Why you shouldn't use set (and what you should use instead)
Matt Austern

Everything in the standard C++ library is there for a reason, but it isn't always obvious what that reason is. The standard isn't a tutorial; it doesn't distinguish between basic, everyday components and ones that are there only for rare and specialized purposes.

One example is the Associative Container `std::set` (and its siblings `map`, `multiset`, and `multimap`). Sometimes it does make sense to use a `set`, but not as often as you might think. The standard library provides other tools for storing and looking up data, and often you can do just as well with a simpler, smaller, faster data structure.

What is a set?

A `set` is an STL container that stores values and permits easy lookup. For example, you might have a set of strings:

```
std::set<std::string> S;
```

You can add a new element by writing

```
S.insert("foo";.
```

A set may not contain more than one element with the same key, so `insert` won't add anything if `S` already contains the string `"foo"`; instead it just looks up the old element. The return value includes a status code indicating whether or not the new element got inserted.

As with all STL containers, you can step through all of the elements in a `set` using the set's *iterators*. `S.begin()` returns an iterator that points to `S`'s first element, and `S.end()` returns an iterator that points immediately after `S`'s last element. The elements of a `set` are always sorted in ascending order-which is why `insert` doesn't have an argument telling it where the new element is supposed to be put. The new element automatically gets inserted in the correct location.

What it means for `set` to be an Associative Container is that you can look up a value, by writing

```
i = S.find("foo");.
```

If the lookup is successful then `find` returns an iterator that points to the appropriate element in the `set`. If there is no such element then `find` returns the iterator `S.end()`, which doesn't point to anything.

What if you need the elements to be arranged in something other than ascending order, or if you're creating a set of elements for which "less than" isn't defined? In that case you can use `set`'s second template parameter, a *function object* that determines which of two objects is supposed to come before the other. By default it is `std::less<T>`, meaning that the smaller element comes first. You can create a set of strings stored in reverse order, though, by using a different function object:

```
std::set<std::string, std::greater<std::string> > S,
```

or you can create a set of strings in case-insensitive order by writing your own function object[1]:

```
struct less_nocase {
  static bool compare_chars(char x, char y) {
    return std::toupper(x) < std::toupper(y);
  }
  bool operator()(const string& x, const string& y) const {
    return std::lexicographical_compare(x.begin(), x.end(),
                                        y.begin(), y.end(),
                                              Compare_chars);
 }
```

[1]Warning: Case-insensitive string comparison is more complicated than it looks, and this code is subtly wrong. In a future column I'll explain what's wrong with it, and how to rewrite it correctly.

```
};

std::set<std::string, less_nocase> S;
```

Even more generally you can store some complicated data structure, and use only a piece of it for comparisons:

```
struct Client {
  unsigned long id;
  string last_name, first_name;
  ...
};

struct id_compare {
  bool operator()(const Client& x, const Client& y) const {
    return x.id < y.id;
 }
};

std::set<Client, id_compare> clients;
```

That's not all there is to `set`, of course. It has all the usual member functions of an STL container, and some special member functions of its own. The basic purpose, though, is managing a collection of elements that can be looked up by key. (The Associative Container `map` is very similar, except that in a `set` the elements are their own keys while in a `map` the elements are `pair`s; the first member of each `pair` is the key, and the second is some value that's associated with the key.)

What's wrong?

Nothing is wrong, precisely-just incomplete. Contrary to appearances, I still haven't given any good reason why you would ever want to use a `set`. You don't need any special data structures or member functions to insert or look up elements, after all. If you've got any STL container C, you can find an element in it with the generic `find` algorithm:

```
i = std::find(C.begin(), C.end(), "foo");
```

Just like with the `set::find` member function, this will return an iterator that points to the element you're looking for, or `C.end()` if there is no such element. There's really only one important difference: `std::find` performs a linear search, so the time it takes is proportional to *N* (where *N* is the number of elements you're looking through), while the time taken by set::find is proportional to *log N*. That is, if *N* is large, `set::find` is much faster.

The real point of `set`, and the other Associative Containers in the standard C++ library, is that they make specific performance guarantees: It's guaranteed that the time to find an element is proportional to *log N*, where *N* is the number of elements in the container, and also that the time to insert an element is proportional to *log N*. There are also specific guarantees about exception safety and iterator invalidation; those guarantees, too, are sometimes important.

If you don't need all of those properties, though, you'll be paying for something you don't use. A data structure that supports fast insertion and fast lookup is necessarily complicated; `set` is typically implemented as a *red-black tree*, which has substantial space and time overhead. Each element in a red-black tree requires more than three extra words of storage (a color marker, and pointers to two children and a parent). Insertions require tree rebalancing, and lookup and iteration require pointer chasing.

If you really need insertion and lookup in guaranteed *log N* time, then `set` is a sensible choice. If you don't, though, then you have other options. Suppose, for example, you're willing to relax the guarantee of insertion in *log N* time.

That's not so unlikely; chances are you'll need to look things up much more often than you'll need to insert them.

Red-black trees aren't the only way to organize data that permits lookup in logarithmic time. One of the basic algorithms of computer science is *binary search*, which works by successively dividing a range in half. Binary search is *log N* and it doesn't require any fancy data structures, just a sorted collection of elements. Binary search is part of the STL-not as a container, but as the generic algorithms `lower_bound`, `upper_bound`, `equal_range`, and `binary_search`. In practice, `lower_bound` tends to be most useful. If `[first, last)` is a range of iterators and if the elements in the range are sorted in ascending, then

```
std::lower_bound(first, last, x);
```

returns an iterator pointing to an element that's equivalent to x (if such an element exists), or, if there is no such element, an iterator pointing to the position where the element would have been. You can use whatever data structure is convenient, so long as it provides STL iterator; usually it's easiest to use a C array, or a `vector`.

Both `std::lower_bound` and `set::find` take time proportional to *log N*, but the constants of proportionality are very different. Using g++ on a 450 MHz Pentium III it takes 0.9 seconds to perform a million lookups in a sorted `vector<double>` of a million elements, and almost twice as long, 1.71 seconds, using a `set`. Moreover, the `set` uses almost three times as much memory (48 million bytes) as the `vector` (16.8 million).

If you need to insert new elements, of course, you have to insert them in the proper position-`lower_bound` will fail badly if it is given a range that isn't sorted. That position is exactly what `lower_bound` returns:

```
template <class Vector, class T>
void insert_into_vector(Vector& v, const T& t) {
  typename Vector::iterator i
    = std::lower_bound(v.begin(), v.end(), t);
  if (i == v.end() || t < *i)
    V.insert(i, t);
}
```

This helper function checks, before inserting `t`, to make sure that it isn't already there. If you wish to allow duplicate elements, as `multiset` does, then you should omit the check.

Using a sorted vector instead of a set gives you faster lookup and much faster iteration, but at the cost of slower insertion. Insertion into a `set`, using `set::insert`, is proportional to *log N*, but insertion into a sorted vector, using `insert_into_vector`, is proportional to *N*. Whenever you insert something into a `vector`, `vector::insert` has to make room by shifting all of the elements that follow it. On average, if you're equally likely to insert a new element anywhere, you'll be shifting *N/2* elements.

There are two special cases where you won't pay such a severe penalty. First, if the elements you insert always go at the end-that is, if you're inserting values in order-then `vector::insert` won't ever have to shift any elements. (And "nearly in order" is nearly as good as "in order"!) Second, you may be able to fill the entire sorted collection before you look anything up in it; a spell check program, for example, starts out with a large dictionary and only occasionally adds new words. The vector doesn't have to be sorted until you need to look something up in it, so you can quickly build an unordered collection, inserting elements with `vector::push_back` instead of `insert_into_vector`, and then sort the vector all at once with

```
std::sort(v.begin(), v.end());.
```

It may sometimes be convenient to bundle all of this together into a small container adaptor. In the interest of space I'll omit the "boilerplate" that's needed to support the complete STL container interface, but the essential parts are simple.

```
template <class T, class Compare = std::less<T> >
struct sorted_vector {
  using std::vector;
  using std::lower_bound;
  vector<T> V;
  Compare cmp;
```

```
  typedef typename vector<T>::iterator       iterator;
  typedef typename vector<T>::const_iterator const_iterator;
  iterator       begin()       { return V.begin(); }
  iterator       end()         { return V.end(); }
  const_iterator begin() const { return V.begin(); }
  const_iterator end()   const { return V.end(); }

  ...

  sorted_vector(const Compare& c = Compare())
    : V(), cmp(c) {}

  template <class InputIterator>
  sorted_vector(InputIterator first, InputIterator last,
                Const Compare& c = Compare())
    : V(first, last), cmp(c)
  {
    std::sort(begin(), end(), cmp);
  }

  ...

  iterator insert(const T& t) {
    iterator i = lower_bound(begin(), end(), t, cmp);
    if (i == end() || cmp(t, *i))
      V.insert(i, t);
    return i;
  }

  const_iterator find(const T& t) const {
    const_iterator i = lower_bound(begin(), end(), t, cmp);
    return i == end() || cmp(t, *i) ? end() : i;
  }
};
```

This class does not satisfy the requirements of a standard Associative Container, since the complexity of `insert` is *O(N)* rather than *O(log N)*, but otherwise it is almost a drop-in replacement for `set`.

What is `set` good for?

The point isn't that `set` is useless-there are times when it's the right choice. We can finally write down the conditions when it's a better choice than `sorted_vecto`r or the equivalent:

- The collection can potentially grow so large that the difference between *O(N*) and *O(log N)* is important.

- The number of lookups is the same order of magnitude as the number of insertions; there aren't so few insertions that insertion speed is irrelevant.

- Elements are inserted in random order, rather than being inserted in order.

- Insertions and lookups are interleaved; we don't have distinct insertion and lookup phases.

Sometimes all four of these conditions are true. If they are, you should use `set`; that's what it's designed for. If any of them are false, though, using such a complicated data structure would be a waste and you could get better performance by using a simple sorted vector.

Every component in the standard C++ library is there because it's useful for some purpose, but sometimes that purpose is narrowly defined and rare. As a general rule you should always use the simplest data structure that meets your needs. The more complicated a data structure, the more likely that it's not as widely useful as it might seem.