Document number: **P0083R3**
Date: 2016-06-24
Prior Papers: N3586, N3645
Audience: Library Working Group
Reply to: **Alan Talbot** cpp@alantalbot.com
Jonathan Wakely cxx@kayari.org
Howard Hinnant howard.hinnant@gmail.com
James Dennett jdennett@google.com

Splicing Maps and Sets (Revision 5)

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 5 (P0083R3 – this paper)

- Moved default constructor and swap into `node_handle` class definition.
- Map nodes now only have `key` and `mapped`. Set nodes only have `value`.
- Added mention of using `std::launder` to narrative.
- Improved wording and added wording per LWG input.
- Fixed typos and changed variable names and formatting.

Changes in Revision 4 (P0083R2)

- Moved node_handle to section 23.
- Replaced 23.X.1 p2 with new wording and a table depicting transfer-compatible container types.
- Reformulated the invalidation language and improved wording in several places.
- Removed vestigial nullptr_t overloads and all uses of smart-pointer-like access.
- Moved insertion return value behavior text to insert row in tables.
- Fixed various wording typo and formatting errors, and replaced normative uses of “node” with “element”.
- Used std::optional for allocator in node_handle and fixed move semantics.
- Replace noexcept with *throws nothing* in several places. Added *throws nothing* to move assignment operator.
- Added new signatures to support merging of transfer-compatible container types.
- Added Destructible and Swappable requirements to insert_return_type.

Changes in Revision 3 (P0083R1)

- Added the **mapped** and **value** accessor functions and the **empty** state test function.
- Removed the **operator*** and **operator->** accessor functions.
- Added an example of changing the key of a map element.
- Changed the name of the node handle type from **node_ptr** to **node_handle**, removed its characterization as a smart pointer, and stressed that it is move-only.
- Fixed several issues in the formal wording, including adding **noexcept** in several places.
- Changed typedefs to alias declarations.
- Improved wording about invalidation of references and pointers per CWG suggestion.
- Added wording to specify that the container's comparator is used by merge.
- Strengthened the wording of the **pair** specialization restriction.
- Added feature test macro.

Changes in Revision 2 (P0083R0)

- 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).

No splice for old maps

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.

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.

And you can't move the key

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.

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 type which we refer to as the *node handle*. The node handle can be thought of as a special type of container which holds the node while in transit. Note that extracting a node naturally invalidates all iterators to it (since it is no longer an element of the container). Extracting a node from a map of any type invalidates pointers and references to it; this does not occur for sets.

Node Handle

The node handle is a move-only type that holds and provides access to the element (the `value_type`) stored in the node, and provides non-const access to the key part of the element (the `key_type`) and the mapped part of the element (the `mapped_type`). If the node handle is allowed to destruct while holding the node, the node is properly destructed using the appropriate allocator for the container. The node handle contains a *copy* of the container's allocator. This is necessary so that the node handle can outlive the container. The container has a type alias for the node handle type (`node_type`).

The node handle 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 handle's allocator or the behavior of node handle **insert** is undefined.

Insert

There is also a new overload of **insert** that takes a node handle 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 member which is a (typically empty) node handle which will preserve the node in the event that the insertion fails:

```
struct insert_return {
   iterator position;
   bool inserted;
   node_type 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 handle **insert** returns an iterator to the newly inserted node.

Inserting a node into a map of any type invalidates all pointers and references to it; this does not occur for sets.

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.) The insertions are done using the comparator of the destination (the container on which merge is called), as with any other insertion.

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 no-throw (which is very common), then removing a node, modifying it, and inserting it is no-throw 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 handle 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 handle. This function requires implementation "magic" to ensure that it works correctly in the presence of compiler optimizations. One way to do this is with a union of `pair<const key_type, mapped_type>` and `pair<key_type, mapped_type>`. The conversion between these can be effected safely using a technique similar to that used by `std::launder` on extraction and reinsertion.

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, and in the formal wording we restrict any specialization of these pairs.

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 handle 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 {{1,”one”}, {2,”two”}, {3,”buckle my shoe”}};
map<int, string> 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 == “buckle my shoe”
```

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 handle 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()).value());
```

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_type()}.

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

This is well defined. The **extract** failed to find 2 and returned an empty node handle, 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.

Changing the key of a map element

This is a very useful operation that is not possible today without deleting the element and constructing a new one. While doing this with a node handle does require the insertion and tree balancing overhead, it does not cause any memory allocation or deallocation.

```
map<int, string> m{{1,”mango”}, {2,”papaya”}, {3,”guava”}};

auto nh = m.extract(2);
nh.key() = 4;
m.insert(move(nh));

// m == {{1,”mango”}, {3,”guava”}, {4,”papaya”}}
```

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 got it right.

Thanks to Daniel Krügler for reviewing a draft of Revision 4 and pointing out many subtle errors and omissions, and for considerable help with the wording in Revision 5.

Feature Test Macro

The suggested feature test macro for addition to SD-6 is:

```
__cpp_lib_node_extract
```

Proposed Wording

Add a new section to clause 23 [containers]:

23.X Node handle [container.node]

23.X.1 node_handle overview [container.node.overview]

1 A *node handle* is an object that accepts ownership of a single element from an associative container or an unordered associative container. It may be used to transfer that ownership to another container with compatible nodes. Containers with compatible nodes have the same node handle type. Elements may be transferred in either direction between container types in the same row of table X.

Table X – Container types with compatible nodes

`map<K, T, C1, A>`	`map<K, T, C2, A>`
`map<K, T, C1, A>`	`multimap<K, T, C2, A>`
`set<K, C1, A>`	`set<K, C2, A>`
`set<K, C1, A>`	`multiset<K, C2, A>`
`unordered_map<K, T, H1, E1, A>`	`unordered_map<K, T, H2, E2, A>`
`unordered_map<K, T, H1, E1, A>`	`unordered_multimap<K, T, H2, E2, A>`
`unordered_set<K, H1, E1, A>`	`unordered_set<K, H2, E2, A>`
`unordered_set<K, H1, E1, A>`	`unordered_multiset<K, H2, E2, A>`

2 If a node handle is not empty, then it contains an allocator that is equal to the allocator of the container when the element was extracted. If a node handle is empty, it contains no allocator.

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

4 If a user-defined specialization of `std::pair` exists for `pair<const Key, T>` or `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 handles is undefined.

```
template<unspecified>
class node_handle
{
public:
   // These type declarations are described in
[tab:containers.associative.requirements] and [tab:HashRequirements]
   using value_type     = see below;  // Not present for map containers
   using key_type       = see below;  // Not present for set containers
   using mapped_type    = see below;  // Not present for set containers
   using allocator_type = see below;

private:
   using container_node_type = unspecified;
   using ator_traits = allocator_traits<allocator_type>;
   typename ator_traits::rebind_traits<container_node_type>::pointer ptr_;
   optional<allocator_type> alloc_;

public:
   constexpr node_handle() noexcept : ptr_(), alloc_() {}

   ~node_handle();

   node_handle(node_handle&&) noexcept;
   node_handle& operator=(node_handle&&);

   value_type& value() const;       // Not present for map containers
   key_type& key() const;           // Not present for set containers
   mapped_type& mapped() const;     // Not present for set containers

   allocator_type get_allocator() const;
   explicit operator bool() const noexcept;
   bool empty() const noexcept;
   void swap(node_handle&)
   noexcept(ator_traits::propagate_on_container_swap::value ||
            ator_traits::is_always_equal::value);

   friend void swap(node_handle& x, node_handle& y) noexcept(noexcept(x.swap(y)))
   { x.swap(y); }
};
```

23.X.2 node_handle constructors, copy, and assignment [container.node.cons]

`node_handle(node_handle&& nh) noexcept;`

Effects: Constructs a *node_handle* object initializing `ptr_` with `nh.ptr_`. Move constructs `alloc_` with `nh.alloc_`. Assigns `nullptr` to `nh.ptr_` and assigns `nullopt` to `nh.alloc_`.

node_handle& operator=(*node_handle*&& nh);

Requires: Either !alloc_, or ator_traits::propagate_on_container_move_assignment is true, or alloc_ == nh.alloc_.
Effects: If ptr_ != nullptr, destroys the value_type subobject in the container_node_type object pointed to by ptr_ by calling ator_traits::destroy, then deallocates ptr_ by calling ator_traits::rebind_traits<container_node_type>::deallocate. Assigns nh.ptr_ to ptr_. If !alloc_ or ator_traits::propagate_on_container_move_assignment is true, move assigns nh.alloc_ to alloc_. Assigns nullptr to nh.ptr_ and assigns nullopt to nh.alloc_.
Returns: *this.
Throws: Nothing.

23.X.3 node_handle destructor [container.node.dtor]

~*node_handle*();

Effects: If ptr_ != nullptr, destroys the value_type subobject in the container_node_type object pointed to by ptr_ by calling ator_traits::destroy, then deallocates ptr_ by calling ator_traits::rebind_traits<container_node_type>::deallocate.

23.X.4 node_handle observers [container.node.observers]

value_type& value() const;

Requires: empty() == false.
Returns: A reference to the value_type subobject in the container_node_type object pointed to by ptr_.
Throws: Nothing.

key_type& key() const;

Requires: empty() == false.
Returns: A non-const reference to the key_type member of the value_type subobject in the container_node_type object pointed to by ptr_.
Throws: Nothing.
Remarks: Modifying the key through the returned reference is permitted.

mapped_type& mapped() const;

Requires: empty() == false.
Returns: A reference to the mapped_type member of the value_type subobject in the container_node_type object pointed to by ptr_.
Throws: Nothing.

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

Requires: `empty() == false`.
Returns: `*alloc_`.
Throws: Nothing.

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

Returns: `ptr_ != nullptr`.

```
bool empty() const noexcept;
```

Returns: `ptr_ == nullptr`.

23.X.5 node_handle modifiers [container.node.modifiers]

```
void swap(node_handle& nh)
   noexcept(ator_traits::propagate_on_container_swap::value ||
   ator_traits::is_always_equal::value);
```

Requires: `!alloc_`, or `!nh.alloc_`, or `ator_traits::propagate_on_container_swap` is `true`, or `alloc_ == nh.alloc_`.

Effects: Calls `swap(ptr_, nh.ptr_)`. If `!alloc_`, or `!nh.alloc_`, or `ator_traits::propagate_on_container_swap` is `true` calls `swap(alloc_, nh.alloc_)`.

23.2.4 Associative containers [associative.reqmts]

In ¶ 8: Change "`a` denotes a value of type `X`," to "`a` denotes a value of type `X`, `a2` denotes a value of a type with nodes compatible with type `X` (Table X)". Add "`nh` denotes a non-const rvalue of type `X::node_type`".

Add to ¶ 9:

> The `extract` members invalidate only iterators to the removed element; pointers and references to the removed element remain valid. However, accessing the element through such pointers and references while the element is owned by a `node_type` is undefined behavior. References and pointers to an element obtained while it is owned by a `node_type` are invalidated if the element is successfully inserted.

Add to **Table 109 — Associative container requirements (in addition to container)**:

Expression
`X::node_type`

Return type
a specialization of a *`node_handle`* class template, such that the public nested types are the same types as the corresponding types in `X`.

Assertion/note/pre-/post-condition

see [container.node].

Complexity
compile time

Expression
`X::insert_return_type`

Return type
A class type used to describe the results of inserting a `node_type` that includes at least the following non-static public data members:

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

The type shall be `MoveConstructible`, `MoveAssignable`, `DefaultConstructible`, `Destructible`, and lvalues of that type shall be swappable ([swappable.requirements]).

Assertion/note/pre-/post-condition

Complexity
compile time

Expression
`a_uniq.insert(nh)`

Return type
`X::insert_return_type`

Assertion/note/pre-/post-condition
Precondition: `nh` is empty or `a_uniq.get_allocator() == nh.get_allocator()`
Effects: If `nh` is empty, has no effect. Otherwise, inserts the element owned by `nh` if and only if there is no element in the container with a key equivalent to `nh.key()`.
Postcondition: If `nh` is empty, `inserted` is `false`, `position` is `end()`, and `node` is empty. Otherwise if the insertion took place, `inserted` is `true`, `position` points to the inserted element, and `node` is empty; if the insertion failed, `inserted` is `false`, `node` has the previous value of `nh`, and `position` points to an element with a key equivalent to `nh.key()`.

Complexity
logarithmic

Expression
`a_eq.insert(nh)`

Return type
`iterator`

Assertion/note/pre-/post-condition
Precondition: `nh` is empty or `a_eq.get_allocator() == nh.get_allocator()`.
Effects: If `nh` is empty, has no effect and returns `a_eq.end()`. Otherwise, inserts the element owned by `nh` and returns an iterator pointing to the newly inserted element. If a range containing elements with keys equivalent to `nh.key()` exists in `a_eq`, the element is inserted at the end of that range.
Postcondition: `nh` is empty.

Complexity
logarithmic

Expression
`a.insert(p, nh)`

Return type
`iterator`

Assertion/note/pre-/post-condition
Precondition: `nh` is empty or `a.get_allocator() == nh.get_allocator()`.
Effects: If `nh` is empty, has no effect and returns `a.end()`. Otherwise, inserts the element owned by `nh` if and only if there is no element with key equivalent to `nh.key()` in containers with unique keys; always inserts the element owned by `nh` in containers with equivalent keys. Always returns the iterator pointing to the element with key equivalent to `nh.key()`. The element is inserted as close as possible to the position just prior to `p`.
Postcondition: `nh` is empty if insertion succeeds, unchanged if insertion fails.

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

Expression
`a.extract(k)`

Return type
`node_type`

Assertion/note/pre-/post-condition
Removes the first element in the container with key equivalent to `k`. Returns a `node_type` owning the element if found, otherwise an empty `node_type`.

Complexity
`log(a.size())`

Expression
`a.extract(q)`

Return type
`node_type`

Assertion/note/pre-/post-condition
Removes the element pointed to by `q`. Returns a `node_type` owning that element.

Complexity
Amortized constant

Expression
`a.merge(a2)`

Return type
`void`

Assertion/note/pre-/post-condition
Precondition: `a.get_allocator() == a2.get_allocator()`.
Attempts to extract each element in `a2` and insert it into `a` using the comparison object of `a`. In containers with unique keys, if there is an element in `a` with key equivalent to the key of an element from `a2`, then that element is not extracted from `a2`.
Postcondition: Pointers and references to the transferred elements of `a2` refer to those same elements but as members of `a`. Iterators referring to the transferred elements will continue to refer to their elements, but they now behave as iterators into `a`, not into `a2`.
Throws: Nothing unless the comparison object throws.

Complexity
`N log(a.size() + N)` (N has the value `a2.size()`)

23.2.5 Unordered associative containers [unord.req]

In ¶ 11: Change "`a` is an object of type `X`," to "`a` is an object of type `X`, `a2` is an object with nodes compatible with type X (Table X)". Add "`nh` denotes a non-const rvalue of type `X::node_type`".

Add a new paragraph after ¶ 15:

> The `extract` members invalidate only iterators to the removed element, and preserve the relative order of the elements that are not erased; pointers and references to the removed element remain valid. However, accessing the element through such pointers and references while the element is owned by a `node_type` is undefined behavior. References and pointers to an element obtained while it is owned by a `node_type` are invalidated if the element is successfully inserted.

Add to **Table 110 — Unordered associative container requirements (in addition to container)**:

Expression
`X::node_type`

Return type
a specialization of a *`node_handle`* class template, such that the public nested types are the same types as the corresponding types in `X`.

Assertion/note/pre-/post-condition

see [container.node].

Complexity
compile time

Expression
`X::insert_return_type`

Return type
A class type used to describe the results of inserting a `node_type` that includes at least the following non-static public data members:

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

The type shall be `MoveConstructible`, `MoveAssignable`, `DefaultConstructible`, `Destructible`, and lvalues of that type shall be swappable ([swappable.requirements]).

Assertion/note/pre-/post-condition

Complexity
compile time

Expression
`a_uniq.insert(nh)`

Return type
`X::insert_return_type`

Assertion/note/pre-/post-condition
Precondition: `nh` is empty or `a_uniq.get_allocator() == nh.get_allocator()`
Effects: If `nh` is empty, has no effect. Otherwise, inserts the element owned by `nh` if and only if there is no element in the container with a key equivalent to `nh.key()`.
Postcondition: If `nh` is empty, `inserted` is `false`, `position` is `end()`, and `node` is empty. Otherwise if the insertion took place, `inserted` is `true`, `position` points to the inserted element, and `node` is empty; if the insertion failed, `inserted` is `false`, `node` has the previous value of `nh`, and `position` points to an element with a key equivalent to `nh.key()`.

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

Expression
`a_eq.insert(nh)`

Return type
`iterator`

Assertion/note/pre-/post-condition
Precondition: `nh` is empty or `a_eq.get_allocator() == nh.get_allocator()`.
Effects: If `nh` is empty, has no effect and returns `a_eq.end()`. Otherwise, inserts the element owned by `nh` and returns an iterator pointing to the newly inserted element.
Postcondition: `nh` is empty.

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

Expression
`a.insert(q, nh)`

Return type
`iterator`

Assertion/note/pre-/post-condition
Precondition: `nh` is empty or `a.get_allocator() == nh.get_allocator()`.
Effects: If `nh` is empty, has no effect and returns `a.end()`. Otherwise, inserts the element owned by `nh` if and only if there is no element with key equivalent to `nh.key()` in containers with unique keys; always inserts the element owned by `nh` in containers with equivalent keys. Always returns the iterator pointing to the element with key equivalent to `nh.key()`. The iterator `q` is a hint pointing to where the search should start. Implementations are permitted to ignore the hint.
Postcondition: `nh` is empty is insertion succeeds, unchanged if insertion fails.

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

Expression
`a.extract(k)`

Return type
`node_type`

Assertion/note/pre-/post-condition
Removes an element in the container with key equivalent to `k`. Returns a `node_type` owning the element if found, otherwise an empty `node_type`.

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

Expression
`a.extract(q)`

Return type
`node_type`

Assertion/note/pre-/post-condition
Removes the element pointed to by `q`. Returns a `node_type` owning that element.

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

Expression
`a.merge(a2)`

Return type
`void`

Assertion/note/pre-/post-condition
Precondition: `a.get_allocator() == a2.get_allocator()`.
Attempts to extract each element in `a2` and insert it into `a` using the hash function and key equality predicate of `a`. In containers with unique keys, if there is an element in `a` with key equivalent to the key of an element from `a2`, then that element is not extracted from `a2`.
Postcondition: Pointers and references to the transferred elements of `a2` refer to those same elements but as members of `a`. Iterators referring to the transferred elements and all iterators referring to `a` will be invalidated, but iterators to elements remaining in `a2` will remain valid.
Throws: Nothing unless the hash function or key equality predicate throws.

Complexity
Average case O(N), where N is `a2.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_type;
typedef unspecified insert_return_type;

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

insert_return_type insert(node_type&& nh);
iterator           insert(const_iterator hint, node_type&& nh);

template<class C2>
  void merge(map<Key, T, C2, Allocator>& source);
template<class C2>
  void merge(map<Key, T, C2, Allocator>&& source);
template<class C2>
  void merge(multimap<Key, T, C2, Allocator>& source);
template<class C2>
  void merge(multimap<Key, T, C2, Allocator>&& source);
```

23.4.5.1 Class template multimap overview [multimap.overview]

Add to `class multimap`:

```
typedef unspecified node_type;

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

iterator insert(node_type&& nh);
iterator insert(const_iterator hint, node_type&& nh);

template<class C2>
  void merge(multimap<Key, T, C2, Allocator>& source);
template<class C2>
  void merge(multimap<Key, T, C2, Allocator>&& source);
template<class C2>
  void merge(map<Key, T, C2, Allocator>& source);
template<class C2>
  void merge(map<Key, T, C2, Allocator>&& source);
```

23.4.6.1 Class template set overview [set.overview]

Add to `class set`:

```
typedef unspecified node_type;
typedef unspecified insert_return_type;

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

insert_return_type insert(node_type&& nh);
iterator           insert(const_iterator hint, node_type&& nh);

template<class C2>
  void merge(set<Key, C2, Allocator>& source);
template<class C2>
  void merge(set<Key, C2, Allocator>&& source);
template<class C2>
  void merge(multiset<Key, C2, Allocator>& source);
template<class C2>
  void merge(multiset<Key, C2, Allocator>&& source);
```

23.4.7.1 Class template multiset overview [multiset.overview]

Add to `class multiset`:

```
typedef unspecified node_type;

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

iterator insert(node_type&& nh);
iterator insert(const_iterator hint, node_type&& nh);

template<class C2>
  void merge(multiset<Key, C2, Allocator>& source);
template<class C2>
  void merge(multiset<Key, C2, Allocator>&& source);
template<class C2>
  void merge(set<Key, C2, Allocator>& source);
template<class C2>
  void merge(set<Key, C2, Allocator>&& source);
```

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

Add to `class unordered_map`:

```
typedef unspecified node_type;
typedef unspecified insert_return_type;

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

insert_return_type insert(node_type&& nh);
iterator           insert(const_iterator hint, node_type&& nh);

template<class H2, class P2>
  void merge(unordered_map<Key, T, H2, P2, Allocator>& source);
template<class H2, class P2>
  void merge(unordered_map<Key, T, H2, P2, Allocator>&& source);
template<class H2, class P2>
  void merge(unordered_multimap<Key, T, H2, P2, Allocator>& source);
template<class H2, class P2>
  void merge(unordered_multimap<Key, T, H2, P2, Allocator>&& source);
```

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

Add to `class unordered_multimap`:

```
typedef unspecified node_type;

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

iterator insert(node_type&& nh);
iterator insert(const_iterator hint, node_type&& nh);

template<class H2, class P2>
  void merge(unordered_multimap<Key, T, H2, P2, Allocator>& source);
template<class H2, class P2>
  void merge(unordered_multimap<Key, T, H2, P2, Allocator>&& source);
template<class H2, class P2>
  void merge(unordered_map<Key, T, H2, P2, Allocator>& source);
template<class H2, class P2>
  void merge(unordered_map<Key, T, H2, P2, Allocator>&& source);
```

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

Add to `class unordered_set`:

```
typedef unspecified node_type;
typedef unspecified insert_return_type;

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

insert_return_type insert(node_type&& nh);
iterator           insert(const_iterator hint, node_type&& nh);

template<class H2, class P2>
  void merge(unordered_set<Key, H2, P2, Allocator>& source);
template<class H2, class P2>
  void merge(unordered_set<Key, H2, P2, Allocator>&& source);
template<class H2, class P2>
  void merge(unordered_multiset<Key, H2, P2, Allocator>& source);
template<class H2, class P2>
  void merge(unordered_multiset<Key, H2, P2, Allocator>&& source);
```

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

Add to `class unordered_multiset`:

```
typedef unspecified node_type;

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

iterator insert(node_type&& nh);
iterator insert(const_iterator hint, node_type&& nh);

template<class H2, class P2>
  void merge(unordered_multiset<Key, H2, P2, Allocator>& source);
template<class H2, class P2>
  void merge(unordered_multiset<Key, H2, P2, Allocator>&& source);
template<class H2, class P2>
  void merge(unordered_set<Key, H2, P2, Allocator>& source);
template<class H2, class P2>
  void merge(unordered_set<Key, H2, P2, Allocator>&& source);
```