A Proposal to Improve `const_iterator` Use from C++0X Containers

Document #:	WG21/N1674 = J16/04-0114
Date:	August 31, 2004
Revises:	None
Project:	Programming Language C++
Reference:	ISO/IEC IS 14882:2003(E)

Reply to: Walter E. Brown<wb@fnal.gov>
CEPA Dept., Computing Division
Fermi National Accelerator Laboratory
Batavia, IL 60510-0500

Contents

Sometimes a good idea comes to you when you are not looking for it.
— Kary B. Mullis

1 Introduction

This paper proposes to improve user access to the `const` versions of C++ container `iterator`s and `reverse_iterator`s.

This proposal was initially motivated by an example that arose in conjunction with the proposals for `decltype`/`auto` [JSGS03, JS03, JS04]. Intended primarily to demonstrate the convenience aspect of the proposed new use for the `auto` keyword, that example exhibited such looping code as:

```
//                                Listing 1
vector<MyType> v;
// fill v ...
for( auto it = v.begin(); it != v.end(); ++it )  {
  // use *it ...
}
```

In that context, the simple `auto` would replace today's rather unwieldy equivalent:

```
//                                 Listing 2
vector<MyType> v;
// fill v ...
typedef  vector<MyType>::iterator  iter;
for( iter it = v.begin(); it != v.end(); ++it )  {
  // use *it ...
}
```

However, when a container traversal is intended for inspection only, it is a generally preferred practice[1] to use a `const_iterator` in order to permit the compiler to diagnose `const`-correctness violations:

```
//                                 Listing 3
vector<MyType> v;
// fill v ...
typedef  vector<MyType>::const_iterator  c_iter;
for( c_iter it = v.begin(), end = v.end(); it != end; ++it )  {
  f( *it );   // error if f takes its argument by non-const reference
  *it = g();  // always an error
}
```

If a formulation such as shown above more clearly expresses the programmer's intent, there ought to be a way to obtain such expression using the more convenient `auto` as proposed. Alas, we find no straightforward way of doing so at the moment.

We initially felt that this counterexample demonstrated a weakness in the proposed use of `auto`. Upon further reflection, we now realize that the counterexample more likely demonstrates a weakness (*i.e.*, an omission) in the *iterators* portion of the interfaces to today's standard containers [ISO03, Clause 23]. In particular, unless a container has been declared `const`, there is today no means of directly obtaining a `const_iterator` via a call to its `begin` member.

Representative workarounds in common use today include (1) a `const_cast` of the container before calling `begin`, or (2) a (possibly implicit) `static_cast` of the `iterator` that results from such a call to `begin`:

```
//                                 Listing 4
typedef  vector<MyType>        vect;
typedef  vect::const_iterator  c_iter;
vect  v;
// alternatives:
c_iter it = const_cast<vect const &>(v).begin(); // 1
c_iter it = static_cast<c_iter>( v.begin() );    // 2 (explicit)
c_iter it = v.begin();                           // 2 (implicit)
```

Of these workaround alternatives, the implicit cast seems generally preferred by programmers. We postulate that this is due to the (deliberately) inconvenient syntax of modern C++ casts. We believe that programmers today lack a convenient means of directly expressing the use of a `const_iterator` in such contexts as we have described, and that the `decltype`/`auto` proposals would exacerbate such an omission from C++0X.

It is not only in connection with the `decltype`/`auto` proposals that this omission manifests. Indeed, every input iterator argument to a generic nonmodifying algorithm (such as those in Clause 25, section [lib.alg.nonmodifying], and elsewhere) provides another context in which programmers might reasonably prefer to provide an instance of a `const_iterator` rather than of an `iterator`. The `accumulate` algorithm provides one common example:

[1] As Herb Sutter succinctly exhorts, "Be `const` correct. In particular, use `const_iterator` when you are not modifying the contents of a container" [Sut05, p. 8].

```
//                                Listing 5
vector<double> v;
// fill v ...
cout << accumulate( v.begin(), v.end(), 0.0 );
```

The const-correctness aspect of type-safety would argue that it would be safer, in this example, to employ const_iterators than the iterators actually used above.[2]

A second illustration focuses on a user error in the context of the for_each algorithm:

```
//                                Listing 6
void reset( double & d )  { d = 0.0; }
void resee( double   d )  { cout << '␣' << d; }
vector<double> v;
// fill v ...
for_each( v.begin(), v.end(), reset ); // oops:  resee intended
```

Such erroneous code is today typically not caught at compile-time. Were const_iterators furnished instead of iterators, contemporary compilers would routinely diagnose this form of erroneous usage. However, as noted previously, it is currently at best inconvenient for a programmer to obtain a const_iterator from a non-const container.

2 Proposal

We believe that the C++ standard library should provide support, absent from C++03, so that a programmer can directly obtain a const_iterator from even a non-const container. **We therefore propose to augment C++ containers' interfaces** with new (member) functions cbegin and cend, and with analogous (member) functions crbegin and crend:

```
//                                Listing 7
const_iterator  cbegin()  const;
const_iterator  cend  ()  const;

const_reverse_iterator  crbegin()  const;
const_reverse_iterator  crend  ()  const;
```

3 Design alternatives

We believe that the desired functionality can be provided via either of two basic approaches. The alternatives are not mutually exclusive and, in fact, both could be adopted. (However, in Section 4 we provide proposed wording for our preferred Alternative 1 only.)

Additionally, either alternative could, in theory, replace the const overloads of the extant container member functions begin, end, rbegin, and rend. This is because the proposed functions would subsume these overloads' functionality. However, in order to preserve backwards compatibility, we prefer to retain all present forms of these member functions (although we are open to the possibility of deprecating their const overloads).

[2] Indeed, it has been (emphatically!) argued to us that the standard library should *diagnose* the use of iterators-to-non-const in the context of calls to standard nonmutating algorithms. We suggest to revisit this notion should C++0X be augmented with some form of concept-checking for template arguments.

3.1 Alternative 1: new container member functions

This first alternative proposes to augment each standard library container template with four new member functions (cbegin, cend, crbegin, and crend) as described above. This would permit user code of the form:

```
//                                Listing 8
vector<MyType> v;
// fill v ...
for( auto it = v.cbegin(), end = v.cend(); it != end; ++it )  {
  // use *it ...
}
```

We find such code very appealing, for it makes clear to a reader that the loop is non-mutating with respect to the container being traversed.

We also note that use of these proposed member functions in an inappropriate context such as the earlier:

```
//                                Listing 9
void reset( double & d )  { d = 0.0; }
void resee( double   d )  { cout << '␣' << d; }
vector<double> v;
// fill v ...
for_each( v.cbegin(), v.cend(), reset ); // oops:  resee intended
```

would now yield a compile-time diagnostic as desired.

3.2 Alternative 2: new generic adapter templates

This second alternative proposes to augment the standard library with four new function templates (cbegin, cend, crbegin, and crend) to provide a common interface to all containers. For example, a generic cbegin adapter might be implemented via a generic function such as:

```
//                               Listing 10
template< class C >
inline
typename C::const_iterator  cbegin( C const & c )  {
  return c.begin();
}
```

Availability of such adaptors would lead to client code of the form:

```
//                               Listing 11
vector<MyType> v;
// fill v ...
for( auto it = cbegin(v), end = cend(v); it != end; ++it )  {
  // use *it ...
}
```

While this generic adapter alternative seems quite straightforward, we nonetheless favor the member function approach as proposed above. It seems more in keeping with current C++ programming idioms, such as the parallel use of rbegin as a container member function rather than as a generic adapter.

We note that this generic adapter approach permits overloading so as to enable its use in connection with native arrays:

```
//                                Listing 12
template< class T, size_t N >
inline
T const *  cend( T const (& a)[N] )  {
  return a + N;
}
```

Whether this provides an advantage or a drawback is a matter of viewpoint. However, should this Alternative 2 be selected, then we would additionally propose, for consistency, to provide similar generic adapters for today's member functions `begin`, `end`, `rbegin`, and `rend`.

4 Proposed wording

The following few additions constitute the necessary changes to standardize our recommended proposal (Alternative 1 above) with respect to C++03. Because analogous additions would be desirable for homogeneous sequential containers[3] that might in the future be adopted into C++0X, we intend that approval of the present proposal constitute authorization for the Project Editor to make such additions at the appropriate time.

4.1 Container requirements

Add the following two new rows to **Table 65—Container requirements** in Clause 23, section [lib.container.requirements]:

expression	**return type**	**assertion/note** . . .	**complexity**
`a.cbegin();`	`const_iterator`	`const_cast<X const &>(X).begin();`	constant
`a.cend();`	`const_iterator`	`const_cast<X const &>(X).end();`	constant

4.2 Reversible container requirements

Add the following two new rows to **Table 66—Reversible container requirements** in Clause 23, section [lib.container.requirements]:

expression	**return type**	**assertion/note** . . .	**complexity**
`a.crbegin();`	`const_reverse_iterator`	`const_cast<X const &>(X).rbegin();`	constant
`a.crend();`	`const_reverse_iterator`	`const_cast<X const &>(X).rend();`	constant

4.3 Synopses

Add the following four declarations to the *iterators* part of Clause 21, section [lib.basic.string], as well as to the *iterators* parts of Clause 23, sections [lib.deque], [lib.list], [lib.vector], [lib.vector.bool], [lib.map], [lib.multimap], [lib.set], and [lib.multiset]:

```
const_iterator          cbegin() const;
const_iterator          cend() const;
const_reverse_iterator  crbegin() const;
const_reverse_iterator  crend() const;
```

[3] For example, the unordered associative containers and fixed size arrays in [Aus04].

5 Summary and conclusion

This paper has described the utility of container begin and end variations whose return types are always const_iterators, independent of a container's constness. The paper has presented use cases based on today's C++03 as well as on the significant C++0X decltype/auto proposals.

Two means of providing such missing functionality have been described herein: per-container member functions and generic adapter functions. In order to maintain parallelism with existing approaches, the former mechanism was recommended.

Finally, this paper has proposed wording consistent with that recommendation. We respectfully urge the C++ standards bodies to consider our proposals in a time frame consistent with that of the forthcoming C++0X standard.

6 Acknowledgments

I am pleased to acknowledge, with sincere thanks, a number of my Fermilab colleagues for their able advice and assistance during the production of this paper: Philippe Canal, Mark Fischler, Chris Green, John Marraffino, and Marc Paterno. I also wish to thank the Fermi National Accelerator Laboratory's Computing Division, sponsor of our participation in the C++ standards effort, for its support. Finally, many thanks to Richard Brown for his careful proofreading of several earlier drafts of this paper.

Bibliography

[Aus04] Matt Austern. (Draft) technical report on standard library extensions. Paper N1660, JTC1-SC22/WG21, July 16 2004. Online: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1660.pdf; same as ANSI NCITS/J16 04-0100.

[ISO98] *Programming Languages — C++, International Standard ISO/IEC 14882:1998(E)*. International Organization for Standardization, Geneva, Switzerland, 1998. 732 pp. Known informally as C++98.

[ISO03] *Programming Languages — C++, International Standard ISO/IEC 14882:2003(E)*. International Organization for Standardization, Geneva, Switzerland, 2003. 757 pp. Known informally as C++03; a revision of [ISO98].

[JS03] Jaako Järvi and Bjarne Stroustrup. Mechanisms for querying types of expressions: Decltype and auto revisited. Paper N1527, JTC1-SC22/WG21, September 21 2003. Online: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1527.pdf; same as ANSI NCITS/J16 03-0110.

[JS04] Jaako Järvi and Bjarne Stroustrup. Decltype and auto (revision 3). Paper N1607, JTC1-SC22/WG21, February 17 2004. Online: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1607.pdf; same as ANSI NCITS/J16 04-0047.

[JSGS03] Jaako Järvi, Bjarne Stroustrup, Douglas Gregor, and Jeremy Siek. Decltype and auto. Paper N1478, JTC1-SC22/WG21, April 28 2003. Online: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1478.pdf; same as ANSI NCITS/J16 03-0061.

[Sut05] Herb Sutter. *Exceptional C++ Style: 40 New Engineering Puzzles, Programming Problems, and Solutions*. Addison-Wesley, Reading, MA, USA, 2005. ISBN 0-201-76042-8. xiv + 325 pp. LCCN QA76.73.C153S885 2005.