The following is an excerpt from Scott Meyers' new book, *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*.

Item 44: Prefer member functions to algorithms with the same names.

Some containers have member functions with the same names as STL algorithms. The associative containers offer count, find, lower_bound, upper_bound, and equal_range, while list offers remove, remove_if, unique, sort, merge, and reverse. Most of the time, you'll want to use the member functions instead of the algorithms. There are two reasons for this. First, the member functions are faster. Second, they integrate better with the containers (especially the associative containers) than do the algorithms. That's because algorithms and member functions that share the same name typically do *not* do exactly the same thing.

We'll begin with an examination of the associative containers. Suppose you have a set<int> holding a million values and you'd like to find the first occurrence of the value 727, if there is one. Here are the two most obvious ways to perform the search:

```
set<int> s;                                          // create set, put
...                                                  // 1,000,000 values
                                                     // into it

set<int>::iterator i = s.find(727);                  // use find member
if (i != s.end()) ...                                // function

set<int>::iterator i = find(s.begin(), s.end(), 727);  // use find algorithm
if (i != s.end()) ...
```

The find member function runs in logarithmic time, so, regardless of whether 727 is in the set, set::find will perform no more than about 40 comparisons looking for it, and usually it will require only about 20. In contrast, the find algorithm runs in linear time, so it will have to perform 1,000,000 comparisons if 727 isn't in the set. Even if 727 is in the set, the find algorithm will perform, on average, 500,000 comparisons to locate it. The efficiency score is thus

Member find:	About 40 (worst case) to about 20 (average case)
Algorithm find:	1,000,000 (worst case) to 500,000 (average case)

As in golf, the low score wins, and as you can see, this matchup is not much of a contest.

I have to be a little cagey about the number of comparison required by member find, because it's partially dependent on the implementation used by the associative containers. Most implementations use red-black trees, a form of balanced tree that may be out of balance by up to a factor of two. In such implementations, the maximum number of comparisons needed to search a set of a million values is 38, but for the vast majority of searches, no more than 22 comparisons is required. An implementation based on perfectly balanced trees would never require more than 21 comparisons, but in practice, the overall performance of such perfectly balanced trees is inferior to that of red-black trees. That's why most STL implementations use red-black trees.

Efficiency isn't the only difference between member and algorithm find. As Item 19 explains, STL algorithms determine whether two objects have the same value by checking for equality, while associative containers use equivalence as their "sameness" test. Hence, the find algorithm searches for 727 using equality, while the find member function searches using equivalence. The difference between equality and equivalence can be the difference between a successful search and an unsuccessful search. For example, Item 19 shows how using the find algorithm to look for something in an associative container could fail even when the corresponding search using the find member function would succeed! You should therefore prefer the member form of find, count, lower_bound, etc., over their algorithm eponyms when you work with associative containers, because they offer behavior that is consistent with the other member functions of those containers. Due to the difference between equality and equivalence, algorithms don't offer such consistent behavior.

This difference is especially pronounced when working with maps and multimaps, because these containers hold pair objects, yet their member functions look only at the key part of each pair. Hence, the count member function counts only pairs with matching keys (a "match," naturally, is determined by testing for equivalence); the value part of each pair is ignored. The member functions find, lower_bound, upper_bound, and equal_range behave similarly. If you use the count algorithm, however, it will look for matches based on (a) equality and (b) both components of the pair; find, lower_bound, etc., do the same thing. To get the algorithms to look at only the key part of a pair, you have to jump through the hoops described in Item 23 (which would also allow you to replace equality testing with equivalence testing).

On the other hand, if you are really concerned with efficiency, you may decide that Item 23's gymnastics, in conjunction with the loga-

rithmic-time lookup algorithms of Item 34, are a small price to pay for an increase in performance. Then again, if you're *really* concerned with efficiency, you'll want to consider the non-standard hashed containers described in Item 25, though there you'll again confront the difference between equality and equivalence.

For the standard associative containers, then, choosing member functions over algorithms with the same names offers several benefits. First, you get logarithmic-time instead of linear-time performance. Second, you determine whether two values are "the same" using equivalence, which is the natural definition for associative containers. Third, when working with maps and multimaps, you automatically deal only with key values instead of with (key, value) pairs. This triumvirate makes the case for preferring member functions pretty iron-clad.

Let us therefore move on to list member functions that have the same names as STL algorithms. Here the story is almost completely about efficiency. Each of the algorithms that list specializes (remove, remove_if, unique, sort, merge, and reverse) copies objects, but list-specific versions copy nothing; they simply manipulate the pointers connecting list nodes. The algorithmic complexity of the algorithms and the member functions is the same, but, under the assumption that manipulating pointers is less expensive than copying objects, list's versions of these functions should offer better performance.

It's important to bear in mind that the list member functions often behave differently from their algorithm counterparts. As Item 32 explains, calls to the algorithms remove, remove_if, and unique must be followed by calls to erase if you really want to eliminate objects from a container, but list's remove, remove_if, and unique member functions honestly get rid of elements; no subsequent call to erase is necessary.

A significant difference between the sort algorithm and list's sort function is that the former can't be applied to lists. Being only bidirectional iterators, list's iterators can't be passed to sort. A gulf also exists between the behavior of the merge algorithm and list's merge. The algorithm isn't permitted to modify its source ranges, but list::merge *always* modifies the lists it works on.

So there you have it. When faced with a choice between an STL algorithm or a container member function with the same name, you should prefer the member function. It's almost certain to be more efficient, and it's likely to be better integrated with the container's usual behavior, too.