This is a pre-publication draft of the article I wrote for the June 2001 issue of *C/C++ Users Journal*. “Pre-publication” means this is what I sent to *CUJ*, but it may not be exactly the same as what appeared in print, because *CUJ* and I typically make small changes during preparation of the final version of the article.

Three Guidelines for Effective Iterator Usage

At first glance, STL iterators appear straightforward. Look more closely, however, and you’ll notice that containers actually offer four different iterator types: iterator, const_iterator, reverse_iterator, and const_reverse_iterator. From there it’s only a matter of time until you note that of these four types, only one is accepted by containers in certain forms of insert and erase. That’s when the questions begin. Why four different types? What is the relationship among them? Are they interconvertible? Can the different types be mixed in calls to algorithms and STL utility functions? How do these types relate to containers and their member functions?

This article, drawn from my new book, *Effective STL*, answers these questions through three specific guidelines that will help you get the most out of STL iterators.

Guideline 1: Prefer iterator to const_iterator, reverse_iterator, and const_reverse_iterator.

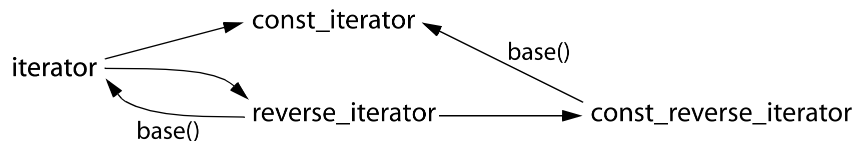
As you know, every standard container offers four types of iterator. For a container<T>, the type iterator acts like a T*, while const_iterator acts like a const T* (which you may also see written as a T const *; they mean the same thing [2]). Incrementing an iterator or a const_iterator moves you to the next element in the container in a traversal starting at the front of the container and proceeding to the back. reverse_iterator and const_reverse_iterator also act like T* and const T*, respectively, except that incrementing these iterator types moves you to the next element in the container in a traversal from back to front.

Let me show you two things. First, take a look at some signatures for insert and erase in vector<T>:

```
iterator insert(iterator position, const T& x);
iterator erase(iterator position);
iterator erase(iterator rangeBegin, iterator rangeEnd);
```

Every standard container offers functions analogous to these, though the return types vary, depending on the container type. The thing to notice is that these functions demand parameters of type iterator. Not const_iterator, not reverse_iterator, not const_reverse_iterator. Always iterator. Though containers support four iterator types, one of those types has privileges the others do not have. That type is iterator. iterator is special [3].

The second thing I want to show you is this diagram, which displays the conversions that exist among iterator types.



The diagram shows that there are implicit conversions from iterator to const_iterator, from iterator to reverse_iterator, and from reverse_iterator to const_reverse_iterator. It also shows that a reverse_iterator may be converted into an iterator by using the reverse_iterator's base member function, and a const_reverse_iterator may similarly be converted into a const_iterator via base. The diagram does not show that the iterators obtained from base may not be the ones you want. We'll explore that issue in Guideline 3.

Observe that there is no way to get from a const_iterator to an iterator or from a const_reverse_iterator to a reverse_iterator. This is important, because it means that if you have a const_iterator or a const_reverse_iterator, you'll find it difficult to use those iterators with some container member functions. Such functions demand iterators, and since there's no conversion path from the const iterator types back to iterator, the const iterator types are largely useless if you want to use them to specify insertion positions or elements to be erased.

Don't be fooled into thinking that this means const iterators are useless in general. They're not. They're perfectly useful with algorithms, because algorithms don't usually care what kind of iterators they work with, as long as they are of the appropriate category. const iterators are also acceptable for many container member functions. It's only some forms of insert and erase that are picky.

I wrote that const iterators are "largely" useless if you want to specify insertion positions or elements to be erased. The implication is that they are not completely useless. That's true. They can be useful if you can find a way to get an iterator from a const_iterator or from a const_reverse_iterator. That's often possible. It isn't *always* possible, however, and even when it is, the way to do it isn't terribly obvious. It may not be terribly efficient, either. Guideline 2 is devoted to this topic, so I'll defer it until then. For now, we have enough information to understand why it often makes sense to prefer iterators to const and reverse iterators:

- Some versions of insert and erase require iterators. If you want to call those functions, you're going to have to produce iterators. const and reverse iterators won't do.
- It's not possible to implicitly convert a const iterator to an iterator, and the technique described in Guideline 2 for generating an iterator from a const_iterator is neither universally applicable nor guaranteed to be efficient.
- Conversion from a reverse_iterator to an iterator may require iterator adjustment after the conversion. Guideline 3 explains when and why.

All these things conspire to make working with containers easiest, most efficient, and least likely to harbor subtle bugs if you prefer iterators to their const and reverse colleagues.

Practically speaking, you are more likely to have a choice when it comes to iterators and const_iterators. The decision between iterator and reverse_iterator is often made for you. You either need a front-to-back traversal or a back-to-front traversal, and that's pretty much that. You choose the one you need, and if that means choosing reverse_iterator, you choose reverse_iterator and use base to convert it to an iterator (possibly preceded by an offset adjustment, as we'll see in Guideline 3) when you want to make calls to container member functions that require iterators.

When choosing between iterators and const_iterators, there are reasons to choose iterators even when you could use a const_iterator and when you have no need to use the iterator as a parameter to a container member function. One of the most irksome involves comparisons between iterators and const_iterators. I hope we can all agree that this is reasonable code:

```
typedef deque<int> IntDeque;                      // STL container and
typedef IntDeque::iterator Iter;                  // iterator types are easier
typedef IntDeque::const_iterator ConstIter;       // to work with if you
                                                  // use some typedefs
Iter i;
ConstIter ci;
...                                               // make i and ci point into
                                                  // the same container
if (i == ci) ...                                  // compare an iterator
                                                  // and a const_iterator
```

All we're doing here is comparing two iterators into a container, the kind of comparison that's the bread and butter of the STL. The only twist is that one object is of type iterator and one is of type const_iterator. This should be no problem. The iterator should be implicitly converted into a const_iterator, and the comparison should be performed between two const_iterators.

With well-designed STL implementations, this is precisely what happens, but with other implementations, the code will not compile. The reason is that such implementations declare operator== for const_iterators as a member function instead of as a non-member function, but the cause of the problem is likely to be of less interest to you than the workaround, which is to swap the order of the iterators, like this:

```
if (ci == i) ...                        // workaround for when the
                                        // comparison above won't compile
```

This kind of problem can arise whenever you mix iterators and const_iterators (or reverse_iterators and const_reverse_iterators) in the same expression, not just when you are comparing them. For example, if you try to subtract one random access iterator from another,

```
if (i - ci >= 3) ...                    // if i is at least 3 beyond ci ...
```

your (valid) code may be (incorrectly) rejected if the iterators aren't of the same type. The workaround is what you'd expect (swap the order of i and ci), but in this case you have to take into account that you can't just replace i - ci with ci - i:

```
if (ci + 3 <= i) ...                    // workaround for when the above
                                        // won't compile
```

The easiest way to guard against these kinds of problems is to minimize your mixing of iterator types, and that, in turn, leads back to preferring iterators to const_iterators. From the perspective of const correctness (a worthy perspective, to be sure), staying away from

const_iterators simply to avoid potential implementation shortcomings (all of which have straightforward workarounds) seems unjustified, but in conjunction with the anointed status of iterators in some container member functions, it's hard to avoid the practical conclusion that const_iterators are not only less useful than iterators, sometimes they're just not worth the trouble.

Guideline 2: Use distance and advance to convert const_iterators to iterators.

Guideline 1 points out that some container member functions that take iterators as parameters insist on *iterators*; const_iterators won't do. So what do you do if you have a const_iterator in hand, and you want to, say, insert a new value into a container at the position indicated by the iterator? Somehow you've got to turn your const_iterator into an iterator, and you have to take an active role in doing it, because there is no implicit conversion from const_iterator to iterator.

I know what you're thinking. You're thinking, "When all else fails, get a bigger hammer." In the world of C++, that can mean only one thing: casting. Shame on you for such thoughts. Honestly, where do you get these ideas?

Let us confront this cast obsession of yours head on. Look what happens when you try to cast a const_iterator to an iterator:

```
typedef deque<int> IntDeque;                     // convenience typedefs
typedef IntDeque::iterator Iter;
typedef IntDeque::const_iterator ConstIter;

ConstIter ci;                                    // ci is a const_iterator

...

Iter i(ci);                              // error! no implicit conversion from
                                         // const_iterator to iterator

Iter i(const_cast<Iter>(ci));            // still an error! can't cast a
                                         // const_iterator to an iterator
```

This example happens to employ deque, but the outcome would be the same for list, set, multiset, map, multimap, as well as the non-standard STL containers based on hash tables [4]. The line using the cast *might* compile in the case of vector or string, but those are special cases we'll consider in a moment.

The reason the cast won't compile is that for these container types, iterator and const_iterator are completely different classes, barely more closely related to one another than are string and complex<double>.

Trying to cast one type to the other is nonsensical, and that's why the const_cast is rejected. static_cast, reinterpret_cast, and a C-style cast would lead to the same end.

Alas, the cast that won't compile might compile if the iterators' container were a vector or a string. That's because it is common for implementations of these containers to use real pointers as iterators. For such implementations, vector<T>::iterator is a typedef for T*, vector<T>::const_iterator is a typedef for const T*, string::iterator is a typedef for char*, and string::const_iterator is a typedef for const char*. With such implementations, the const_cast from a const_iterator to an iterator will compile and do the right thing, because the cast is converting a const T* into a T*. Even under such implementations, however, reverse_iterators and const_reverse_iterators are true classes, so you can't const_cast a const_reverse_iterator to a reverse_iterator. Also, even implementations where vector and string iterators are pointers might use that representation only when compiling in release mode [5]. All these factors lead to the conclusion that casting const iterators to iterators is ill-advised even for vector and string, because its portability is doubtful.

If you have access to the container a const_iterator came from, there is a safe, portable way to get its corresponding iterator, and it involves no casts that circumvent the type system. Here's the essence of the solution, though it has to be modified slightly before it will compile:

```
typedef deque<int> IntDeque;                    // as before
typedef IntDeque::iterator Iter;
typedef IntDeque::const_iterator ConstIter;

IntDeque d;

ConstIter ci;
...                                   // make ci point into d

Iter i(d.begin());                    // initialize i to d.begin()

advance(i, distance(i, ci));          // move i up to where ci is
                                      // (but see below for why this must
                                      // be tweaked before it will compile)
```

This approach is so simple and straightforward, it's startling. To get an iterator pointing to the same container element as a const_iterator, create a new iterator at the beginning of the container, then move it forward it until it's as far from the beginning of the container as the const_iterator is! This task is facilitated by the utility algorithms advance and distance, both of which are declared in <iterator>. distance reports how far apart two iterators into the same container are, and advance moves an iterator a specified distance. When i and ci point

into the same container, the expression advance(i, distance(i, ci)) makes i and ci point to the same place in the container.

Well, it would it if would compile, but it won't. To see why, look at the declaration for distance:

```
template <typename InputIterator>
typename iterator_traits<InputIterator>::difference_type
distance(InputIterator first, InputIterator last);
```

Don't get hung up on the fact that the return type of the function is 56 characters long and mentions dependent types like difference_type. Instead, focus your attention on the uses of the type parameter InputIterator:

```
template <typename InputIterator>
typename iterator_traits<InputIterator>::difference_type
distance(InputIterator first, InputIterator last);
```

When faced with a call to distance, your compilers must deduce the type represented by InputIterator by looking at the arguments used in the call. Look again at the call to distance in the code I said wasn't quite right:

```
advance(i, distance(i, ci));                    // move i up to where ci is
```

Two parameters are passed to distance, i and ci. i is of type Iter, which is a typedef for deque<int>::iterator. To compilers, that implies that InputIterator in the call to distance is deque<int>::iterator. ci, however, is of type ConstIter, which is a typedef for deque<int>::const_iterator. *That* implies that InputIterator is of type deque<int>::const_iterator. It's not possible for InputIterator to be two different types at the same time, so the call to distance fails, typically yielding some long-winded error message that may or may not indicate that the compiler couldn't figure out what type InputIterator is supposed to be.

To get the call to compile, you must eliminate the ambiguity. The easiest way to do that is to explicitly specify the type parameter to be used by distance, thus obviating the need for your compilers to figure it out for themselves:

```
advance(i, distance<ConstIter>(i, ci));         // figure the distance between
                                                // i and ci (as const_iterators),
                                                // then move i that distance
```

We now know how to use advance and distance to get an iterator corresponding to a const_iterator, but we have so far side-stepped a question of considerable practical interest: How efficient is this technique? The answer is simple. It's as efficient as the iterators allow it to be. For

random access iterators (such as those sported by vector, string, and deque), it's a constant-time operation. For bidirectional iterators (i.e., those for all other standard containers and for some implementations of the hashed containers [6]), it's a linear-time operation.

Because it may take linear time to produce an iterator equivalent to a const_iterator, and because it can't be done at all unless the container for the const_iterator is available when the const_iterator is, you may wish to rethink designs that require producing iterators from const_iterators.

Guideline 3: Understand how to use a reverse_iterator's base iterator.

Invoking the base member function on a reverse_iterator yields the "corresponding" iterator, but it's not really clear what that means. As an example, take a look at this code, which puts the numbers 1-5 in a vector, sets a reverse_iterator to point to the 3, and sets an iterator to the reverse_iterator's base:

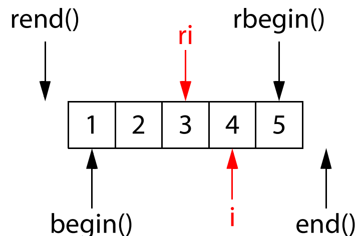
```
vector<int> v;

for (int i = 1; i <= 5; ++i) {                // put 1-5 in the vector
   v.push_back(i);
}

vector<int>:: reverse_iterator ri =           // make ri point to the 3
   find(v.rbegin(), v.rend(), 3);

vector<int>::iterator i(ri.base());           // make i the same as ri's base
```

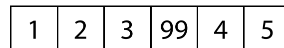
After executing this code, things can be thought of as looking like this:



This picture is nice, displaying the characteristic offset of a reverse_iterator and its corresponding base iterator that mimics the offset of rbegin() and rend() with respect to begin() and end(), but it doesn't tell you everything you need to know. In particular, it doesn't explain how to use i to perform operations you'd like to perform on ri.

As Guideline 1 explains, some container member functions accept only iterators as iterator parameters, so if you want to, say, insert a new element at the location identified by ri, you can't do it directly, because vector's insert function won't take reverse_iterators. You'd have a similar problem if you wanted to erase the element pointed to by ri. The erase member functions haughtily reject reverse_iterators, insisting instead on iterators. To perform erasures and some forms of insertion, you must convert reverse_iterators into iterators via the base function, then use the iterators to get the jobs done.

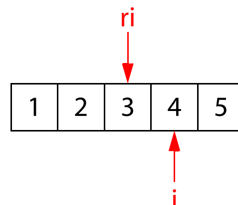
So let's suppose you *did* want to insert a new element into v at the position indicated by ri. In particular, let's assume you want to insert the value 99. Bearing in mind that ri is part of a traversal from right to left in the picture above and that insertion takes place in *front* of the element indicated by the iterator used to specify the insertion position, we'd expect the 99 to end up in front of the 3 with respect to a reverse traversal. After the insertion, then, v would look like this:



Of course, we can't use ri to indicate where to insert something, because it's not an iterator. We must use i instead. As noted above, when ri points at 3, i (which is ri.base()) points at 4. That's exactly where i needs to point for an insertion if the inserted value is to end up where it would if we had used ri to specify the insertion location. Conclusion?

- To emulate insertion at a position specified by a reverse_iterator ri, insert at the position ri.base() instead. For purposes of insertion, ri and ri.base() are equivalent, and ri.base() is truly the iterator *corresponding* to ri.

Let us now consider erasing an element. Look again at the relationship between ri and i in the original vector (i.e., prior to the insertion of 99):



If we want to erase the element pointed to by ri, we can't just use i, because i doesn't point to the same element as ri. Instead, we must erase the element *preceding* i. Hence,

- To emulate erasure at a position specified by a reverse_iterator ri, erase at the position *preceding* ri.base() instead. For purposes of erasure, ri and ri.base() are *not* equivalent, and ri.base() is *not* the iterator corresponding to ri.

It's worth looking at the code to perform such an erasure, because it holds a surprise.

```
vector<int> v;
...                                     // as above, put 1-5 in v
vector<int>:: reverse_iterator ri =     // as above, make ri point to the 3
  find(v.rbegin(), v.rend(), 3);
v.erase(--ri.base());                   // attempt to erase at the position
                                        // preceding ri.base(); for a vector,
                                        // this will typically not compile
```

There's nothing wrong with this design. The expression --ri.base() correctly specifies the element we'd like to erase. Furthermore, this code will work with every standard container except vector and string. It might work with vector and string, too, but for many vector and string implementations, it won't compile. In such implementations, iterators (and const_iterators) are implemented as built-in pointers, so the result of ri.base() is a pointer.

Both C and C++ impose the constraint that pointers returned from functions may not be modified, so for STL platforms where string and vector iterators are pointers, expressions like --ri.base() won't compile. To portably erase something at a position specified by a reverse_iterator, then, you must take pains to avoid modifying base's return value. No problem. If you can't decrement the result of calling base, just increment the reverse_iterator and *then* call base!

```
...                                     // as above
v.erase((++ri).base());                 // erase the element pointed to
                                        // ri; this should always compile
```

Because this approach works with every standard container, it is the preferred technique for erasing an element pointed to by a reverse_iterator.

It should now be clear that it's not accurate to say that a reverse_iterator's base member function returns the "corresponding" iterator. For insertion purposes, it does, but for erasure purposes, it

does not. When converting reverse_iterators to iterators, it's important that you know what you plan to do with the resulting iterator, because only then can you determine whether the iterator you have is the one you need.

Summary

STL containers offer four iterator types, but in practice, iterator tends to be the most useful. If you have a const_iterator and access to the container for that const_iterator, you can use distance and advance to generate the corresponding iterator. If you have a reverse_iterator, you can use its base member function to generate a "corresponding" iterator, but if you want to use the iterator in a call to erase, you'll probably need to increment the reverse_iterator before calling base.

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 Items 26-28 in *Effective STL*.

[2] For an entire article on this topic, look up "const T vs. T const" by Dan Saks in the February 1999 *Embedded Systems Programming*.

[3] The reason for iterator's special status of isn't clear. HP's original STL implementation had insert and erase take iterator parameters, and this aspect of the design wasn't reconsidered during standardization. It may be changed in the future, however, as Library Working Group Issue #180 notes that "this issue should be considered as part of a general review of const issues for the next revision of the standard." (C++ Library Issues may be viewed at http://anubis.dkuug.dk/jtc1/sc22/wg21/docs/lwg-defects.html.)

[4] The two most common implementations of STL containers based on hash tables are from Dinkumware and SGI. You'll find an overview of the Dinkumware approach in P. J. Plauger's November 1998 CUJ column, "Hash Tables." The only source I know of for an implementation overview of SGI's approach is Item 25 in *Effective STL* [1], but the interface is described at the SGI STL web site, http://www.sgi.com/tech/stl/HashedAssociativeContainer.html.

[5] This is the case when using STLport's Debug Mode. You can read about STLport and its debug mode at the STLport web site, http://www.stlport.org/.

[6] Dinkumware's hashed containers offer bidirectional iterators. SGI's, STLport's, and Metrowerks' hashed containers offer only forward iterators.

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, is a member of the Advisory Boards of NumeriX LLC and InfoCruiser Inc., and provides training and consulting services to clients worldwide. His web site is http://www.aristeia.com/.