Document number: **P0083R0**
Date: 2015-09-24
Project: Programming Language C++
Reference: N4527, N3645
Reply to: **Alan Talbot**
cpp@alantalbot.com
Howard Hinnant
howard.hinnant@gmail.com
Jonathan Wakely
cxx@kayari.org
James Dennett
jdennett@google.com

Splicing Maps and Sets (Revision 2)

Related Documents

This proposal addresses the following open issues in LEWG status:

839. Maps and sets missing splice operation

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3518.html#839

1041. Add associative/unordered container functions that allow to extract elements

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3518.html#1041

Changes in Revision 2

This paper updates N3645 as follows:

- Added the **key** accessor function.
- Added a discussion of concerns raised by previous versions.
- Fixed several problems with the proposed wording.
- Improved the organization overall, and improved the narrative in several places.

The Problem

Node-based containers are excellent for creating collections of large or unmovable objects. Maps in particular provide a great way to create database-like tables where objects may be looked up by ID and used in various ways. Since the memory allocations are stable, once you build a map you can take references to its elements and count on them to remain valid as long as the map exists.

The emplace functions were designed precisely to facilitate this pattern by eliminating the need for a copy or a move when creating elements in a map (or any other container). When using a list, map or set, we can construct objects, look them up, use them, and eventually discard them, all without *ever* having to copy or move them (or construct them more than once). This is very useful if the objects are expensive to copy, or have construction/destruction side effects (such as in the classic RAII pattern).

But what happens when we want to take some elements from one table and move them to another? If we were using a list, this would be easy: we would use splice. Splice allows logical manipulation of the list without copying or moving the nodes—only the pointers are changed. But lists are not a good choice to represent tables, and there is no splice for maps.

The key type is const

Yet another problem is that the key type of maps is const. You can’t move out of it at all. This alone was enough of a problem to motivate Issue 1041. We believe that the const key is a basic design flaw in the original map specification which we now have no way to fix because the value type is exposed directly by the API. We feel the solution we are proposing is the best possible given the need to preserve the current container design.

What about move?

Don’t move semantics basically solve all these problems? Unfortunately they don’t. Move is very effective for small collections of objects which are *indirectly* large; that is, which own resources that are expensive to copy. But if the object *itself* is large, or has some limitation on construction (as in the RAII case), then move does not help at all. And “large” in this context may not be very big. A 256 byte object may not seem large until you have several million of them and start comparing the copy times of 256 bytes to the 16 bytes or so of a pointer swap.

But even if the mapped type itself is very small, an **int** for example, the heap allocations and deallocations required to insert a new node and erase an old one are very expensive compared to swapping pointers. When there are large numbers of objects to move around, this overhead can be very significant.

Does anyone care?

Yes! We know of several instances (at CppCon, on Stack Overflow, etc.) where people have asked for functionality that we are proposing and the current Library cannot provide. We believe that real people working on real problems very much need and want this functionality.

History

Talbot's original idea for solving this issue was to add splice-like members to associative containers that took the source container and iterators, and dealt with the splice action under the hood. This would have solved the splice problem, but offered no further advantages.

In Issue 1041, Alisdair Meredith suggested that we need a way to move an element out of a container with a combined move/erase operation. This solves another piece of the problem, but does not help if move is not helpful, and does not address the allocation issue.

Hinnant then suggested that there should be a way to actually remove the node and hold it outside the container. This solves all of the problems, and it is this design that we are proposing. However, although it works fine, it introduces a theoretical problem because it requires casting the const key to a non-const key, which invokes undefined behavior.

Wakely then proposed a refinement that we believe will help make the solution acceptable to the Committee and library vendors.

The Solution

Can you really splice a map?

It turns out that what we need is not actually a splice in the sense of **list::splice**. Because elements must be inserted into their correct positions, a splice-like operation for associative containers must remove the element from the source and insert it into the destination, both of which are non-trivial operations. Although these will have the same *complexity* as a conventional insert and erase, the actual *cost* will typically be much less since the objects do not need to be copied nor the nodes reallocated.

Overview

This design allows splicing operations of all kinds, moving elements (including map keys) out of the container, and a number of other useful operations and designs. It is an enhancement to the associative and unordered associative containers to support the manipulation of nodes. This is a pure addition to the Standard Library.

Extract

The key to the design is a new function **extract** which unlinks the selected node from the container (performing the same balancing actions as **erase**). The **extract** function has the same overloads as the single parameter **erase** function: one that takes an iterator and one that takes a key type. They return an implementation-defined smart pointer type modeled after `unique_ptr` which holds the node while in transit. We will refer to this pointer as the *node pointer* (not to be confused with a raw pointer to the internal node type of the container).

Node Pointer

The node pointer allows iterator-like access to the element (the `value_type`) stored in the node, and non-const access to the key part of the element (the `key_type`) If the node pointer is allowed to destruct while holding the node, the node is properly destructed using the appropriate allocator for the container. The node pointer contains a *copy* of the container's allocator. This is necessary so that the node pointer can outlive the container. (It is interesting to note that the node pointer cannot be an iterator, since an iterator must refer to a particular container.) The container has a typedef for the node pointer type (`node_ptr_type`).

The node pointer type will be independent of the Compare, Hash or Pred template parameters, but will depend on the Allocator parameter. This allows a node to be transferred from `set<T,C1,A>` to `set<T,C2,A>` (for example), but *not* from `set<T,C,A1>` to `set<T,C,A2>`. Even though the allocator types are the same, the container's allocator must also test equal to the node pointer's allocator or the behavior of node pointer **insert** is undefined.

Insert

There is also a new overload of **insert** that takes a node pointer and inserts the node directly, without copying or moving it. For the unique containers, it returns a struct which contains the same information as the `pair<iterator, bool>` returned by the value insert, and also has a `node_ptr` member which is a (typically empty) node pointer which will preserve the node in the event that the insertion fails:

```
struct insert_return {
   iterator position;
   bool inserted;
   node_ptr node;
};
```

(We examined several other possibilities for this return type and decided that this was the best of the available options.) For the multi containers, the node pointer **insert** returns an iterator to the newly inserted node.

Merge

There is also a **merge** operation which takes a non-const reference to the container type and attempts to insert each node in the source container. Merging a container will remove from the source all the elements that can be inserted successfully, and (for containers where the insert may fail) leave the remaining elements in the source. This is very important—none of the operations we propose ever lose elements. (What to do with the leftovers is left up to the user.)

This operation is worth a dedicated function because although it is possible to write fairly efficient client code that does the same thing, it is not quite trivial to do so in the case of the unique containers. (See the *Inserting an entire set* example below for details.) Furthermore, in some cases the merge operation does not need to balance the source container until the merge is complete.

Exception safety

If the container's Compare function is nothrow (which is very common), then removing a node, modifying it, and inserting it is nothrow unless modifying the value throws. And if modifying the value does throw, it does so outside of the containers involved.

If the Compare function does throw, **insert** will not yet have moved its node pointer argument, so the node will still be owned by the argument and will remain available to the caller.

Concerns

Several concerns have been raised about this design. We will address them here.

Undefined behavior

The most difficult part of this proposal from a theoretical perspective is the fact that the extracted element retains its const key type. This prevents moving out of it or changing it. To solve this, we have provided the **key** accessor function, which provides non-const access to the key in the element held by the node pointer. This function requires implementation "magic" to ensure that it works correctly in the presence of compiler optimizations. A straightforward way to do this is with a union of `pair<const key_type, mapped_type>` and `pair<key_type, mapped_type>`.

We do not feel that this poses any technical or philosophical problem. One of the reasons the Standard Library exists is to write non-portable and magical code that the client can’t write in portable C++ (e.g. <atomic>, <typeinfo>, <type_traits>, etc.). This is just another such example. All that is required of compiler vendors to implement this magic is that they not exploit undefined behavior in unions for optimization purposes—and currently compilers already promise this (to the extent that it is being taken advantage of here).

This does impose a restriction on the client that, if these functions are used, std::pair cannot be specialized such that `pair<const key_type, mapped_type>` has a different layout than `pair<key_type, mapped_type>`. We feel the likelihood of anyone actually wanting to do this is effectively zero.

Note that the **key** member function is the only place where such tricks are necessary, and that no changes to the containers or pair are required.

Limitations on implementation

Matt Austern, Chandler Carruth and others have expressed concern that this change limits the implementation options for the associative containers. But these limits already exist. §23.2.4 Associative containers [associative.reqmts] ¶9, and §23.2.5 Unordered associative containers [unord.req] ¶14, effectively require implementations to use node-based designs. So while non-node-based implementations are valid and useful, the Committee has not chosen to standardize such implementations, so we can rely on node-based containers.

Allocator considerations

All allocation is done by the container. The node pointer preserves the allocator type *and* state to ensure that nodes are not exchanged between allocator-incompatible containers, and to ensure that destruction of the element, should the need arise, is done by the correct allocator.

Implementation experience

Hinnant has implemented almost all of this design and feels there is also a great deal of implementation and positive field experience in this area. We believe this is strong evidence that it is implementable and practical.

Examples

Moving elements from one map to another

```
map<int, string> src, dst;
src[1] = “one”;
src[2] = “two”;
src[3] = “trois”;
dst[3] = “three”;

dst.insert(src.extract(src.find(1)));     // Iterator version.
dst.insert(src.extract(2));               // Key type version.
auto r = dst.insert(src.extract(3));      // Key type version.

// src == {}
// dst == {“one”, “two”, “three”}
// r.position == dst.begin() + 2
// r.inserted == false
// r.node == “trois”
```

We have moved elements of `src` into `dst` without any heap allocation or deallocation, and without constructing, destroying or losing any elements. The third insert failed, returning the usual insert return values and the orphaned node.

Inserting an entire set

```
set<int> src{1, 3, 5};
set<int> dst{2, 4, 5};

dst.merge(src);   // Merge src into dst.

// src == {5}
// dst == {1, 2, 3, 4, 5}
```

Here is what you would have to do to get the same functionality with similar efficiency:

```
for (auto i = src.begin(); i != src.end();)
{
   auto p = dst.equal_range(*i);
   if (p.first == p.second)
      dst.insert(p.first, src.extract(i++));
   else
      ++i;
}
```

However, this user code could lose nodes if the comparator throws during insert. The merge operation does not need to do the second comparison and can be made exception-safe.

Surviving the death of the container

The node pointer does not depend on the allocator instance in the container, so it is self-contained and can outlive the container. This makes possible things like very efficient factories for elements:

```
auto new_record()
{
   table_type table;
   table.emplace(...); // Create a record with some parameters.
   return table.extract(table.begin());
}

table.insert(new_record());
```

Moving an object out of a set

Today we can put move-only types into a set using **emplace**, but in general we cannot move them back out. The **extract** function lets us do that:

```
set<move_only_type> s;
s.emplace(...);
move_only_type mot = move(*s.extract(s.begin()));
```

Failing to find an element to remove

What happens if we call the value version of **extract** and the value is not found?

```
set<int> src{1, 3, 5};
set<int> dst;

dst.insert(src.extract(1));
dst.insert(src.extract(2));   // Returns {src.end(), false, node_ptr_type()}.

// src == {3, 5}
// dst == {1}
```

This is well defined. The **extract** failed to find 2 and returned an empty node pointer, which **insert** then trivially failed to insert.

If **extract** is called on a multi container, and there is more than one element that matches the argument, the first matching element is removed.

Proposed Wording

Add a new section to clause 20 [utilities]

20.X Node pointer [associative.nodeptr]

20.X.1 Class node_ptr overview [associative.nodeptr.overview]

1 A *node pointer* is a smart pointer (similar to `unique_ptr`) that accepts ownership of a node from an associative container. It may be used to transfer that ownership to another container of a sufficiently similar type.

2 It is a move-only type associated with the container's `value_type` and `allocator_type`. It is independent of the container's Compare template parameter (for the associative containers) and Hash and Pred template parameters (for the unordered associative containers).

3 Class `node_ptr` is for exposition only. An implementation is permitted to provide equivalent functionality without providing a class with this name.

4 If a client-supplied specialization of `std::pair` causes `pair<const Key, T>` to have a different layout than `pair<Key, T>`, where `Key` is the container's `key_type` and `T` is the container's `mapped_type`, the behavior of operations involving node pointers is undefined.

```
class node_ptr  // Exposition only
{
   typedef unspecified                          container;
public:
   typedef container::value_type                value_type;
   typedef container::key_type                  key_type;
   typedef container::allocator_type            allocator_type;
   typedef value_type&                          reference;
   typedef typename allocator_traits<allocator_type>::pointer pointer;

private:
   unspecified                   container_node_type; // Exposition only
   container_node_type*          ptr_;          // Exposition only
   allocator_type                alloc_;        // Exposition only
public:
   constexpr node_ptr() noexcept;
   constexpr node_ptr(nullptr_t) noexcept
       : node_ptr() { }

   node_ptr(node_ptr&& np) noexcept;
   node_ptr& operator=(node_ptr&& p) noexcept;
   node_ptr& operator=(nullptr_t) noexcept;

   ~node_ptr();

   reference operator*() const;
   pointer operator->() const;

   key_type& key() const noexcept;
   allocator_type get_allocator() const noexcept;
   explicit operator bool() const noexcept;

   void swap(node_ptr&);
};
```

```
void swap(node_ptr& x, node_ptr& y);

bool operator==(const node_ptr& x, nullptr_t) noexcept;

bool operator!=(const node_ptr& x, nullptr_t) noexcept;

bool operator==(nullptr_t, const node_ptr& y) noexcept;

bool operator!=(nullptr_t, const node_ptr& y) noexcept;
```

20.X.2 node_ptr constructors, copy, and assignment [associative.nodeptr.cons]

```
constexpr node_ptr() noexcept;
```

Effects: Constructs a `node_ptr` object that owns nothing.
Postconditions: `static_cast<bool>(*this) == false.`
`get_allocator() == allocator_type().`

```
node_ptr(node_ptr&& np) noexcept;
```

Effects: Constructs a `node_ptr` object initializing `ptr_` with `np.ptr_.`
Move constructs `alloc_` with `np.alloc_.` Sets `np.ptr_` to `nullptr.`

```
node_ptr& operator=(node_ptr&& p) noexcept;
```

Requires: Either
`allocator_traits<allocator_type>::propagate_on_container_move_assignment`
is `true,` or `alloc_ == p.alloc_.`

Effects: If `ptr_ != nullptr`, destroys the `value_type` in the `container_node_ptr` by calling `allocator_traits<allocator_type>::destroy,` deallocates `ptr_` by calling `allocator_traits<allocator_type>::deallocate` and then sets `ptr_` to `nullptr`. Then assigns `p.ptr_` to `ptr_.` If `allocator_traits<allocator_type>::propagate_on_container_move_assignment` is `true`, move assigns `p.alloc_` to `alloc_.`
Assigns `nullptr` to `p.ptr_.`

Returns: `*this.`

```
node_ptr& operator=(nullptr_t) noexcept;
```

Effects: If `ptr_ != nullptr`, destroys the `value_type` in the `container_node_ptr` by calling `allocator_traits<allocator_type>::destroy,` deallocates `ptr_` by calling `allocator_traits<allocator_type>::deallocate and then sets ptr_ to nullptr.`

Returns: `*this.`

20.X.3 node_ptr destructor [associative.nodeptr.dtor]

```
~node_ptr();
```

Effects: If `ptr_ != nullptr`, destroys the `value_type` in the `container_node_ptr` by calling `allocator_traits<allocator_type>::destroy`, deallocates `ptr_` by calling `allocator_traits<allocator_type>::deallocate`.

20.X.4 node_ptr observers [associative.nodeptr.observers]

```
reference operator*() const;
```

Requires: `static_cast<bool>(*this) == true`.
Returns: A reference to the `value_type` in the `container_node_type`.
Throws: Nothing.

```
pointer operator->() const;
```

Requires: `static_cast<bool>(*this) == true`.
Returns: A pointer to the `value_type` in the `container_node_type`.
Throws: Nothing.

```
key_type& key() const noexcept;
```

Requires: `static_cast<bool>(*this) == true`.
Returns: A non-const reference to the `key_type` member of the `value_type` in the `container_node_type`.
Throws: Nothing.

```
allocator_type get_allocator() const noexcept;
```

Returns: `alloc_`.

```
allocator_type get_allocator() const noexcept;
```

Returns: `alloc_`.

```
explicit operator bool() const noexcept;
```

Returns: `ptr_ != nullptr`.

20.X.5 node_ptr modifiers [associative.nodeptr.modifiers]

```
void swap(node_ptr& p);
```

Requires: If `allocator_traits<allocator_type>::propagate_on_container_swap` is `false`, then `alloc_ == p.alloc_`.

Effects: Calls `swap(ptr_, p.ptr_)`. If `allocator_traits<allocator_type>::propagate_on_container_swap` is `true` calls `swap(alloc_, p.alloc_)`.

Throws: Nothing.

20.X.6 node_ptr non-member functions [associative.nodeptr.nonmember]

```
void swap(node_ptr& x, node_ptr& y);
```

Effects: Equivalent to `x.swap(y).`

```
bool operator==(const node_ptr& x, nullptr_t) noexcept;
```

Returns: `!static_cast<bool>(x).`

```
bool operator!=(const node_ptr& x, nullptr_t) noexcept;
```

Returns: `!(x == nullptr).`

```
bool operator==(nullptr_t, const node_ptr& y) noexcept;
```

Returns: `!static_cast<bool>(y).`

```
bool operator!=(nullptr_t, const node_ptr& y) noexcept;
```

Returns: `!(nullptr == y).`

23.2.4 Associative containers [associative.reqmts]

In ¶ 8: change “a denotes a value of X,” to “a and s denote values of X,”.

Add to ¶ 9:

The `extract` members shall invalidate only iterators to the removed elements; references and pointers to the elements remain valid.

Add to table 102:

Expression
`X::node_ptr_type`

Return type
`unspecified` `node_ptr class.`

Note, …
`see 20.X.`

Expression
`X::insert_return_type`

Return type
A `MoveConstructible, MoveAssignable, DefaultConstructible` class type used to describe the results of inserting a `node_ptr`, including at least the following fields:

```
  bool inserted;
  X::iterator position;
  X::node_ptr_type node;
```

Note, …
For an attempt to insert an empty `node_ptr`, `inserted` is false, `position` is `end()`, and `node` is empty. If insertion took place, `inserted` is `true`, `position` points to the inserted element, and `node` is `empty`.

If insertion failed, `inserted` is `false`, `node` owns the node previously owned by `np`, and `position` points to an element with an equivalent key to `*node`.

Expression
`a_uniq.insert(np)`

Return type
`X::insert_return_type`

Note, …
Precondition: `a_uniq.get_allocator() == np.get_allocator().`
Effects: If `np` is empty, has no effect. Otherwise, inserts `*np` if and only if there is no element in the container with key equivalent to the key of `*np.` Postcondition: `np` is empty.

Complexity
Logarithmic

Expression
`a_eq.insert(np)`

Return type
`iterator`

Note, …
Precondition: `a_eq.get_allocator() == np.get_allocator().`
Effects: If `np` is empty, has no effect and returns `a_eq.end()`. Otherwise, inserts `*np` and returns the iterator pointing to the newly inserted element. If a range containing elements equivalent to `*np` exists in `a_eq, *np` is inserted at the end of that range.
Postcondition: `np` is empty.

Complexity
Logarithmic

Expression
`a.insert(p, np)`

Return type
`iterator`

Note, …
Precondition: `a.get_allocator() == np.get_allocator().`
Effects: If `np` is empty, has no effect and returns `a_eq.end()`. Otherwise, inserts `*np` if and only if there is no element with key equivalent to the key of `*np` in containers with unique keys; always inserts `*np` in containers with equivalent keys. Always returns the iterator pointing to the element with key equivalent to the key of `*np.` `*np` is inserted as close as possible to the position just prior to `p`. Postcondition: `np` is empty.

Complexity
Logarithmic in general, but amortized constant if `*np` is inserted right before `p`.

Expression
`a.extract(k)`

Return type
`node_ptr_type`

Note, …
Removes the first element in the container with key equivalent to `k`. Returns a `node_ptr` owning the element if found, otherwise an empty `node_ptr`.

Complexity
`log(a.size())`

Expression
`a.extract(q)`

Return type
`node_ptr_type`

Note, …
Removes the element pointed to by `q`. Returns a `node_ptr` owning the element at `q`.

Complexity
Amortized constant

Expression
`a.merge(s)`

Return type
`void`

Note, …
Precondition: `a.get_allocator() == s.get_allocator().`
Attempts to extract each element in `s` and insert it into `a`. In containers with unique keys, if there is an element in `a` with key equivalent to the key of an element from `s`, then that element is not extracted from `s`. Pointers and references to the moved elements of `s` now refer to those same elements but as members of `a`. Iterators referring to the moved elements will continue to refer to their elements, but they now behave as iterators into `a`, not into `s`.

Complexity
`N log(a.size() + N)` (N has the value `s.size()`)

23.2.5 Unordered associative containers [unord.req]

In ¶ 11: change “a is an object of type X,” to “a and s are objects of type X,”.

Add to ¶ 14:

> The `extract` members shall invalidate only iterators to the removed elements; references and pointers to the elements remain valid.

Add to table 103:

Expression
`X::node_ptr_type`

Return type
`unspecified node_ptr class.`

Note, …
`see 20.X.`

Expression
`X::insert_return_type`

Return type
A MoveConstructible, MoveAssignable, DefaultConstructible class type used to describe the results of inserting a `node_ptr`, including at least the following fields:

```
  bool inserted;
  X::iterator position;
  X::node_ptr_type node;
```

Note, …
For an attempt to insert an empty node, `inserted` is false, `position` is `end()`, and `node` is empty.
If insertion took place, `inserted` is `true`, `position` points to the inserted element, and `node` is empty.
If insertion failed, `inserted` is false, `node` owns the node previously owned by `np`, and `position` points to an element with an equivalent key to `*node`.

Expression
`a_uniq.insert(np)`

Return type
`X::insert_return_type`

Note, …
Precondition: `a_uniq.get_allocator() == np.get_allocator().`
Effects: If `np` is empty, has no effect. Otherwise, inserts `*np` if and only if there is no element in the container with key equivalent to the key of `*np.` Postcondition: `np` is empty.

Complexity
Average case O(1), worst case O(a_uniq.size()).

Expression
`a_eq.insert(np)`

Return type
`X::insert_return_type`

Note, …
Precondition: `a_eq.get_allocator() == np.get_allocator().`
Effects: If `np` is empty, has no effect and returns `a_eq.end()`. Otherwise, inserts `*np` and returns the iterator pointing to the newly inserted element.
Postcondition: `np` is empty.

Complexity
Average case O(1), worst case O(a_eq.size()).

Expression
`a.insert(q, np)`

Return type
`iterator`

Note, …
Precondition: `a.get_allocator() == np.get_allocator().`
Effects: If `np` is empty, has no effect and returns `a_eq.end()`. Otherwise, inserts `*np` if and only if there is no element with key equivalent to the key of `*np` in containers with unique keys; always inserts `*np` in containers with equivalent keys. Always returns the iterator pointing to the element with key equivalent to the key of `*np.` The iterator `q` is a hint pointing to where the search should start. Implementations are permitted to ignore the hint.
Postcondition: `np` is empty.

Complexity
Average case O(1), worst case O(a.size()).

Expression
`a.extract(k)`

Return type
`node_ptr_type`

Note, …
Removes an element in the container with key equivalent to `k`. Returns a `node_ptr` owning the element if found, otherwise an empty `node_ptr`.

Complexity
Average case O(1), worst case O(a.size()).

Expression
`a.extract(q)`

Return type
`node_ptr_type`

Note, …
Removes the element pointed to by `q`. Returns a `node_ptr` owning the element at `q`.

Complexity
Average case O(1), worst case O(a.size()).

Expression
`a.merge(s)`

Return type
`void`

Note, …
Precondition: `a.get_allocator() == s.get_allocator().`
Attempts to extract each element in `s` and insert it into `a`. In containers with unique keys, if there is an element in a with key equivalent to the key of an element from `s`, then that element is not extracted from `s`. Pointers and references to the moved elements of `s` now refer to those same elements but as members of `a`. Iterators referring

to the moved elements and all iterators referring to `a` will be invalidated, but iterators to elements remaining in `s` will remain valid.

Complexity

Average case O(N), where N is s.size(). Worst case O(N * a.size() + N).

23.4.4.1 Class template map overview [map.overview]

Add to `class map`:

```
typedef unspecified node_ptr_type;
typedef unspecified insert_return_type;

node_ptr_type extract(const_iterator position);
node_ptr_type extract(const key_type& x);

insert_return_type      insert(node_ptr_type&& np);
iterator           insert(const_iterator hint, node_ptr_type&& np);
template<class Comp>
void merge(map<Key, T, Comp, Allocator>& source);
template<class Comp>
void merge(map<Key, T, Comp, Allocator>&& source);
```

23.4.5.1 Class template multimap overview [multimap.overview]

Add to `class multimap`:

```
typedef unspecified node_ptr_type;

node_ptr_type extract(const_iterator position);
node_ptr_type extract(const key_type& x);

iterator insert(node_ptr_type&& np);
iterator insert(const_iterator hint, node_ptr_type&& np);
template<class Comp>
void merge(multimap<Key, T, Comp, Allocator>& source);
template<class Comp>
void merge(multimap<Key, T, Comp, Allocator>&& source);
```

23.4.6.1 Class template set overview [set.overview]

Add to `class set`:

```
typedef unspecified node_ptr_type;
typedef unspecified insert_return_type;

node_ptr_type extract(const_iterator position);
node_ptr_type extract(const key_type& x);

insert_return_type      insert(node_ptr_type&& np);
iterator           insert(const_iterator hint, node_ptr_type&& np);
template<class Comp>
void merge(set<Key, Comp, Allocator>& source);
template<class Comp>
void merge(set<Key, Comp, Allocator>&& source);
```

23.4.7.1 Class template multiset overview [multiset.overview]

Add to `class multiset`:

```
typedef unspecified node_ptr_type;

node_ptr_type extract(const_iterator position);
node_ptr_type extract(const key_type& x);

iterator insert(node_ptr_type&& np);
iterator insert(const_iterator hint, node_ptr_type&& np);
template<class Comp>
void merge(multiset<Key, Comp, Allocator>& source);
template<class Comp>
void merge(multiset<Key, Comp, Allocator>&& source);
```

23.5.4.1 Class template unordered_map overview [unord.map.overview]

Add to `class unordered_map`:

```
typedef unspecified node_ptr_type;
typedef unspecified insert_return_type;

node_ptr_type extract(const_iterator position);
node_ptr_type extract(const key_type& x);

insert_return_type      insert(node_ptr_type&& np);
iterator           insert(const_iterator hint, node_ptr_type&& np);
template<class Hsh, class Prd>
void merge(unordered_map<Key, T, Hsh, Prd, Allocator>& source);
template<class Hsh, class Prd>
void merge(unordered_map<Key, T, Hsh, Prd, Allocator>&& source);
```

23.5.5.1 Class template unordered_multimap overview [unord.multimap.overview]

Add to `class unordered_multimap`:

```
typedef unspecified node_ptr_type;

node_ptr_type extract(const_iterator position);
node_ptr_type extract(const key_type& x);

iterator insert(node_ptr_type&& np);
iterator insert(const_iterator hint, node_ptr_type&& np);
template<class Hsh, class Prd>
void merge(unordered_multimap<Key, T, Hsh, Prd, Allocator>& source);
template<class Hsh, class Prd>
void merge(unordered_multimap<Key, T, Hsh, Prd, Allocator>&& source);
```

23.5.6.1 Class template unordered_set overview [unord.set.overview]

Add to `class unordered_set`:

```
typedef unspecified node_ptr_type;
typedef unspecified insert_return_type;

node_ptr_type extract(const_iterator position);
node_ptr_type extract(const key_type& x);

insert_return_type      insert(node_ptr_type&& np);
iterator           insert(const_iterator hint, node_ptr_type&& np);
template<class Hsh, class Prd>
void merge(unordered_set<Key, Hsh, Prd, Allocator>& source);
template<class Hsh, class Prd>
void merge(unordered_set<Key, Hsh, Prd, Allocator>&& source);
```

23.5.7.1 Class template unordered_multiset overview [unord.multiset.overview]

Add to `class unordered_multiset`:

```
typedef unspecified node_ptr_type;

node_ptr_type extract(const_iterator position);
node_ptr_type extract(const key_type& x);

iterator insert(node_ptr_type&& np);
iterator insert(const_iterator hint, node_ptr_type&& np);
template<class Hsh, class Prd>
void merge(unordered_multiset<Key, Hsh, Prd, Allocator>& source);
template<class Hsh, class Prd>
void merge(unordered_multiset<Key, Hsh, Prd, Allocator>&& source);
```

Acknowledgements

Thanks to Alisdair Meredith for long ago pointing out that this problem is more interesting than it first appears, and for Issue 1041.

Thanks to Pablo Halpern, John Lakos, and Alisdair Meredith for reviewing draft materials for Revision 2.

Thanks to Matt Austern, Chandler Carruth and others at the Bristol meeting who encouraged us to spend more time on this to be sure we get it right.