Algorithms and Containers

Matt Austern

Here are three code samples that illustrate some of the most common mistakes people make when using C++ library containers, iterators, and algorithms. Do you see what's wrong with each one?

Fragment 1

```
// Fill two vectors with random numbers
std::vector<int> v1;
std::vector<int> v2;
std::generate_n(std::back_inserter(v1), 5, std::rand);
std::generate_n(std::back_inserter(v2), 5, std::rand);

// Sort the vectors
std::sort(v1.begin(), v2.end());
std::sort(v2.begin(), v2.end());

// Print the vectors
std::copy(v1.begin(), v1.end(), std::ostream_iterator<int>(cout, "\n"));
std::copy(v2.begin(), v2.end(), std::ostream_iterator<int>(cout, "\n"));
```

Fragment 2

```
// Create a vector of integers
int A[5] = { 1, 2, -3, -4, 5 };
std::vector<int> v(A, A + 5);

// Remove negative numbers from the vector
std::remove_if(v.begin(), v.end(),
               std::bind2nd(std::less<int>(), 0));

// Print a sequence of non-negative numbers
std::copy(v.begin(), v.end(), std::ostream_iterator<int>(cout, "\n"));
```

Fragment 3

```
// Create a set
int A[5] = { 1, 5, 2, 3, 4};
std::set<int> s(A, A + 5);

// Remove an element from the set
std::remove(s.begin(), s.end(), 2);

// Print the set
std::copy(s.begin(), s.end(), std::ostream_iterator<int>(cout, "\n"));
```

These fragments are wrong for different reasons. Fragment 1 is merely a careless typographical error: writing `sort(v1.begin(), v2.end())` instead of `sort(v1.begin(), v1.end())`. It's an error because `sort`'s arguments `first` and `last` are supposed to be a valid range, and `v1.begin()` and `v2.end()` do not form a valid range. They're iterators from two different containers, and if you try to step from `v1.begin()` to `v2.end()` you'll end up stepping through memory you don't own. Calling a standard algorithm with iterators from two different containers is an easy mistake; it will probably cause your program to crash or to go into an infinite loop. If your program is crashing and you're looking over your code to find the cause, this should be one of the first things you look for.

Fragment 2 is more interesting; it's a conceptual mistake, not just a typo. The third argument to `remove_if` is a function object that returns `true` when it's passed a negative number, so you might expect that

```
  remove_if(v.begin(), v.end(), bind2nd(std::less<int>(), 0))
```

would remove all of the negative elements from `v`. That's not what happens. On my system, the contents of `v` after the call to `remove_if` are:

```
   1 2 5 -4 5.
```

The `vector` still has five elements, just like it did before.

The problem is a conceptual mismatch between the goal (performing an operation on a container) and the means (performing an operation on a range of values). There's no way for `remove_if` to change the number of elements in a `vector`, because it never sees a `vector`. All it sees is a range of iterators that point to values. It can examine those values, it can copy a value from one location to another, but it can't change the number of locations themselves. If you think about this the right way, it's even obvious: pointers into a C array are iterators, and you can't very well change the number of elements in an C array.

What it means to "remove" elements from a range `[first, last)` using `remove_if` is that all of the elements you want to keep get moved to the beginning of the range; they're contained in the range `[first, new_last)`, where `new_last` is `remove_if`'s return value. Sometimes (again, for example, if you're using a C array) all you need to do is keep track of that value `new_last`. If you're trying to physically remove elements from a `vector`, though, you need to do one more step: erasing elements using one of `vector`'s member functions. The complete operation is:

```
  v.erase(remove_if(v.begin(), v.end(), bind2nd(less<int>(), 0)),
          v.end())
```

Finally, the third fragment is incorrect in the way that it combines the standard algorithm `remove` and the standard container `set`. As we saw above, `remove` and `remove_if` "remove" elements by copying values from one place in a range to another; they just come down to a series of assignments `*i1 = *i2`, where `i1` and `i2` are iterators in the original range. You can't do that to a `set`, though. The elements in a `set` are always sorted in ascending order, so you can't just arbitrarily assign a new value to a `set`'s element-you'd be breaking the ordering invariant. Depending on your implementation, you may find that it fails to compile or you may find that it compiles but that it crashes when you try to run it.

The correct way to remove all copies of the number 2 from a `set<int>` is to use one of `set`'s member functions:

```
  s.erase(2).
```

The correct equivalent of `remove_if` for a `set` is more complicated.

Algorithms on containers

It may appear that I'm just presenting a grab bag of common mistakes, but there's more to it than that. These three mistakes have something in common: they all involve the difference between applying an algorithm to a container and applying an algorithm to the range of elements in a container. If you're familiar with the C++ Standard Library, you're used to
the idea that the natural idiom for finding something in a container is

```
  i = std::find(C.begin(), C.end(), x).
```

It's important, though, to remember what the individual pieces of this expression mean. The first two parameters aren't just a funny syntax for passing a container to `find`; `find` never sees a container, just iterators that (in this case) happen to point to the beginning and end of a container. We could equally well have written

```
  i = std::find(previous, C.end(), x)
```

where `previous` isn't the beginning of the container, but the end point of an earlier search.

Generic algorithms like `find` are possible because iterators have a well-defined interface: you can use `find` with any iterator type that satisfies the Input Iterator requirements described in Table 72 of the C++ Standard. Containers, though, also have a well-defined interface: the container requirements are Table 65 of the C++ Standard. There are no container-based algorithms in the C++ Standard Library, but the container requirements make it possible to write such algorithms.

In some cases, such as `sort`, a container-based algorithm can be nothing more than a thin wrapper around an existing algorithm from the Standard Library:

```
  template <class Container>
  inline void sort(Container& C)
  {
    std::sort(C.begin(), C.end());
  }
```

In other cases, such as `find`, the situation is slightly more complicated: `find` doesn't modify any elements, and it can perfectly well be applied to a `const` container. What kind of iterator should a container-based version of `find` return? The only sensible answer is that, since containers' `begin` and `end` member functions are overloaded on const, the container-based version of find should be overloaded on const as well:

```
  template <class Container, class T>
  inline typename Container::iterator
  find(Container& C, const T& value)
  {
    return std::find(C.begin(), C.end(), value);
  }

  template <class Container, class T>
  inline typename Container::const_iterator
  find(const Container& C, const T& value)
  {
    return std::find(C.begin(), C.end(), value);
  }
```

These sorts of wrappers are already useful: it's common to apply an algorithm to an entire container, and if you write

```
 sort(v);
```

it's impossible for you to make the first of the three mistakes I showed at the beginning of this column. The more interesting container-based algorithms, though, are the ones that use the container interface to do things that containers on ranges can't.

A generic algorithm that operates on a range of iterators `[first, last)` can't change the number of elements in the range, but a generic algorithm that operates on a container suffers no such limitation. The second code fragment at the beginning of this column was broken because it was incomplete. To remove elements from a container like `vector` or `list`, you can instead use this version of `remove` or `remove_if`:

```
  template <class Sequence, class T>
  inline void remove(Sequence& S, const T& x) {
    S.erase(std::remove(S.begin(), S.end(), x), S.end());
  }

  template <class Sequence, class Predicate>
  inline void remove_if(Sequence& S, Predicate p) {
    S.erase(std::remove_if(S.begin(), S.end(), p), S.end());
  }
```

As before, these are thin wrappers around existing Standard Library algorithms.

Container traits

This still isn't quite right: our `remove` and `remove_if` algorithms aren't as general as they ought to be. Naturally, it doesn't make sense to talk about removing elements from any arbitrary container type; it only makes sense for containers that allow you to delete elements. As defined, however, these algorithms are even more restrictive than that.
The C++ Standard defines two categories of variable-size containers: *sequences*, like `vector` and `list`, and *associative containers*, like `set` and `map`. Our versions of `remove` and `remove_if` are, as the name of the first template parameter suggests, appropriate for sequences; the `erase` member function is part of the sequence requirements. These algorithms are not, however, appropriate for associative containers. We've already seen the problem: `std::remove` and `std::remove_if` work by assigning one element to another, and that's not something you can do with the elements of an associative container.

Removing elements from an associative container is well defined, but we have to go about it differently. Writing `remove` for associative containers is easy, since the associative container requirements already have a member function that does just the same thing:

```
  template <class AssociativeContainer, class T>
  inline void remove(AssociativeContainer& C, const T& x) {
    C.erase(x);
  }
```

Writing `remove_if` is slightly trickier, but only slightly: we use linear search to find the elements we want to remove, and then erase them one at a time.

```
  template <class AssociativeContainer, class Predicate>
  void remove_if(AssociativeContainer& C, Predicate p)
  {
    typedef typename AssociativeContainer::iterator iterator;
    iterator cur = C.begin();
    const iterator last = C.end();
    while ((cur = std::find_if(cur, last, p)) != last) {
      iterator tmp = cur++;
      C.erase(tmp);
    }
  }
```

(Strictly speaking, this isn't completely general: it assumes that erasing an element from an associative container doesn't invalidate iterators that point to any other elements. That property happens to be true for the standard associative containers `set`, `map`, `multiset`, and `multimap`, and also for the nonstandard but widely available containers `hash_set`, `hash_map`, `hash_multiset`, and `hash_multimap`, but it isn't guaranteed to be true for all associative containers. I don't know of a good way to write `remove_if` for an associative container where erasing an element might invalidate all other iterators. Here's one possibility:

```
  template <class AssociativeContainer, class Predicate>
  void remove_if(AssociativeContainer& C, Predicate p)
  {
    typename AssociativeContainer::iterator i;
    while ((i = std::find_if(C.begin(), C.end(), p)) != C.end())
      C.erase(i);
  }
```

It's terribly slow, since it goes back to the beginning of the container every time.)

Whether or not we want to rely on iterator noninvalidation, we can write a version of `remove_if` for sequences and a different version for associative containers. What are we to do? We can't very well overload them, since an expression like

```
  remove_if(C, p)
```

would be ambiguous; there's nothing to tell the compiler which version you mean. We could distinguish the two versions by name (`remove_if_seq` and `remove_if_assoc`, perhaps), or we could leave out support for associative containers altogether, but neither of those options is satisfactory either.

Fortunately, this is a well-known problem and there's a well-known solution. Some of the algorithms in the C++ Standard Library already present exactly the same difficulty. Consider, for example, the generic algorithm `advance`, which takes an iterator `i` and a number n, and increments `i` n times. There's an obvious implementation when `i` is a Forward Iterator (a loop with `++i` in its body), and there's an equally obvious implementation when `i` is a Random Access Iterator (`i += n`). Again this is vaguely like overloading, but not in the sense that the C++ overload resolution mechanism can work with. Overloading in C++ works with types, not with abstract requirements like "Forward Iterator" and "Random Access Iterator".

The solution in the Standard Library is to represent these abstract descriptions within the C++ type system, thus allowing us to use overload resolution. First we define a set of tag classes, `input_iterator_tag`, `output_iterator_tag`, `forward_iterator_tag`, `bidirectional_iterator_tag`, and `random_access_iterator_tag`, and then we define a mechanism so that every iterator type is associated with one of those tag classes. For example, `int*` and `std::vector<bool>::const_iterator` are associated with the tag class `random_access_iterator_tag`, while `std::list<std::string>` is associated with the tag class `bidirectional_iterator_tag`.

As usual, the easiest way to define such a mapping between types is with a traits class. The standard traits class for iterators is called `std::iterator_traits`. For any iterator type `Iter`, we can find the appropriate tag class by writing

```
  typename std::iterator_traits<Iter>::iterator_category.
```

Finally, then, we write several versions of advance overloaded on an iterator tag argument, and write a wrapper that dispatches to one of them using `iterator_traits`. The dispatch takes place at compile time, so there is no performance penalty.

The C++ standard provides all of the infrastructure for compile-time dispatching on iterator categories, and several standard algorithms use this mechanism. The standard doesn't provide such an infrastructure for dispatching on container categories, but we can easily provide it for ourselves:

```
  struct container_tag { };
  struct sequence_tag { };
  struct associative_container_tag { };

  template <class Container> struct container_traits {
    typedef container_tag container_category;
  };

  template <class T, class Allocator>
  struct container_traits<std::vector<T, Allocator> > {
    typedef sequence_tag container_category;
  };

  template <class T, class Allocator>
  struct container_traits<std::list<T, Allocator> > {
    typedef sequence_tag container_category;
  };

  template <class T, class Allocator>
  struct container_traits<std::set<T, Allocator> > {
    typedef associative_container_tag container_category;
  };

  // ...
```

The "..." is because we have to specialize `container_traits` for every container type we want to work with. If we fail to specialize it for some container type, then that type will be tagged as a general container, not as a sequence or an associative container; a generic algorithm operating on that type won't be able to use any member functions from the sequence or associative container requirements.

Using `container_traits`, we can define `remove_if` properly both for sequences and for associative containers:

```
  template <class AssociativeContainer, class Predicate>
  void remove_if(AssociativeContainer& C, Predicate pred,
                 associative_container_tag)
  {
    typedef typename AssociativeContainer::iterator iterator;
    iterator cur = C.begin();
    const iterator last = C.end();
    while ((cur = std::find_if(cur, last, pred)) != last) {
      iterator tmp = cur++;
      C.erase(tmp);
    }
  }

  template <class Sequence, class Predicate>
  inline void remove_if(Sequence& S, Predicate p, sequence_tag)
  {
    S.erase(std::remove_if(S.begin(), S.end(), p), S.end());
  }

  template <class Container, class Predicate>
  inline void remove_if(Container& C, Predicate p)
  {
    typedef typename container_traits<C>::container_category cat;
    remove_if(C, p, cat());
  }
```

Conclusion

This isn't finished work. Traits are a important, general mechanism. The C++ Standard Library includes `iterator_traits` so that we can write generic algorithms that depend on the properties of specific iterators, and perhaps a future version of the Standard Library will include something like `container_traits` so that we can more easily write generic algorithms that operate on containers. Is the minimal version of `container_traits` we've presented here sufficient, or should it be something more extensive? Might it be, for example, that the kind of algorithm one writes for linked lists (algorithms based on splicing list nodes) is fundamentally different from the kind of algorithm one writes for vectors, and that this difference should be reflected in `container_traits`? Should `container_traits` provide information about iterator invalidation guarantees, and about exception safety?

At present the C++ community has very little experience with container-based generic
algorithms, so we can't do much more than speculate. Container-based generic algorithms are surely useful, though, and some version of `container_traits` will surely be an important tool for writing them.