Distinguishing STL Search Algorithms

This article is based on material in Scott's new book, Effective STL *[1].*

So you want to look for something, and you have a container or you have iterators demarcating a range where you think it's located. How do you conduct the search? Your quiver is fairly bursting with arrows: count, count_if, find, find_if, binary_search, lower_bound, upper_bound, and equal_range. Decisions, decisions! How do you choose?

Easy. You reach for something that's fast and simple. The faster and simpler, the better.

For the time being, I'll assume that you have a pair of iterators specifying a range to be searched. Later, I'll consider the case where you have a container instead of a range.

In selecting a search strategy, much depends on whether your iterators define a sorted range. If they do, you can get speedy (usually logarithmic-time) lookups via binary_search, lower_bound, upper_bound, and equal_range. If the iterators don't demarcate a sorted range, you're limited to the linear-time algorithms count, count_if, find, and find_if. In what follows, I'll ignore the _if variants of count and find, just as I'll ignore the variants of binary_search, lower_ and upper_bound, and equal_range taking a predicate. Whether you rely on the default search predicate or you specify your own, the considerations for choosing a search algorithm are the same.

If you have an unsorted range, your choices are count or find. They answer slightly different questions, so it's worth taking a closer look at them. count answers the question, "Is the value there, and if so, how many copies are there?" while find answers the question, "Is it there, and if so, where is it?"

Suppose all you want to know is whether some special Widget value w is in a list. Using count, the code looks like this:

```
list<Widget> lw;                         // list of Widgets
Widget w;                                // special Widget value
...
```

```
if (count(lw.begin(), lw.end(), w)) {
    ...                                         // w is in lw
} else {
    ...                                         // it's not
}
```

This demonstrates a common idiom: using count as an existence test. count returns either zero or a positive number, so we rely on the conversion of nonzero values to true and zero to false. It would arguably be clearer to be more explicit about what we are doing,

```
if (count(lw.begin(), lw.end(), w) != 0) ...
```

and some programmers write it that way, but it's common to rely on the implicit conversion, as in the original example.

Compared to that original code, using find is slightly more complicated, because you have to test find's return value against the list's end iterator:

```
if (find(lw.begin(), lw.end(), w) != lw.end()) {
    ...                                         // found it
} else {
    ...                                         // didn't find it
}
```

For existence testing, the idiomatic use of count is slightly simpler to code. At the same time, it's also less efficient when the search is successful, because find stops once it's found a match, while count must continue to the end of the range looking for additional matches. For most programmers, find's edge in efficiency is enough to justify the slight increase in usage complexity.

Often, knowing whether a value is in a range isn't enough. Instead, you'll want to know the first object in the range with the value. For example, you might want to print the object, you might want to insert something in front of it, or you might want to erase it. When you need to know not just whether a value exists but also which object (or objects) has that value, you need find:

```
list<Widget>::iterator i = find(lw.begin(), lw.end(), w);
if (i != lw.end()) {
    ...                                         // found it, i points to the first one
} else {
    ...                                         // didn't find it
}
```

For sorted ranges, you have other choices, and you'll definitely want to use them. count and find run in linear time, but the search algorithms for sorted ranges (binary_search, lower_bound, upper_bound, and equal_range) run in logarithmic time.

The shift from unsorted ranges to sorted ranges leads to another shift: from using equality to determine whether two values are the same to using equivalence [2]. That's because the count and find algorithms both search using equality, while binary_search, lower_bound, upper_bound, and equal_range employ equivalence.

To test for the existence of a value in a sorted range, use binary_search. Unlike bsearch in the standard C library (and hence also in the standard C++ library), binary_search returns only a bool: whether the value was found. binary_search answers the question, "Is it there?," and its answer is either yes or no. If you need more information than that, you need a different algorithm.

Here's an example of binary_search applied to a sorted vector:

```
vector<Widget> vw;                              // create vector, put
...                                             // data into it, sort the
sort(vw.begin(), vw.end());                     // data

Widget w;                                       // value to search for
...

if (binary_search(vw.begin(), vw.end(), w)) {
  ...                                           // w is in vw
} else {
  ...                                           // it's not
}
```

If you have a sorted range and your question is, "Is it there, and if so, where is it?" you want equal_range, but you may think you want lower_bound. I'll discuss equal_range shortly, but first, let's examine lower_bound as a way of locating values in a range.

When you ask lower_bound to look for a value, it returns an iterator pointing to either the first copy of that value (if it's found) or to the proper insertion location for that value (if it's not). lower_bound thus answers the question, "Is it there? If so, where is the first copy, and if it's not, where would it go?" As with find, the result of lower_bound must be tested to see if it's pointing to the value you're looking for. Unlike find, you can't just test lower_bound's return value against the end iterator. Instead, you must test the object lower_bound identifies to see if that's the value you want.

Many programers use lower_bound like this:

```
vector<Widget>::iterator i = lower_bound(vw.begin(), vw.end(), w);

if (i != vw.end() && *i == w) {           // make sure i points to an object;
                                          // make sure the object has the
                                          // correct value; this has a bug!
```

```
    ...                                    // found the value, i points to the
                                           // first object with that value
  } else {
    ...                                    // not found
  }
```

This works most of the time, but it's not really correct. Look again at the test to determine whether the desired value was found:

```
  if (i != vw.end() && *i == w) ...
```

This is an *equality* test, but lower_bound searched using *equivalence*. Most of the time, tests for equivalence and equality yield the same results, but as note [2] demonstrates, it's not that hard to come up with situations where equality and equivalence are different. In such situations the code above is wrong.

To do things properly, you must check to see if the iterator returned from lower_bound points to an object with a value that is equivalent to the one you searched for. You could do that manually, but it can get tricky, because you have to be sure to use the same comparison function that lower_bound used. In general, that could be an arbitrary function (or function object). If you passed a comparison function to lower_bound, you'd have to be sure to use the same comparison function in your hand-coded equivalence test. *That* would mean that if you changed the comparison function you passed to lower_bound, you'd have to make the corresponding change in your check for equivalence. Keeping the comparison functions in sync isn't rocket science, but it is another thing to remember, and I suspect you already have plenty you're expected to keep in mind.

There is an easier way: use equal_range. equal_range returns a *pair* of iterators, the first equal to the iterator lower_bound would return, the second equal to the one upper_bound would return (i.e., the one-past-the-end iterator for the range of values equivalent to the one searched for). equal_range, then, returns a pair of iterators that demarcate the range of values equivalent to the one you searched for. A well-named algorithm, no? (equivalent_range would be better, of course, but equal_range is still pretty good.)

There are two important observations about equal_range's return value. First, if the two iterators are the same, that means the range of objects is empty; the value wasn't found. That observation is the key to using equal_range to answer the question, "Is it there?" You use it like this:

```
  vector<Widget> vw;
  ...
  sort(vw.begin(), vw.end());

  typedef vector<Widget>::iterator VWIter;    // convenience typedefs
  typedef pair<VWIter, VWIter> VWIterPair;
```

```
VWIterPair p = equal_range(vw.begin(), vw.end(), w);
if (p.first != p.second) {                // if equal_range didn't return
                                          // an empty range...
   ...                                    // found it, p.first points to the
                                          // first one and p.second
                                          // points to one past the last
} else {
   ...                                    // not found, both p.first and
                                          // p.second point to the
}                                         // insertion location for
                                          // the value searched for
```

This code uses only equivalence, so it is always correct.

The second thing to note about equal_range's return value is that the distance between its iterators is equal to the number of objects in the range, i.e., the objects with a value equivalent to the one that was searched for. As a result, equal_range not only does the job of find for sorted ranges, it also replaces count. For example, to locate the Widgets in vw with a value equivalent to w and then print out how many such Widgets exist, you could do this:

```
VWIterPair p = equal_range(vw.begin(), vw.end(), w);
cout << "There are " << distance(p.first, p.second)
     << " elements in vw equivalent to w.";
```

So far, our discussion has assumed we want to search for a value in a range, but sometimes we're more interested in finding a *location* in a range. For example, suppose we have a Timestamp class and a vector of Timestamps that's sorted so that older timestamps come first:

```
class Timestamp { ... };
bool operator<(const Timestamp& lhs,        // returns whether lhs
               const Timestamp& rhs);       // precedes rhs in time
vector<Timestamp> vt;                       // create vector, fill it with
...                                         // data, sort it so that older
sort(vt.begin(), vt.end());                 // times precede newer ones
```

Now suppose we have a special timestamp, ageLimit, and we want to remove from vt all the timestamps that are older than ageLimit. In this case, we don't want to search vt for a Timestamp equivalent to ageLimit, because there might not be any elements with that exact value. Instead, we need to find a *location* in vt: the first element that is no older than ageLimit. This is as easy as easy can be, because lower_bound will give us precisely what we need:

```
Timestamp ageLimit;
...
```

```
vt.erase(vt.begin(), lower_bound(vt.begin(),          // eliminate from vt all
                                 vt.end(),            // objects that precede
                                 ageLimit));          // ageLimit's value
```

If our requirements change slightly so that we want to eliminate all the timestamps that are at least as old as ageLimit, we need to find the location of the first timestamp that is *younger* than ageLimit. That's a job tailor-made for upper_bound:

```
vt.erase(vt.begin(), upper_bound(vt.begin(),          // eliminate from vt all
                                 vt.end(),            // objects that precede
                                 ageLimit));          // or are equivalent
                                                      // to ageLimit's value
```

upper_bound is also useful if you want to insert things into a sorted range so that objects with equivalent values are stored in the order in which they were inserted. For example, we might have a sorted list of Person objects, where the objects are sorted by name:

```
class Person {
public:
  ...
  const string& name() const;
  ...
};

struct PersonNameLess:
  public binary_function<Person, Person, bool> {

  bool operator()(const Person& lhs, const Person& rhs) const
  {
    return lhs.name() < rhs.name();
  }

};

list<Person> lp;
...

lp.sort(PersonNameLess());                    // sort lp using
                                              // PersonNameLess
```

To keep the list sorted the way we desire (by name, with equivalent names stored in the order in which they are inserted), we can use upper_bound to specify the insertion location:

```
Person newPerson;

...

lp.insert( upper_bound(lp.begin(),             // insert newPerson after
                       lp.end(),               // the last object in lp
                       newPerson,              // that precedes or is
                       PersonNameLess()),      // equivalent to
         newPerson);                           // newPerson
```

This works fine and is quite convenient, but it's important not to be misled by this use of upper_bound into thinking that we're magically looking up an insertion location in a list in logarithmic time. We're not. Because we're working with a list, the lookup takes linear time, but it performs only a logarithmic number of comparisons.

Up to this point, I have considered only the case where you have a pair of iterators defining a range to be searched. Often you have a container, not a range. In that case, you must distinguish between the sequence and associative containers. For the standard sequence containers (vector, string, deque, and list), you follow the advice I've outlined in this Item, using the containers' begin and end iterators to demarcate the range.

The situation is different for the standard associative containers (set, multiset, map, and multimap), because they offer member functions for searching that are generally better choices than the STL algorithms [3]. Fortunately, the member functions usually have the same names as the corresponding algorithms, so where the foregoing discussion recommends you choose algorithms named count, find, equal_range, lower_bound, or upper_bound, you simply select the same-named member functions when searching associative containers. binary_search calls for a different strategy, because there is no member function analogue to this algorithm. To test for the existence of a value in a set or map, use count in its idiomatic role as a test for membership:

```
set<Widget> s;                          // create set, put data into it
...
Widget w;                               // w still holds the value to search for
...
if (s.count(w)) {
   ...                                  // a value equivalent to w exists
} else {
   ...                                  // no such value exists
}
```

To test for the existence of a value in a multiset or multimap, find is generally superior to count, because find can stop once it's found a single object with the desired value, while count, in the worst case, must examine every object in the container.

However, count's role for counting things in associative containers is secure. In particular, it's a better choice than calling equal_range and applying distance to the resulting iterators. For one thing, it's clearer: count means "count." For another, it's easier; there's no need to create a pair and pass its components to distance. For a third, it's probably a little faster.

Given everything we've considered in this Item, where do we stand? The following table says it all.

What You Want to Know	Algorithm to Use		Member Function to Use	
	On an Unsorted Range	On a Sorted Range	With a set or map	With a multiset or multimap
Does the desired value exist?	find	binary_search	count	find
Does the desired value exist? If so, where is the first object with that value?	find	equal_range	find	find or lower_bound (see article)
Where is the first object with a value not preceding the desired value?	find_if	lower_bound	lower_bound	lower_bound
Where is the first object with a value succeeding the desired value?	find_if	upper_bound	upper_bound	upper_bound
How many objects have the desired value?	count	equal_range	count	count
Where are all the objects with the desired value?	find (iteratively)	equal_range	equal_range	equal_range

In the column summarizing how to work with sorted ranges, the frequency with which equal_range occurs may be surprising. That frequency arises from the importance of testing for equivalence when searching. With lower_bound and upper_bound, it's too easy to fall back on equality tests, but with equal_range, testing only for equivalence is the natural thing to do. In the second row for sorted ranges, equal_range beats out find for an additional reason: equal_range runs in logarithmic time, while find takes linear time.

For multisets and multimaps, the table lists both find and lower_bound as candidates when you're looking for the first object with a particular value. find is the usual choice for this job, and you may have noticed that it's the one listed in the table for sets and maps. For the multi containers, however, find is not guaranteed to identify the *first* element in the container with a given value if more than one is present; its charter is only to identify *one* of those elements. If you really need to find the *first* object with a given value, you'll want to employ lower_bound, and you'll have to manually perform the second half of the equivalence test. (You could avoid the manual equivalence test by using equal_range, but calling equal_range is more expensive than calling lower_bound.)

Selecting among count, find, binary_search, lower_bound, upper_bound, and equal_range is easy. Choose the algorithm or member function that offers you the behavior and performance you need and that requires the least amount of work when you call it. Follow that advice (or consult the table), and you should never get confused.

Notes and References

[1] Scott Meyers, *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*, Addison-Wesley, 2001, ISBN 0-201-74962-9. This article is based on Item 45 in *Effective STL*.

[2] Two objects are *equivalent* if neither precedes the other in some sort order of interest. Often, equivalent values are equal, but not always. For example, the strings "STL" and "stl" are equivalent in a case-insensitive sort, but they are certainly not equal. For details on the distinction between equivalence and equality, consult any good reference on the STL. In *Effective STL*, the issue is examined in Item 19.

[3] You'll find a justification for this claim in Item 44 of *Effective STL*. That Item is available on-line at http://www.awl.com/cseng/titles/0-201-74962-9/item44-2.pdf.

About the Author

Scott Meyers is one of the world's foremost authorities on C++; *Effective STL* is his third C++ book. He has a Ph.D. in Computer Science from Brown University, sits on the technical advisory boards of several companies, and provides training and consulting services to clients worldwide. His web site is http://www.aristeia.com/.