental hashing and traditional rehashing; both are legitimate implementation techniques. A standard, of course, need not and should not dictate implementation technique. The goal of this proposal is to allow both.

From a user's perspective, all of this is invisible in normal use. It's visible when users want to do one of these things:

- Examine the number of buckets, or the distribution of elements between buckets. (See III.G.)
- Request a rehash even when the load factor isn't yet large enough so that one would be triggered automatically. Suppose, for example, that a hash table currently contains 100 elements, and a user is about to add 1000000 elements. A user would save a lot of time by requesting a rehash before performing those insertions.
- Control the parameters that govern automatic rehashing.

What are those parameters? The most obvious is the maximum load factor, since that's what triggers an automatic rehash. There's also a second parameter, which can be thought of in two different ways: as a growth factor (the constant of proportionality by which the bucket count grows in a rehash) or as a minimum load factor.

Letting users control that second parameter is more complicated than it seems at first.

- For incremental hashing, it makes no sense to talk about a growth factor; incremental hashing doesn't work by exponential growth.
- Even if we regard the second parameter as a minimum load factor, it's not clear how it would be used in an incremental-rehashing implementation. The natural strategy for incremental rehashing is to add one bucket whenever the load factor exceeds the maximum; this automatically keeps the load factor within a tight range.
- The interactions between a minimum load factor and manual rehash are tricky. Again, consider the example where a hash table has 100 elements, and the user, anticipating a large insertion, requests that the table rehash itself for 1000000 elements. In between the rehash and the insertion, the load factor will be very small—probably below reasonable values of the minimum.
- A minimum load factor presents difficulties for the hash table's initial state. An empty hash table—one that has a nonzero bucket count, but zero elements—has a load factor of zero. If we impose an invariant that the load factor always lie between z_{min} and z_{max}, an empty hash table would fail to satisfy that invariant.
- The second parameter, however it's expressed, would probably have to be interpreted in some approximate or asymptotic sense. The number of buckets must always be an integer, and in some implementations it has to obey other constraints. (e.g. primality) If we specify an exact growth factor, or a tight minimum and maximum, then there might simply be no suitable number available.

I don't know how to specify invariants that are precise enough to be meaningful and normative, but loose enough to accommodate traditional and incremental hashing, empty hash tables, hash tables with bucket counts that are restricted to prime numbers, manual rehashing, and empty hash tables. We could include a member function for setting the growth factor (or minimum load factor) but not say exactly how that number is used. However, I see very little value in such a vacuous requirement. This proposal provides user control of the maximum load factor, but not of growth factor or minimum load factor.

One implication is that this proposal says what happens as the number of elements in a hash table increase, but doesn't say what happens as the number decreases. This is unfortunate. An unnecessarily low load factor wastes space (the magnitude of the load factor is a time/space tradeoff), and can lead to unnecessarily slow iteration.

There's still one last question about the maximum load factor: how should the user specify it? An integer is an unnecessarily restrictive choice, since fractional values (especially, ones in the range $0 < z_{max} < 1$) are sensible. There are three reasonable options: as a rational number (perhaps an ad hoc rational number, where the user provides the numerator and denominator separately), as an enum (the user may select one of a small number of predetermined values, such as 1/4, 1/2, 1, 3/2, 2), or as a floating-point number.

This proposal provides a member function that allows the user to set the maximum load factor at runtime, using a floating-point number. The reasons are:

- In my opinion, there is no compelling performance advantage gained from setting the maximum load factor at compile time rather than at runtime. The cost of a runtime floating-point parameter is one floating-point multiplication at every rehash (*not* every insertion). Even with incremental hashing, this is almost certain to be dwarfed by the cost of a rehash. And, except when using a compile-time constant of 1, the costs of an integer or rational limit only slightly less in any case.
- An enum is less flexible from the user's point of view, because a user might want a load factor that's not in the predetermined list. Ad hoc rational numbers are just clumsy, both for the user and for the implementer.

Should the floating-point parameter be of type `float`, or of type `double`? It makes very little difference. On the one hand, `double` is typically the "natural" floating-point type that is used in the absence of a strong reason to the contrary. On the other hand, `float` may allow the hash table implementation to save some space, and may alert users to the fact that the value will not be used in any context that involves high precision. I have chosen `float`.

The Dinkumware hash table implementation uses a compile-time integer constant (part of a hash traits class) to control the maximum load factor, and the Metrowerks implementation uses a runtime floating-point parameter. The SGI implementation does not provide any mechanism for controlling the maximum load factor.

F. Iterators

There is one basic decision to be made about hash table iterators, which can be expressed either from the implementer's or the user's point of view. From the implementer's point of view: are the buckets singly linked lists, or doubly linked lists? From the user's point of view: are the iterators forward iterators, or bidirectional iterators?

From the implementer's point of view, there's no question that doubly linked lists are much easier to work with. One advantage is that you don't have to maintain a separate list for each bucket. You can keep a single long list, taking care that elements within a bucket remain adjacent; a bucket is then just a pair of pointers into the list. This is nice for the implementer, because a hash table iterator can just be a recycled std::list<>::iterator. It's nice for the user, because iteration is fast. I don't know of a way to make the single-long-list technique work for singly linked lists: some operations that ought to be constant time would require a linear search through buckets. (The sticking point turns out to be that erasing a node in a singly linked list requires access to the node before it. It's possible to get around this problem, but every technique I know of ends up introducing linear time behavior somewhere else.)

From the user's point of view, the choice is a tradeoff. Singly linked lists have slower iterators, because an iterator first steps within a bucket and then, upon reaching the end of a bucket, steps to the next. Additionally, users may sometimes want to apply algorithms that require bidirectional iterators. If a hash table supplies bidirectional iterators, it's easier for users to switch between (say) unordered_set<> and std::set<>. (But "easier" still doesn't mean easy. Some applications use the standard associative containers because of those containers' ordering guarantees, which, as the name suggests, aren't preserved by unordered associative containers.)

For the user, the disadvantage of bidirectional iterators is greater space overhead. The space overhead for singly linked lists is N + B words, where N is the number of elements and B is the bucket count, and the space overhead for doubly linked lists is 2N + 2B. This is an important consideration, because the main reason for using hashed associative containers is performance.

The SGI and Metrowerks implementations provide forward iterators. The Dinkumware implementation provides bidirectional iterators.

This proposal allows both choices. It requires hashed associative containers to provide forward iterators. An implementation that provides bidirectional iterators is conforming, because bidirectional iterators are forward iterators.

G. Bucket Interface

Like all standard containers, each of the hashed containers has member function begin() and end(). The range [c.begin(), c.end()) contains all of the elements in the container, presented as a flat range. Elements within a bucket are adjacent, but the iterator interface presents no information about where one bucket ends and the next begins.

It's also useful to expose the bucket structure, for two reasons. First, it lets users investigate how well their hash function performs: it lets them test how evenly elements are distributed within buckets, and to look at the elements within a bucket to see if they have any common properties. Second, if the iterators have an underlying segmented structure (as they do in existing singly linked list implementations), algorithms that exploit that structure, with an explicit nested loop, can be more efficient than algorithms that view the elements as a flat range.

The most important part of the bucket interface is an overloading of begin() and end(). If n is an integer, [begin(n), end(n)) is a range of iterators pointing to the elements in the nth bucket. These member functions return iterators, of course, but not of type

X::iterator or X::const_iterator. Instead they return iterators of type X::local_iterator or X::const_local_iterator. A local iterator is able to iterate within a bucket, but not necessarily between buckets; in some implementations it's possible for X::local_iterator to be a simpler data structure than X::iterator. X::iterator and X::local_iterator are permitted to be the same type; implementations that use doubly linked lists will probably take advantage of that freedom.

It is likely that the bucket interface will change in the future. Other segmented containers may also want to define an interface that exposes the underlying segmentation, and greater experience with segmented containers may give us more insight into what a uniform interface should look like. We can't define a uniform interface for segmented containers until we've done it at least twice.

This bucket interface is not provided by the SGI, Dinkumware, or Metrowerks implementations. It is inspired partly by the Metrowerks collision-detection interface, and partly by earlier work (see [Austern 1998]) on algorithms for segmented containers.

H. Exception guarantees

The C++ Standard gives a minimum set of exceptions guarantees for library components. (Roughly: exceptions don't corrupt data structures or cause memory leaks.) There are two important questions we have to answer. First: which operations on hash tables, if any, provide a stronger guarantee? Second: what restrictions, if any, do we need to impose on the user-defined function objects, the hash function and the equality function, used to instantiate hash tables.

In practice, I believe there are only two interesting operations: erase and insert. Erase is an interesting operation because in general it must invoke both of these function objects and may therefore throw exceptions. We have to say something about the circumstances in which it may throw exceptions (answer: only when they're thrown from one of these function objects), and we need to say that `clear` may not throw exceptions even though it's defined in terms of erase.

Insert is interesting because we have to decide whether it's practical for the single-element insert to provide the stronger success-or-no-effect guarantee. I believe it is not.

In the simple case (no rehash is necessary), the strong guarantee is easy: we can invoke the hash code and find the appropriate bucket before performing any allocations. After that point, there isn't any need to modify any list pointers until all comparisons have been performed and the insertion point is known. The trouble comes if a rehash is necessary, and if the user-provided hash function throws an exception during the rehash. At that point it's likely that the data structures will have been

modified in unrecoverable ways (the only way to recover would involve invoking the hash function again), and the only way to ensure integrity of the data structures is to lose some or all elements.

What we can say is that single-element insert provides the strong guarantee if the hash function is guaranteed not to throw exceptions. Note that this is true for the default hash functions.

I. Stored hash codes

There is an interesting space/time tradeoff for hash table implementers: along with an element, should one store the element's hash code? This can improve speed in two ways. First, it makes rehashes faster, because there's no need to recompute the hash code of every element. Second, it may make searches faster: when searching through a bucket the implementation can compare hash codes before doing a full element comparison. This is two tests instead of one, but integer comparisons are inexpensive and full element comparisons may sometimes (for strings, for example) be expensive.

Again, my goal is neither to require nor to forbid stored hash codes. I don't know of an implementation that currently stores hash codes, but I also don't know of anything in this proposal that would forbid it.

One might imagine trying to achieve greater flexibility: allowing users to control whether or not hash codes are stored and used for searches, so that they're only stored in cases where the user believes that this would be a performance benefit. (One might imagine using a policy class, for example.) I haven't tried to provide that kind of flexibility, because I don't think the extra gain would be justified by the increased complexity of the interface.

IV. Proposed Text

A. Requirements

1. To be added as a separate requirements section, following clause 23.1.2

Unordered associative containers provide an ability for fast retrieval of data based on keys. The worst-case complexity for most operations is linear, but the average case is much faster. The library provides four basic kinds of hashed associative containers: `unordered_set`, `unordered_map`, `unordered_multiset`, and `unordered_multimap`.

Each hashed associative container is parameterized by `Key`, by a function object `Hash` that acts as a hash function for values of type `Key`, and on a binary predicate `Pred` that induces an equivalence relation on values of type `Key`. Additionally, `unordered_map` and `unordered_multimap` associate an arbitrary *mapped type* `T` with the `Key`.

A hash function is a function object that takes a single argument of type `Key` and returns a value of type `std::size_t` in the range `[0, std::numeric_limits<std::size_t>::max())`.

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

A hashed 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.

For `unordered_set` and `unordered_multiset` the value type is the same as the key type. For `unordered_map` and `unordered_multimap` it is equal to `std::pair<const Key, T>`.

The elements of a hashed associative container are organized into *buckets*. Keys with the same hash code appear in the same bucket. The number of buckets is automatically increased as elements are added to a hashed associative container, so that the average number of elements per bucket is kept below a bound. Rehashing invalidates iterators, changes ordering between elements, and changes which buckets elements appear in, but does not invalidate pointers or references to elements.

In the following table, `X` is a hashed associative container class, `a` is an object of type `X`, `b` is a possibly const object of type `X`, `a_uniq` is an object of type `X` when `X` supports unique keys, `a_eq` is an object of type `X` when `X` supports equivalent keys, `i` and `j` are input iterators that refer to `value_type`, `[i, j)` is a valid range, `p` and `q2` are valid iterators to `a`, `q` and `q1` are valid dereferenceable iterators to `a`, `[q1, q2)` is a valid range in `a`, `r` and `r1` are valid dereferenceable const iterators to `a`, `r2` is a valid const iterator to `a`, `[r1, r2)` is a valid range in `a`, `t` is a value of type `X::value_type`, `k` is a value of type `key_type`, `hf` is a possibly const value of type `hasher`, `eq` is a possibly const value of type `key_equal`, `n` is a value of type `size_type`, and `z` is a value of type `float`.

Unordered associative container requirements (in addition to container)			
Expression	**Return type**	**assertion/note pre/post-condition**	**complexity**
`X::key_type`	`Key`	`Key` is `Assignable` and `CopyConstructible`	compile time
`X::hasher`	`Hash`	`Hash` is a unary function object that take an argument of type `Key` and returns a value of type `std::size_t`.	compile time
`X::key_equal`	`Pred`	`Pred` is a binary predicate that takes two arguments of type `Key`. `Pred` is an equivalence relation.	compile time
`X::local_iterator`	An iterator type whose category, value type, difference type, and pointer and reference types are the same as `X::iterator`'s.	A `local_iterator` object may be used to iterate through a single bucket, but may not be used to iterated across buckets.	compile time
`X::const_local_iterator`	An iterator type whose category, value type, difference type, and pointer and reference types are the same as `X::const_iterator`'s.	A `const_local_iterator` object may be used to iterate through a single bucket, but may not be used to iterated across buckets.	compile time

`X(n, hf, eq)`
`X a(n, hf, eq)`
	X	Constructs an empty container with at least `n` buckets, using `hf` as the hash function and `eq` as the key equality predicate.	O(n)

`X(n, hf)`
`X a(n, hf)`
	X	Constructs an empty container with at least `n` buckets, using `hf` as the hash function and `key_equal()` as the	O(n)

		key equality predicate.	

`X(n)`
`X a(n)`
	X	Constructs an empty container with at least `n` buckets, using `hasher()` as the hash function and `key_equal()` as the key equality predicate.	O(n)

`X()`
`X a`
	X	Constructs an empty container with an unspecified number of buckets, using `hasher()` as the hash function and `key_equal` as the key equality predicate.	constant

`X(i, j, n, hf, eq)`
`X a(i, j, n, hf, eq)`
	X	Constructs an empty container with at least `n` buckets, using `hf` as the hash function and `eq` as the key equality predicate, and inserts elements from `[i, j)` into it.	Average case O(N) (N is `std::distance(i, j)`), worst case O(N^2)

`X(i, j, n, hf)`
`X a(i, j, n, hf)`
	X	Constructs an empty container with at least `n` buckets, using `hf` as the hash function and `key_equal()` as the key equality predicate, and inserts elements from `[i, j)` into it.	Average case O(N) (N is `std::distance(i, j)`), worst case O(N^2)

`X(i, j, n)`
`X a(i, j, n)`
	X	Constructs an empty container with at least `n` buckets, using `hasher()` as the hash function and `key_equal()` as the key equality predicate, and inserts elements from `[i, j)` into it.	Average case O(N) (N is `std::distance(i, j)`), worst case O(N^2)

`X(i, j)`
`X a(i, j)`
	X	Constructs an empty container with an unspecified number of buckets, using `hasher()` as the hash function and `key_equal` as the key equality predicate, and inserts elements from `[i, j)` into it.	Average case O(N) (N is `std::distance(i, j)`), worst case O(N^2)

`X(b)`
`X a(b)`
	`X`	Copy constructor. In addition to the contained elements, the hash function, predicate, and maximum load factor are copied.	Average case linear in `b.size()`, worst case quadratic.

`a = b`	`X`	Copy assignment operator. In addition to the contained elements, the hash function, predicate, and maximum load factor are copied.	Average case linear in `b.size()`, worst case quadratic.
`b.hash_function()`	`hasher`	Returns the hash function out of which `a` was constructed.	constant
`b.key_eq()`	`key_equal`	Returns the key equality function out of which `a` was constructed.	constant
`a_uniq.insert(t)`	`std::pair<iterator, bool>`	Inserts `t` if and only if there is no element in the container with key equivalent to the key of `t`. The `bool` component of the returned pair indicates whether the insertion takes place, and the `iterator` component points to the element with key	Average case O(1), worst case O(`a_uniq.size()`).

		equivalent to the key of `t`.	
`a_eq.insert(t)`	`iterator`	Inserts `t`, and returns an iterator pointing to the newly inserted element.	Average case O(1), worst case O(`a_uniq.size()`).
`a.insert(r, t)`	`iterator`	Equivalent to a.insert(t). Return value is an iterator pointing to the element with the key equivalent to that of `t`. The const iterator `r` is a hint pointing to where the search should start. Implementations are permitted to ignore the hint.	Average case O(1), worst case O(`a_uniq.size()`).
`a.insert(i, j)`	`void`	Pre: `i` and `j` are not iterators in `a`. Equivalent to `a.insert(t)` for each element in `[i,j)`.	Average case O(N), where N is `std::distance(i, j)`. Worst case O(N * `a.size()`).
`a.erase(k)`	`size_type`	Erases all elements with key equivalent to `k`. Returns the number of elements erased.	Average case O(`a.count(k)`). Worst case O(`a.size())`.
`a.erase(r)`	`void`	Erases the element pointed to by `r`.	Average case O(1), worst case O(`a.size()`).
`a.erase(r1, r2)`	`void`	Erases all elements in the range `[r1, t2)`.	Average case O(`std::distance(r1, r2)`), worst case O(`a.size()`).
`a.clear()`	`void`	Erases all elements in the container. Post: `a.size() == 0`	Linear.
`b.find(k)`	`iterator`; `const_iterator` for const `a`.	Returns an iterator pointing to an element with key equivalent to `k`, or	Average case O(1), worst case O(`a.size()`).

		`a.end()` if no such element exists.	
`b.count(k)`	`size_type`	Returns the number of elements with key equivalent to `k`.	Average case O(1), worst case O(`a.size()`).
`b.equal_range(k)`	`std::pair<iterator, iterator>`; `std::pair<const_iterator, const_iterator>` for const `b`.	Returns a range containing all elements with keys equivalent to `k`. Returns `std::make_pair(a.end(), a.end())` if no such elements exist.	Average case O(`a.count(k)`). Worst case O(`a.size()`).
`b.bucket_count()`	`size_type`	Returns the number of buckets that `b` contains.	Constant
`b.max_bucket_count()`	`size_type`	Returns an upper bound on the number of buckets that `b` might ever contain.	Constant
`b.bucket(k)`	`size_type`	Returns the index of the bucket in which elements with keys equivalent to `k` would be found, if any such element existed. Post: the return value is in the range `[0, b.bucket_count())`.	Constant
`b.bucket_size(n)`	`size_type`	Pre: `n` is in the range `[0, b.bucket_count())`. Returns the number of elements in the `n`th bucket.	O(`a.bucket_size(n)`)
`b.begin(n)`	`local_iterator`; `const_local_iterator` for const `b`	Pre: `n` is in the range `[0, b.bucket_count())`. Note: `[b.begin(n), b.end(n))` is a valid range containing all	Constant

		of the elements in the `n`th bucket.	
`b.end(n)`	`local_iterator`; `const_local_iterator` for const `b`	Pre: `n` is in the range `[0, b.bucket_count())`.	Constant
`b.load_factor()`	`float`	Returns the average number of elements per bucket.	Constant
`b.max_load_factor()`	`float`	Returns a number that the container attempts to keep the load factor less than or equal to. The container automatically increases the number of buckets as necessary to keep the load factor below this number. Post: return value is positive.	Constant
`a.max_load_factor(z)`	`void`	Pre: `z` is positive. Changes the container's maximum load load factor, using `z` as a hint.	Constant
`a.rehash(n)`	`void`	Pre: `n > a.size() / a.max_load_factor()`. Changes the number of buckets so that it is at least `n`.	Average case linear in `a.size()`, worst case quadratic.

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.]

The iterator types `iterator` and `const_iterator` of a hashed associative container are of at least the forward iterator category. For hashed associative containers where the key type and value type are the same, both `iterator` and `const_iterator` are const iterators.

The insert members shall not affect the validity of references to container elements, but may invalidate all iterators to the container. The erase members shall invalidate only iterators and references to the erased elements.

2. Exception safety guarantees

Add the following bullet items to the list of exception safety guarantees in clause 23.1, paragraph 10:

- **For hashed associative containers, no `clear()` function throws an exception. No `erase()` function throws an exception unless that exception is thrown by the container's Hash or Pred object (if any).**
- **For hashed associative containers, if an exception is thrown by an insert() function while inserting a single element other than by the container's hash function, the insert() function has no effects.**
- **For hashed associative containers, no `swap` function throws an exception unless that exception is thrown by the copy constructor or copy assignment operator of the container's Hash or Pred object (if any).**

B. Hash Function

1. To be added to the <functional> synopsis

```
// Hash function base template
template <class T> struct hash;

// Hash function specializations

template <> struct hash<bool>;
template <> struct hash<char>;
template <> struct hash<signed char>;
template <> struct hash<unsigned char>;
template <> struct hash<wchar_t>;
template <> struct hash<short>;
template <> struct hash<int>;
template <> struct hash<long>;
template <> struct hash<unsigned short>;
template <> struct hash<unsigned int>;
template <> struct hash<unsigned long>;

template <> struct hash<float>;
template <> struct hash<double>;
template <> struct hash<long double>;

template<class T>
struct hash<T*>

template <class charT, class traits, class Allocator>
struct hash<std::basic_string<charT, traits, Allocator> >;
```

2. Class template `hash`

The function object `hash` is used as the default hash function by the *hashed associative containers*. This class template is only required to be instantiable for integer types (3.9.1), floating point types (3.9.1), pointer types (8.3.1), and (for any valid set of `charT`, `traits`, and `Alloc`) `std::basic_string<charT, traits, Alloc>`.

```
    template <class T>
    struct hash : public std::unary_function<T, std::size_t>
    {
      std::size_t operator()(T val) const;
    };
```

The return value of `operator()` is unspecified, except that equal arguments yield the same result. `operator()` shall not throw exceptions.

C. Unordered Associative Containers

1. Header <unordered_set> synopsis

```
namespace std {
  template <class Value,
            class Hash = hash<Value>,
            class Pred = std::equal_to<Value>,
            class Alloc = std::allocator<Value> >
  class unordered_set;

  template <class Value,
            class Hash = hash<Value>,
            class Pred = std::equal_to<Value>,
            class Alloc = std::allocator<Value> >
  class unordered_multiset;
}
```

2. Header <unordered_map> synopsis

```
namespace std {
  template <class Key,
            class T,
            class Hash = hash<Key>,
            class Pred = std::equal_to<Key>,
            class Alloc = std::allocator<std::pair<const Key, T> > >
  class unordered_map;

  template <class Key,
            class T,
            class Hash = hash<Key>,
            class Pred = std::equal_to<Key>,
            class Alloc = std::allocator<std::pair<const Key, T> > >
  class unordered_multiset;
}
```

3. Class template `unordered_set`

An `unordered_set` is a kind of hashed associative container that supports unique keys (an `unordered_set` contains at most one of each key value) and in which the elements' keys are the elements themselves.

An `unordered_set` satisfies all of the requirements of a container and of a hashed associative container. It provides the operations described in the preceding requirements table for unique keys; that is, an `unordered_set` supports the `a_uniq` operations in that table, not the `a_eq` operations. For a `unordered_set<Value>` the `key type` and the value type are both `Value`. The `iterator` and `const_iterator` types are both const iterator types. It is unspecified whether or not they are the same type.

This section only describes operations on `unordered_set` that are not described in one of the requirement tables, or for which there is additional semantic information.

```
  namespace std {
    template <class Value,
              class Hash  = hash<Value>,
              class Pred  = std::equal_to<Value>,
              class Alloc = std::allocator<Value> >
    class unordered_set
    {
    public:
      // types
      typedef Value                                      key_type;
      typedef Value                                      value_type;
      typedef Hash                                       hasher;
      typedef Pred                                       key_equal;
      typedef Alloc
allocator_type;
      typedef typename allocator_type::pointer           pointer;
      typedef typename allocator_type::const_pointer     const_pointer;
      typedef typename allocator_type::reference         reference;
      typedef typename allocator_type::const_reference
const_reference;
      typedef implementation defined                     size_type;
      typedef implementation defined
difference_type;

      typedef implementation defined                     iterator;
      typedef implementation defined
const_iterator;
      typedef implementation defined
local_iterator;
      typedef implementation defined
const_local_iterator;


      // construct/destroy/copy
      explicit unordered_set(size_type n = implementation defined,
                             const hasher& hf = hasher(),
                             const key_equal& eql = key_equal(),
                             const allocator_type& a =
allocator_type());
      template <class InputIterator>
        unordered_set(InputIterator f, InputIterator l,
                      size_type n = implementation defined,
                      const hasher& hf = hasher(),
                      const key_equal& eql = key_equal(),
                      const allocator_type& a = allocator_type());
      unordered_set(const unordered_set&);
      ~unordered_set();
      unordered_set& operator=(const unordered_set&);
      allocator_type get_allocator() const;

      // size and capacity
      bool empty() const;
      size_type size() const;
      size_type max_size() const;

      // iterators
      iterator       begin();
      const_iterator begin() const;
      iterator       end();
      const_iterator end() const;

      // modifiers
      std::pair<iterator, bool> insert(const value_type& obj);
      iterator insert(const_iterator hint, const value_type& obj);
      template <class InputIterator>
        void insert(InputIterator first, InputIterator last);

      void erase(const_iterator position);
      size_type erase(const key_type& k);
      void erase(const_iterator first, const_iterator last);
      void clear();

      void swap(unordered_set&);

      // observers
      hasher hash_function() const;
      key_equal key_eq() const;

      // lookup
      iterator       find(const key_type& k);
      const_iterator find(const key_type& k) const;
      size_type count(const key_type& k) const;
      std::pair<iterator, iterator>
        equal_range(const key_type& k);
      std::pair<const_iterator, const_iterator>
        equal_range(const key_type& k) const;

      // bucket interface
      size_type bucket_count() const;
      size_type max_bucket_count() const;
      size_type bucket_size(size_type n);
      size_type bucket(const key_type& k);
      local_iterator begin(size_type n);
      const_local_iterator begin(size_type n) const;
      local_iterator end(size_type n);
      const_local_iterator end(size_type n) const;
      // hash policy
      float load_factor() const;
      float max_load_factor() const;
      void max_load_factor(float z);
      void rehash(size_type n);
    };

    template <class Value, class Hash, class Pred, class Alloc>
      void swap(const unordered_set<Value, Hash, Pred, Alloc>& x,
                const unordered_set<Value, Hash, Pred, Alloc>& y);

  }
```

a. `unordered_set` **constructors**

```
      explicit unordered_set(size_type n = implementation defined,
                             const hasher& hf = hasher(),
                             const key_equal& eql = key_equal(),
                             const allocator_type& a =
allocator_type());
```

Effects: Constructs an empty `unordered_set` **using the specified hash function, key equality function, and allocator, and using at least** ***n*** **buckets. If** ***n*** **is not provided, the number of buckets is implementation defined.** `max_load_factor()` **is 1.0.**

Complexity: Constant.

```
      template <class InputIterator>
        unordered_set(InputIterator f, InputIterator l,
                 size_type n = implementation defined,
                      const hasher& hf = hasher(),
                      const key_equal& eql = key_equal(),
                      const allocator_type& a = allocator_type());
```

Effects: Constructs an empty `unordered_set` **using the specified hash function, key equality function, and allocator, and using at least** ***n*** **buckets. (If** ***n*** **is not provided, the number of buckets is implementation defined.) Then inserts elements from the range** `[`*`first`*`, `*`last`*`)`**.** `max_load_factor()` **is 1.0.**

Complexity: Average case linear, worst case quadratic.

b. `unordered_set swap`

```
      template <class Value, class Hash, class Pred, class Alloc>
        void swap(const unordered_set<Value, Hash, Pred, Alloc>& x,
                  const unordered_set<Value, Hash, Pred, Alloc>& y);
```

Effects:

```
        x.swap(y);
```

4. Class template `unordered_map`

An `unordered_map` is a kind of hashed associative container that supports unique keys (an `unordered_map` contains at most one of each key value) and that associates values of another type `mapped_type` with the keys.

An `unordered_map` satisfies all of the requirements of a container and of a hashed associative container. It provides the operations described in the preceding requirements table for unique keys; that is, an `unordered_map` supports the `a_uniq` operations in that table, not the `a_eq` operations. For a `unordered_map<Key, T>` the `key type` is `Key`, the mapped type is `T`, and the value type is `std::pair<const Key, T>`.

This section only describes operations on `unordered_map` that are not described in one of the requirement tables, or for which there is additional semantic information.

```
  namespace std {
    template <class Key,
              class T,
              class Hash  = hash<Key>,
              class Pred  = std::equal_to<Key>,
              class Alloc = std::allocator<std::pair<const Key,
T> > >
    class unordered_map
    {
    public:
      // types
      typedef Key                                     key_type;
      typedef std::pair<const Key, T>                 value_type;
      typedef T                                       mapped_type;
      typedef Hash                                    hasher;
      typedef Pred                                    key_equal;
      typedef Alloc
allocator_type;
      typedef typename allocator_type::pointer        pointer;
      typedef typename allocator_type::const_pointer  const_pointer;
      typedef typename allocator_type::reference      reference;
      typedef typename allocator_type::const_reference
const_reference;
      typedef implementation defined                  size_type;
      typedef implementation defined
difference_type;

      typedef implementation defined                  iterator;
      typedef implementation defined
const_iterator;
      typedef implementation defined
local_iterator;
      typedef implementation defined
const_local_iterator;
```

```
      // construct/destroy/copy
      explicit unordered_map(size_type n = implementation defined,
                             const hasher& hf = hasher(),
                             const key_equal& eql = key_equal(),
                             const allocator_type& a =
allocator_type());
      template <class InputIterator>
        unordered_map(InputIterator f, InputIterator l,
                      size_type n = implementation defined,
                      const hasher& hf = hasher(),
                      const key_equal& eql = key_equal(),
                      const allocator_type& a = allocator_type());
      unordered_map(const unordered_map&);
      ~unordered_map();
      unordered_map& operator=(const unordered_map&);
      allocator_type get_allocator() const;

      // size and capacity
      bool empty() const;
      size_type size() const;
      size_type max_size() const;

      // iterators
      iterator       begin();
      const_iterator begin() const;
      iterator       end();
      const_iterator end() const;

      // modifiers
      std::pair<iterator, bool> insert(const value_type& obj);
      iterator insert(const_iterator hint, const value_type& obj);
      template <class InputIterator>
        void insert(InputIterator first, InputIterator last);

      void erase(const_iterator position);
      size_type erase(const key_type& k);
      void erase(const_iterator first, const_iterator last);
      void clear();

      void swap(unordered_map&);

      // observers
      hasher hash_function() const;
      key_equal key_eq() const;

      // lookup
      iterator       find(const key_type& k);
      const_iterator find(const key_type& k) const;
      size_type count(const key_type& k) const;
      std::pair<iterator, iterator>
        equal_range(const key_type& k);
      std::pair<const_iterator, const_iterator>
        equal_range(const key_type& k) const;

      mapped_type& operator[](const key_type& k);

      // bucket interface
      size_type bucket_count() const;
      size_type max_bucket_count() const;
      size_type bucket_size(size_type n);
      size_type bucket(const key_type& k);
      local_iterator begin(size_type n);
      const_local_iterator begin(size_type n) const;
      local_iterator end(size_type n);
      const_local_iterator end(size_type n) const;

      // hash policy
      float load_factor() const;
      float max_load_factor() const;
      void max_load_factor(float z);
      void rehash(size_type n);
    };

    template <class Key, class T, class Hash, class Pred, class
Alloc>
      void swap(const unordered_map<Key, T, Hash, Pred, Alloc>& x,
                const unordered_map<Key, T, Hash, Pred, Alloc>& y);
  }
```

a. `unordered_map` **constructors**

```
      explicit unordered_map(size_type n = implementation defined,
                             const hasher& hf = hasher(),
                             const key_equal& eql = key_equal(),
                             const allocator_type& a =
allocator_type());
```

Effects: Constructs an empty `unordered_map` **using the specified hash function, key equality function, and allocator, and using at least** *`n`* **buckets. If** *`n`* **is not provided, the number of buckets is implementation defined.** `max_load_factor()` **is 1.0.**

Complexity: Constant.

```
      template <class InputIterator>
        unordered_map(InputIterator f, InputIterator l,
                      size_type n = implementation defined,
                      const hasher& hf = hasher(),
                      const key_equal& eql = key_equal(),
                      const allocator_type& a = allocator_type());
```

Effects: Constructs an empty `unordered_map` **using the specified hash function, key equality function, and allocator, and using at least** *`n`* **buckets. (If** *`n`* **is not provided, the number of buckets is implementation defined.) Then inserts elements from the range** `[`*`first`*`, `*`last`*`)`**.** `max_load_factor()` **is 1.0.**

Complexity: Average case linear, worst case quadratic.

b. `unordered_map` **element access**

```
      mapped_type& operator[](const key_type& k);
```

Effects: If the `unordered_map` does not already contain an element whose key is equivalent to *`k`*, inserts `std::pair<const key_type, mapped_type>(k, mapped_type()).`

Returns: A reference to `x.second`, where `x` is the (unique) element whose key is equivalent to *`k`*.

c. `unordered_map swap`

```
      template <class Value, class Hash, class Pred, class Alloc>
        void swap(const unordered_map<Value, Hash, Pred, Alloc>& x,
                  const unordered_map<Value, Hash, Pred, Alloc>& y);
```

Effects:

```
          x.swap(y);
```

5. Class template `unordered_multiset`

An `unordered_multiset` is a kind of hashed associative container that supports equivalent keys (an `unordered_multiset` may contain multiple copies of the same key value) and in which the elements' keys are the elements themselves.

An `unordered_multiset` satisfies all of the requirements of a container and of a hashed associative container. It provides the operations described in the preceding requirements table for equivalent keys; that is, an `unordered_multiset` supports the `a_eq` operations in that table, not the `a_uniq` operations. For a `unordered_multiset<Value>` the `key type` and the value type are both `Value`. The `iterator` and `const_iterator` types are both const iterator types. It is unspecified whether or not they are the same type.

This section only describes operations on `unordered_multiset` that are not described in one of the requirement tables, or for which there is additional semantic information.

```
  namespace std {
    template <class Value,
              class Hash  = hash<Value>,
              class Pred  = std::equal_to<Value>,
              class Alloc = std::allocator<Value> >
    class unordered_multiset
    {
    public:
      // types
      typedef Value                                     key_type;
      typedef Value                                     value_type;
      typedef Hash                                      hasher;
      typedef Pred                                      key_equal;
      typedef Alloc
allocator_type;
      typedef typename allocator_type::pointer         pointer;
      typedef typename allocator_type::const_pointer   const_pointer;
      typedef typename allocator_type::reference       reference;
      typedef typename allocator_type::const_reference
const_reference;
      typedef implementation defined                    size_type;
      typedef implementation defined
difference_type;

      typedef implementation defined                    iterator;
      typedef implementation defined
const_iterator;
      typedef implementation defined
local_iterator;
      typedef implementation defined
const_local_iterator;


      // construct/destroy/copy
      explicit unordered_multiset(size_type n = implementation
defined,
                                  const hasher& hf = hasher(),
                                  const key_equal& eql = key_equal(),
                                  const allocator_type& a =
allocator_type());
      template <class InputIterator>
        unordered_multiset(InputIterator f, InputIterator l,
                      size_type n = implementation defined,
                      const hasher& hf = hasher(),
                      const key_equal& eql = key_equal(),
                      const allocator_type& a = allocator_type());
      unordered_multiset(const unordered_multiset&);
      ~unordered_multiset();
      unordered_multiset& operator=(const unordered_multiset&);
      allocator_type get_allocator() const;

      // size and capacity
      bool empty() const;
      size_type size() const;
      size_type max_size() const;

      // iterators
      iterator       begin();
      const_iterator begin() const;
      iterator       end();
      const_iterator end() const;

      // modifiers
      iterator insert(const value_type& obj);
      iterator insert(const_iterator hint, const value_type& obj);
      template <class InputIterator>
        void insert(InputIterator first, InputIterator last);

      void erase(const_iterator position);
      size_type erase(const key_type& k);
      void erase(const_iterator first, const_iterator last);
      void clear();

      void swap(unordered_multiset&);

      // observers
      hasher hash_function() const;
      key_equal key_eq() const;

      // lookup
      iterator       find(const key_type& k);
      const_iterator find(const key_type& k) const;
      size_type count(const key_type& k) const;
      std::pair<iterator, iterator>
        equal_range(const key_type& k);
      std::pair<const_iterator, const_iterator>
        equal_range(const key_type& k) const;

      // bucket interface
      size_type bucket_count() const;
      size_type max_bucket_count() const;
      size_type bucket_size(size_type n);
      size_type bucket(const key_type& k);
      local_iterator begin(size_type n);
      const_local_iterator begin(size_type n) const;
      local_iterator end(size_type n);
      const_local_iterator end(size_type n) const;

      // hash policy
      float load_factor() const;
      float max_load_factor() const;
      void max_load_factor(float z);
      void rehash(size_type n);
    };

    template <class Value, class Hash, class Pred, class Alloc>
      void swap(const unordered_multiset<Value, Hash, Pred, Alloc>&
x,
                const unordered_multiset<Value, Hash, Pred, Alloc>&
y);
  }
```

a. `unordered_multiset` constructors

```
      explicit unordered_multiset(size_type n = implementation
defined,
                                  const hasher& hf = hasher(),
                                  const key_equal& eql = key_equal(),
                                  const allocator_type& a =
allocator_type());
```

Effects: Constructs an empty `unordered_multiset` using the specified hash function, key equality function, and allocator, and using at least *n* buckets. If *n* is not provided, the number of buckets is implementation defined. `max_load_factor()` is 1.0.

Complexity: Constant.

```
      template <class InputIterator>
        unordered_multiset(InputIterator f, InputIterator l,
                           size_type n = implementation defined,
                           const hasher& hf = hasher(),
                           const key_equal& eql = key_equal(),
                           const allocator_type& a =
allocator_type());
```

Effects: Constructs an empty `unordered_multiset` using the specified hash function, key equality function, and allocator, and using at least *`n`* buckets. (If *`n`* is not provided, the number of buckets is implementation defined.) Then inserts elements from the range `[`*`first`*`, `*`last`*`)`. `max_load_factor()` is 1.0.

Complexity: Average case linear, worst case quadratic.

b. `unordered_multiset swap`

```
      template <class Value, class Hash, class Pred, class Alloc>
        void swap(const unordered_multiset<Value, Hash, Pred, Alloc>&
x,
                  const unordered_multiset<Value, Hash, Pred, Alloc>&
y);
```

Effects:

```
          x.swap(y);
```

6. Class template `unordered_multimap`

An `unordered_multimap` is a kind of hashed associative container that supports equivalent keys (an `unordered_multimap` may contain multiple copies of each key value) and that associates values of another type `mapped_type` with the keys.

An `unordered_multimap` satisfies all of the requirements of a container and of a hashed associative container. It provides the operations described in the preceding requirements table for equivalent keys; that is, an `unordered_multimap` supports the `a_eq` operations in that table, not the `a_uniq` operations. For an `unordered_multimap<Key, T>` the `key type` is `Key`, the mapped type is `T`, and the value type is `std::pair<const Key, T>`.

This section only describes operations on `unordered_multimap` that are not described in one of the requirement tables, or for which there is additional semantic information.

```
  namespace std {
    template <class Key,
              class T,
              class Hash  = hash<Key>,
              class Pred  = std::equal_to<Key>,
              class Alloc = std::allocator<std::pair<const Key,
T> > >
    class unordered_multimap
    {
    public:
      // types
      typedef Key                                        key_type;
      typedef std::pair<const Key, T>                    value_type;
      typedef T                                          mapped_type;
      typedef Hash                                       hasher;
      typedef Pred                                       key_equal;
      typedef Alloc
allocator_type;
      typedef typename allocator_type::pointer         pointer;
      typedef typename allocator_type::const_pointer   const_pointer;
      typedef typename allocator_type::reference       reference;
      typedef typename allocator_type::const_reference
const_reference;
      typedef implementation defined                   size_type;
      typedef implementation defined
difference_type;

      typedef implementation defined                   iterator;
      typedef implementation defined
const_iterator;
      typedef implementation defined
local_iterator;
      typedef implementation defined
const_local_iterator;

      // construct/destroy/copy
      explicit unordered_multimap(size_type n = implementation
defined,
                            const hasher& hf = hasher(),
                            const key_equal& eql = key_equal(),
                            const allocator_type& a =
allocator_type());
      template <class InputIterator>
        unordered_multimap(InputIterator f, InputIterator l,
                       size_type n = implementation defined,
                       const hasher& hf = hasher(),
                       const key_equal& eql = key_equal(),
                       const allocator_type& a =
allocator_type());
      unordered_multimap(const unordered_multimap&);
      ~unordered_multimap();
      unordered_multimap& operator=(const unordered_multimap&);
      allocator_type get_allocator() const;

      // size and capacity
      bool empty() const;
      size_type size() const;
      size_type max_size() const;

      // iterators
      iterator       begin();
      const_iterator begin() const;
      iterator       end();
      const_iterator end() const;

      // modifiers
      iterator insert(const value_type& obj);
      iterator insert(const_iterator hint, const value_type& obj);
      template <class InputIterator>
        void insert(InputIterator first, InputIterator last);

      void erase(const_iterator position);
      size_type erase(const key_type& k);
      void erase(const_iterator first, const_iterator last);
      void clear();

      void swap(unordered_multimap&);

      // observers
      hasher hash_function() const;
      key_equal key_eq() const;

      // lookup
      iterator       find(const key_type& k);
      const_iterator find(const key_type& k) const;
      size_type count(const key_type& k) const;
      std::pair<iterator, iterator>
        equal_range(const key_type& k);
      std::pair<const_iterator, const_iterator>
        equal_range(const key_type& k) const;

      mapped_type& operator[](const key_type& k);

      // bucket interface
      size_type bucket_count() const;
      size_type max_bucket_count() const;
      size_type bucket_size(size_type n);
      size_type bucket(const key_type& k);
      local_iterator begin(size_type n);
      const_local_iterator begin(size_type n) const;
      local_iterator end(size_type n);
      const_local_iterator end(size_type n) const;

      // hash policy
      float load_factor() const;
      float max_load_factor() const;
      void max_load_factor(float z);
      void rehash(size_type n);
    };

    template <class Key, class T, class Hash, class Pred, class
Alloc>
      void swap(const unordered_multimap<Key, T, Hash, Pred, Alloc>&
x,
                const unordered_multimap<Key, T, Hash, Pred, Alloc>&
y);
  }
```

a. `unordered_multimap` constructors

```
      explicit unordered_multimap(size_type n = implementation
defined,
                                  const hasher& hf = hasher(),
                                  const key_equal& eql = key_equal(),
                                  const allocator_type& a =
allocator_type());
```

Effects: Constructs an empty `unordered_multimap` using the specified hash function, key equality function, and allocator, and using at least *`n`* buckets. If *`n`* is not provided, the number of buckets is implementation defined. `max_load_factor()` is 1.0.

Complexity: Constant.

```
      template <class InputIterator>
        unordered_multimap(InputIterator f, InputIterator l,
                           size_type n = implementation defined,
                           const hasher& hf = hasher(),
                           const key_equal& eql = key_equal(),
                           const allocator_type& a =
allocator_type());
```

Effects: Constructs an empty `unordered_multimap` using the specified hash function, key equality function, and allocator, and using at least *`n`* buckets. (If *`n`* is not provided, the number of buckets is implementation defined.) Then inserts elements from the range `[`*`first`*`, `*`last`*`)`. `max_load_factor()` is 1.0.

Complexity: Average case linear, worst case quadratic.

b. `unordered_multimap swap`

```
      template <class Value, class Hash, class Pred, class Alloc>
        void swap(const unordered_multimap<Value, Hash, Pred, Alloc>&
x,
                  const unordered_multimap<Value, Hash, Pred, Alloc>&
y);
```

Effects:

```
          x.swap(y);
```

V. Unresolved issues

The following issues have been raised and not addressed, or have been addressed in a way that some people may consider inadequate.

1. Naming: Howard proposes changing the name `rehash` to an overload of `bucket_count`. Should we do that? I've chosen not to, because I believe it would be too misleading: this operation does not necessarily change the bucket count to the value the user requests. Besides, "rehash" is a commonly used name for this operation.

2. Naming: what should be the name for `hash_function`'s return type? This proposal, following the original Barreiro/Fraley/Musser proposal, chooses "hasher". That sounds funny. Do we care? If so, is there a better choice?

3. Hash table equality. From the container requirements, we know that two hash tables x and y are equal if and only if the expression `std::equal(x.begin(), x.end(), y.begin())` returns true. This is not a useful definition for hash tables, so this proposal leaves out operator== altogether. As a consequence, this proposal does not satisfy the container requirements. Do we care?

4. Interface: should we have policy classes (or some other mechanism) to affect (a) whether hash codes are stored; and/or (b) whether the hash table uses forward iterators or bidirectional iterators?

5. Iterator complexity. The container requirements specify that x.begin() is *O(1)*. Implementations can do this, but it's a burden. Is it worth requiring them to do that? (Note that we're implicitly making that requirement just by saying that a hash table is a container.) This is an annoying problem: on the one hand, we don't want to impose a requirement that may be widely ignored. On the other hand, we don't want to do something so drastic as changing the container requirements.

7. Pairs and combining. Should we define a general hash combiner, that takes two hash codes and gives a hash code for the combination? Should we define a default hash function for std::pair<T,U>? (A 'yes' answer to the latter question essentially implies a 'yes' answer to the former.)

7. Default hash function. What, if anything, should the generic hash<T> do? In this proposal it's left as an incomplete type.

8. Bucket interface. Should it be kept as is, or should it be changed to a more container-like interface? (*e.g.* `bucket(n)` might have a return type `const bucket_type&`, where `bucket_type` is an implementation defined container type.

VI. Revision history

Differences from revision 3:

- **Changed the name of the container classes from hash_* to unordered_*, and included an extensive definition of that choice in section III.A. Similarly, changed the header names to <unordered_set> and <unordered_map>, and the name of the concept from Hashed Associative Container to Unordered Associative Container.**
- **Removed the equality operator.**
- **Changed load factor operations to use `float` instead of `double`, and removed the postcondition that appeared to promise that setting the max load factor was something more than just a hint.**

Differences from revision 2:

- **Explicitly said which functions may invalidate iterators and references.**
- **Changed exceptions guarantees: we don't provide the strong guarantee for single-element insert except when the hash function is nothrow.**
- **Increased the number of types that we've defined a default hash function for. It's now defined for all integer types, all pointers (not just char*, wchar_t*, and void*), and all floating-point types.**
- **Removed special treatement for char*, const char*, wchar_t*, const wchar_t*.**

Differences from revision 1:

- **Some changes in naming, to reflect comments at and after the Redmond meeting.**
- **Added wchar_t* and const wchar_t* specializations for std::hash.**
- **Specified 1.0 as the default maximum load factor. (Not in the requirements table, but in the documentation for the predefined hashed associative containers.**
- **Changed requirements table to clarify which operations can be performed on a const hash table**
- **Clarified that copy constructor and copy assignment operator copy the hash function, equality predicate, and max load factor. Deliberately did not answer the question of whether they copy the bucket count; I have left that as an unresolved issue.**
- **Changed rehash complexity. I hope I've gotten it right this time...**
- **Added notes on exception guarantees.**
- **Added list of unresolved issues.**
- **Minor changes in wording, typo correction, etc.**

VII. References

- M. H. Austern, A Proposal to Add Hash Tables to the Standard Library, J16/01-0040 = WG21/N1326, 2001.
- M. H. Austern, "Segmented Iterators and Hierarchical Algorithms", 1998, in M. Jazayeri, R. G. K. Loos, and D. R. Musser, ed., *Generic Programming: International Seminar on Generic Programming, Castle Dagstuhl*, Springer, 2001.
- J. Barreiro, R. Fraley, D. R. Musser, "Hash Tables for the Standard Template Library", X3J16/94-0218 = WG21/N0605, 1995.
- Dinkumware, "Dinkum C++ Library Reference", http://www.dinkumware.com/htm_cpl/index.html.
- Howard Hinnant, "Hashing with Pro 6", http://home.twcny.rr.com/hinnant/tip_archive/MSL%20C++%20Tip%20%2310
- Metrowerks, "Metrowerks CodeWarrior Pro 7 MSL C++ Reference Manual".
- P. J. Plauger, "State of the Art: Hash It", *Embedded Systems Programming*, September, 1998.
- SGI, "Standard Template Library Programmer's Guide", http://www.sgi.com/tech/stl