Document number: **N3586**
Date: 2013-03-17
Project: Programming Language C++
Reference: N3485
Reply to: **Alan Talbot**
cpp@alantalbot.com
Howard Hinnant
howard.hinnant@gmail.com

Splicing Maps and Sets

Related Documents

This proposal addresses the following NAD Future issues:

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

Motivation

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 change the ID of an entry in our table? Or 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? Actually not. 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 enormous.

Yet another problem is that the key type of maps is (necessarily) const. You can't move out of it at all. This alone was enough of a problem to motivate Issue 1041.

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 time will typically be much less since the objects do not need to be copied nor the nodes reallocated.

What is the solution?

Alan'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 have 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.

Howard then suggested that there should be a way to actually remove the node and hold it outside the container. It is this design that we are proposing.

Summary

This is an enhancement to the associative and unordered associative containers to support the manipulation of nodes. It is a pure addition to the Library.

The key to the design is a new function **remove** which unlinks the selected node from the container (performing the same balancing actions as **erase**). The **remove** 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).

The node pointer allows pointer-like non-const access to the element (the `value_type`) stored in the node. (It can be dereferenced just like an iterator.) 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`).

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 derives from `pair<iterator, bool>` and has a `node_ptr` member which is a (typically empty) node pointer which will preserve the node in the event that the insertion fails. The pair is the same as the pair returned by the value type **insert**. (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.

For convenience we offer another overload of **insert** that takes a non-const reference to the container type and attempts to insert the entire container. This can be thought of as a merge operation on the two containers. Inserting 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 design allows splicing operations of all kinds, repositioning of elements in the container, moving elements (including map keys) out of the container, and a number of other useful operations and designs.

Examples

Moving elements from one map to another

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

dst.insert(src.remove(src.find(1)));      // Iterator version.
dst.insert(src.remove(2));                // Key type version.
```

We have moved the contents of `src` into `dst` without any heap allocation or deallocation, and without constructing or destroying any objects.

Inserting an entire set

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

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

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

This operation is worth a dedicated function because although it is possible to write equally efficient client code, it is not quite trivial to do so in the case of the unique containers. Here is what you 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.remove(i++));
   else
      ++i;
}
```

Changing the key of an element

Returning to our example of the database-like table, suppose we want to change the ID of an element? This is very easy:

```
struct record { ... };
typedef map<int, record> table_type;
table_type table;
table.emplace(38, ...);

auto elem = table.remove(38);
elem->first = 97;
table.insert(move(elem));
```

We have changed the key of our record from 38 to 97 without actually touching the `record` object at all.

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:

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

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

Moving an object out of a container

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

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

Failing to find an element to remove

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

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

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

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

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

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

Details

The return type of insert

The unique containers return `pair<iterator, bool>` from the value type **insert**. The node pointer **insert** will return a struct derived from this pair:

```
struct insert_return_type : public pair<iterator, bool> {
   node_ptr_type node_ptr;
};
```

This makes it easy to use the node pointer **insert** in the same way as the value type **insert** (including in generic contexts), while also providing a node pointer to hold the node if the insertion fails.

The node pointer allocator

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 if the allocator types are the same, the container's allocator must test equal to the node pointer's allocator or the behavior of node pointer **insert** is undefined.

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 be available to the caller.

Proposed Wording

This is a summary of the wording. A future version of this paper will contain complete wording.

Add to the unique associative containers:

```
typedef implementation-defined node_ptr_type;

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

struct insert_return_type : public std::pair<iterator, bool> {
   node_ptr_type node_ptr;
};
insert_return_type insert(node_ptr_type&& np);
iterator           insert(const_iterator hint, node_ptr_type&& np);
template<...> void insert(container<...>& source);
```

Add to the multi associative containers:

```
typedef implementation-defined node_ptr_type;

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

iterator           insert(node_ptr_type&& np);
iterator           insert(const_iterator hint, node_ptr_type&& np);
template<...> void insert(container<...>& 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.