A Proposal to Improve `const_iterator` Use (version 2)

Document #: WG21/N1865 = J16/05-0125
Date: 2005–08–24
Revises: WB21/N1674 = J16/04-0114
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

1 Introduction **1**
2 Motivation **2**
3 Proposal **4**
4 Two design alternatives **4**
5 Proposed wording for Alternative 1 **6**
6 Proposed wording for Alternative 2 **6**
7 Discussion **9**
8 Summary and conclusion **10**
9 Acknowledgments **10**
Bibliography **11**

> *A new idea is delicate. It can be killed by a sneer or a yawn; it can be stabbed to death by a quip and worried to death by a frown on the right man's brow.*
>
> — Charles Brower

1 Introduction

This paper, as requested by the Library Working Group, is an expanded version of N1674 [Bro04], a proposal to improve user access to the `const` versions of C++ container `iterator`s and their `reverse_iterator` variants. N1674 had its first hearing before the LWG at the 2005 Lillehammer meeting. LWG reaction at that meeting was largely bimodal.

While two LWG members flatly saw no need for this proposed library enhancement ("no user demand" was the stated objection), several others felt strongly that the proposal was very straightforward and addressed "an obvious omission" from the current standard. More importantly, these latter individuals concurred that the "omission" would be exacerbated in C++0X by the `auto` facility.

Given no clear consensus, the LWG in Lillehammer requested a revised paper that addressed these points in more detail. The LWG also requested additional proposed wording: Where N1674

provided proposed wording for our Alternative 1 (chapter 5 herein) only, the LWG also wished to have wording for Alternative 2 (chapter 6) as well, in order to consider both on an equal footing and thus to make realistic the options of adopting either or both of the formulations.

2 Motivation

2.1 Background: a first example

This proposal was initially motivated by an example that arose in conjunction with the proposals for decltype/auto [JSGS03, JS03, JS04, JSDR04a][1]. 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 ...
}
```

This seems a clear advantage to the programmer, and is a strong argument in favor of the auto proposals.

2.2 The problem

When a container, during traversal, is intended for inspection only, it is a generally preferred practice[2] to traverse via 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.

[1] The auto proposal has since been decoupled from decltype and is being considered separately [JSDR04b, JSDR05].

[2] 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].

We initially felt that this counterexample demonstrated a weakness in the proposed use of `auto`. Upon further reflection, we came to realize that the counterexample instead 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 standard means of directly obtaining a `const_iterator` via a call to its `begin` member.

2.3 Today's workarounds

Representative workarounds in common use today involve casting and include (1) an explicit `const_cast` (or call to an equivalent user-defined function often named `as_const()`) of the container before calling `begin`, or (2) a (possibly implicit) `static_cast` of the `iterator` that results from such a call to `begin`:

```
1 //                                  Listing 4
2 typedef  vector<MyType>         vect;
3 typedef  vect::const_iterator   c_iter;
4 vect  v;
5 // alternatives:
6 c_iter it = const_cast<vect const &>(v).begin(); // 1 (explicit)
7 c_iter it = as_const(v).begin();                 // 1 (implicit)
8 c_iter it = static_cast<c_iter>( v.begin() );    // 2 (explicit)
9 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.

It is our position 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: How would a programmer take advantage of `auto`'s signfnicant convenience while simultaneously obtaining the well-known benefits of `const`-correctness?

2.4 More examples

It is not only in connection with the `decltype`/`auto` proposals that the above-described 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 //                                  Listing 5
2 vector<double> v;
3 // fill v ...
4 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_iterator`s than the `iterator`s actually used above.[3]

Another illustration focuses on a user error in the context of the `for_each` algorithm:

[3] 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 provide some form of concept-checking for template arguments.

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

Such erroneous code[4] 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.

3 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 or nonmember) functions cbegin and cend, and with analogous (member or nonmember) functions crbegin and crend:

```
1 //                                  Listing 7
2 const_iterator  cbegin()  const;
3 const_iterator  cend  ()  const;
4
5 const_reverse_iterator  crbegin()  const;
6 const_reverse_iterator  crend  ()  const;
```

4 Two 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. We will provide proposed wording for Alternative 1 in section 5 and proposed wording for Alternative 2 in section 6.

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).

4.1 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:

[4] "The function passed as the third argument is not permitted to make any modifications to the sequence . . . " [Rog03, §13.8].

```
//                                  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.

4.2 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 ...
}
```

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 additionally propose, for consistency, to provide similar generic adapters for today's member functions `begin`, `end`, `rbegin`, and `rend`.

5 Proposed wording for Alternative 1

The following few additions constitute the necessary changes, with respect to C++03, to standardize our proposed Alternative 1 (section 4.1).

5.1 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

5.2 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

5.3 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;
```

6 Proposed wording for Alternative 2

The following few additions constitute the necessary changes, with respect to C++03, to standardize our proposed Alternative 2 (section 4.2).

6.1 New function templates

Add a new section to Clause 24, with the following content.

24.x Container endpoint access To access container endpoints in a generic fashion, the library provides a number of overloaded function templates. In the remainder of this section, template parameter `C` shall denote an instance of one of the following library containers: `deque`, `list`, `queue`, `stack`, `vector`, `map`, `multimap`, `set`, `multiset`.

24.x.1 begin() and cbegin()

```
template<class C>
  typename C::iterator        begin( C & c );
```

```
  template<class C>
    typename C::const_iterator  begin( C const & c );
  template<class C>
    typename C::const_iterator  cbegin( C const & c );
```

1. Returns: Each of these functions returns `c.begin()`.

```
template<class T, size_t N>
  T *        begin( T (&a)[N] );
template<class T, size_t N>
  T const *  begin( T const (&a)[N] );
template<class T, size_t N>
  T const *  cbegin( T const (&a)[N] );
```

2. Returns: Each of these functions returns `a+0`.

24.x.2 end() and cend()

```
  template<class C>
    typename C::iterator        end( C & c );
  template<class C>
    typename C::const_iterator  end( C const & c );
  template<class C>
    typename C::const_iterator  cend( C const & c );
```

3. Returns: Each of these functions returns `c.end()`.

```
  template<class T, size_t N>
    T *        end( T (&a)[N] );
  template<class T, size_t N>
    T const *  end( T const (&a)[N] );
  template<class T, size_t N>
    T const *  cend( T const (&a)[N] );
```

4. Returns: Each of these functions returns `a+N`.

24.x.3 rbegin() and crbegin()

```
  template<class C>
    typename C::reverse_iterator        rbegin( C & c );
  template<class C>
    typename C::const_reverse_iterator  rbegin( C const & c );
  template<class C>
    typename C::const_reverse_iterator  crbegin( C const & c );
```

5. Returns: Each of these functions returns `c.rbegin()`.

```
template<class T, size_t N>
  reverse_iterator<T *>        rbegin( T (&a)[N] );
template<class T, size_t N>
  reverse_iterator<T const *>  rbegin( T const (&a)[N] );
template<class T, size_t N>
  reverse_iterator<T const *>  crbegin( T const (&a)[N] );
```

6. Returns: Each of these functions returns `a+N`.

24.x.2 rend() and crend()

```
  template<class C>
    typename C::reverse_iterator        rend( C & c );
  template<class C>
    typename C::const_reverse_iterator  rend( C const & c );
  template<class C>
    typename C::const_reverse_iterator  crend( C const & c );
```

7. Returns: Each of these functions returns `c.rend()`.

```
  template<class T, size_t N>
    reverse_iterator<T *>        rend( T (&a)[N] );
  template<class T, size_t N>
    reverse_iterator<T const *>  rend( T const (&a)[N] );
  template<class T, size_t N>
    reverse_iterator<T const *>  crend( T const (&a)[N] );
```

8. Returns: Each of these functions returns `a+0`.

6.2 Synopses

Add the following declarations into namespace `std` in Clause 24, section [lib.iterator.synopsis]:

```
// 24.x Container endpoint access
template<class C>
  typename C::iterator        begin( C & );
template<class C>
  typename C::const_iterator  begin( C const & );
template<class C>
  typename C::const_iterator  cbegin( C const & );

template<class C>
  typename C::iterator        end( C & );
template<class C>
  typename C::const_iterator  end( C const & );
template<class C>
  typename C::const_iterator  cend( C const & );

template<class C>
  typename C::reverse_iterator        rbegin( C & );
template<class C>
  typename C::const_reverse_iterator  rbegin( C const & );
template<class C>
  typename C::const_reverse_iterator  crbegin( C const & );

template<class C>
  typename C::reverse_iterator        rend( C & );
template<class C>
  typename C::const_reverse_iterator  rend( C const & );
template<class C>
  typename C::const_reverse_iterator  crend( C const & );

template<class T, size_t N>
  T *        begin( T (&)[N] );
template<class T, size_t N>
  T const *  begin( T const (&)[N] );
template<class T, size_t N>
  T const *  cbegin( T const (&)[N] );

template<class T, size_t N>
  T *        end( T (&)[N] );
template<class T, size_t N>
```

```
  T const *  end( T const (&)[N] );
template<class T, size_t N>
  T const *  cend( T const (&)[N] );

template<class T, size_t N>
  reverse_iterator<T *>        rbegin( T (&)[N] );
template<class T, size_t N>
  reverse_iterator<T const *>  rbegin( T const (&)[N] );
template<class T, size_t N>
  reverse_iterator<T const *>  crbegin( T const (&)[N] );

template<class T, size_t N>
  reverse_iterator<T *>        rend( T (&)[N] );
template<class T, size_t N>
  reverse_iterator<T const *>  rend( T const (&)[N] );
template<class T, size_t N>
  reverse_iterator<T const *>  crend( T const (&)[N] );
```

7 Discussion

7.1 Naming

We understand that the names of our proposed functions and templates have been the topic of some discussion on the Boost mailing list, and that some preference has been expressed favoring longer versions of these names (*e.g.*,, `const_begin` instead of `cbegin`). While we would not object in principle to longer names, we nonetheless prefer our (shorter) names for two reasons: First, the shorter names seem to us to be more consistent with and more in keeping with existing practice in the current language (*viz.*, `rbegin`). Second, the longer names approach the unwieldiness of the casting syntax, especially when used twice within a single `for`-statement (*e.g.*, to obtain both ends of an iterator range).

7.2 Additional containers

While we focus herein on the native and library containers of only C++03, we believe analogous additions would be desirable for homogeneous sequential containers[5] that might in the future be adopted into C++0X. We therefore intend that approval of either or both of the present proposal's Alternatives shall constitute authorization for the Project Editor at the appropriate time to make comparable additions with respect to those containers.

It may also be possible to extend this proposal (especially Alternative 2) to additional container-like entities such as `std::pair` and `std::tr1::tuple`, especially when homogeneous. We have not proposed such extension because it is unclear to us how (or even whether) to handle the heterogeneous cases.

7.3 A complementary language proposal

In the event that rvalue references [AAD+05] are adopted into C++0X, and that this paper's Alternative 2 is also adopted, then we intend that the proposed functions be additionally overloaded with variants taking arguments by const rvalue reference. Such approvals therefore shall constitute authorization for the Project Editor at the appropriate time to make the obvious additions with respect to call by const rvalue reference.

[5] For example, the unordered associative containers and fixed size arrays in TR1 [Aus05, clause 6].

7.4 A complementary library proposal

A forthcoming proposal (by Thorsten Ottosen) for a "Range Library" is similar, in some respects, to the current proposal. However, we believe that there is no fundamental inconsistency between the two proposals.

The focus of the (as yet unpublished) Ottosen proposal is on introducing, into the C++0X library, additional functionality that is based on the notion of iterator ranges. In contrast, our proposal has a far less ambitious goal, namely in the current library to improve access to const_iterators. The present proposal relies on no new language features, and can easily be implemented in today's C++. In contrast, although based on an existing boost library by the same name, the "Range Library" proposal in its present (prepublication) form does explicitly rely[6] on new language features (auto, decltype, rvalue references) as well as on other new library features (iterator concepts [ASW04b] and iterator adaptors [ASW04a]) in order to make the proposal easier to specify.

In addition, the present proposal provides complete wording. We therefore respectfully recommend that the proposals be considered independently. While there may be some overlap, they appear to us to be mutually consistent.

8 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. The two approaches are orthogonal; either or both may be adopted.

Finally, this paper has proposed wording for both alternatives, and has discussed their interaction with other proposals. We respectfully urge the C++ standards bodies to consider the present proposal in a time frame consistent with that of the forthcoming C++0X standard.

9 Acknowledgments

I am pleased to acknowledge, with sincere thanks, a number of individuals for their technical comments during the production of this paper: Mark Fischler, Chris Green, Jim Kowalkowski, John Marraffino, Thorsten Ottosen, 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 drafts of this paper.

[6] Example: `template< class T > auto begin( T&& t ) -> decltype( t.begin() );`

Bibliography

[AAD+05] David Abrahams, J. Stephen Adamczyk, Peter Dimov, Howard E. Hinnant, and Andreas Hommel. A proposal to add an rvalue reference to the C++ language: Proposed wording. Paper N1770, JTC1-SC22/WG21, March 5 2005. Online: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1770.asc; same as ANSI NCITS/J16 05-0030.

[ASW04a] David Abrahams, Jeremy Siek, and Thomas Witt. Iterator facade and adaptor. Paper N1641, JTC1-SC22/WG21, April 10 2004. Online: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1641.asc; same as ANSI NCITS/J16 04-0081.

[ASW04b] David Abrahams, Jeremy Siek, and Thomas Witt. New iterator concepts. Paper N1640, JTC1-SC22/WG21, April 10 2004. Online: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1640.asc; same as ANSI NCITS/J16 04-0080.

[Aus05] Matt Austern. (Draft) technical report on standard library extensions. Paper N1836, JTC1-SC22/WG21, June 24 2005. Online: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1836.pdf; same as ANSI NCITS/J16 05-0096.

[Bro04] Walter E. Brown. A proposal to improve `const_iterator` use from C++0x containers. Paper N1674, JTC1-SC22/WG21, August 31 2004. Online: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1674.pdf; same as ANSI NCITS/J16 04-0114.

[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.

[JSDR04a] Jaako Järvi, Bjarne Stroustrup, and Gabriel Dos Reis. Decltype and auto (revision 4). Paper N1705, JTC1-SC22/WG21, September 12 2004. Online: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1705.pdf; same as ANSI NCITS/J16 04-0145.

[JSDR04b] Jaako Järvi, Bjarne Stroustrup, and Gabriel Dos Reis. Deducing the type of variable from its initializer expression. Paper N1721, JTC1-SC22/WG21, October 21 2004. Online: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1721.pdf; same as ANSI NCITS/J16 04-0161.

[JSDR05] Jaako Järvi, Bjarne Stroustrup, and Gabriel Dos Reis. Deducing the type of variable from its initializer expression (revision 2). Paper N1794, JTC1-SC22/WG21, April 13 2005. Online: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1794.pdf; same as ANSI NCITS/J16 05-0054.

[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.

[Rog03] Rogue Wave Software. Online documentation: Standard C++ library module user's guide. Online: http://www.roguewave.com/support/docs/sourcepro/stdlibug/index.html, August 2003.

[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.