Doc. No.: WG21/N1015
X3J16/96-0197
Date: November 11, 1996
Project: C++ Standard Library
Reply to: Pete Becker
pbecker@oec.com

Clause 24 (Iterators Library) Issues

Work Group: Library Clause 24
Issue Number: 24-021
Title: Separate Header for Stream Iterators
Section: 24.4
Status: active
Description:
From public review:

Drawing iostream into an implementation that just needs iterators is most unfortunate.

The current iterator header includes headers <ios> and <streambuf> to handle the stream iterators in 24.4. This requires all of I/O to be included in the iterators header. Yet I/O only needs this if the iterators are used.

If a new header is used should it be in clause 24 or in clause 27?
Is <iositer> a good name for the new header?
Should the stream iterators be incorporated into current I/O headers?

From Nathan Myers:
Message c++std-lib-4174
There are natural places for each of these iterator templates.
Move istream_iterator<> to <istream>.
Move ostream_iterator<> to <ostream>.
Move istreambuf_iterator<> and ostreambuf_iterator<> to <streambuf>.
Add forward declarations of all four to <iosfwd>.

Changes to be made would include:
Move the stream iterators into the I/O headers.

Remove #include's for iosfwd, ios, and streambuf from 24.1.6 [lib.iterator.tags] Header <iterator> synopsis and tags for subclause 24.4.

Move istream_iterator to <istream>, ostream_iterator to <ostream>, and the streambuf iterators to <streambuf>. Add forward declarations of all four to <iosfwd>. Add #include <iterator> in these headers.

Proposed Resolution:
Close the issue without change.

Because there is no longer any requirement that specific I/O
headers be included with <iterator>, it is possible to implement
the stream iterators without including all of I/O.

Requester: Public Review & Library WG
Owner: David Dodgson (Iterators)
Emails: lib-4174,4186,4191,4199,4202
Papers:

Work Group: Library Clause 24
Issue Number: 24-038
Title: Removal of proxy class
Section: 24.4.3 [lib.istreambuf.iterator]
Status: active
Description:
24.4.3:

The changes to input iterator semantics make the proxy class
an implementation detail. It should not be required as part
of the standard.

>From P.J. Plauger in N0795:
24.4.3:
istreambuf_iterator should remove all references to proxy, whether
or not Koenig's proposal passes to make more uniform the definition
of all input iterators. It is over specification.

24.4.3.1:
istreambuf_iterator::proxy is not needed (once istreambuf_iterator
is corrected as described below). It should be removed.

24.4.3.2:
istreambuf_iterator(const proxy&) should be removed.

24.4.3.4:
istreambuf_iterator::operator++(int) Effects should say that it
saves a copy of *this, then calls operator++(), then returns
the stored copy. Its return value should be istreambuf_iterator,
not proxy.

Editorial box 69 suggests that proxy be replaced by an opaque unnamed type.

See also issue 42 regarding the return type of operator++(int).

Proposed Resolution:

Input iterators do not require a specific class to be returned from operator++(int). (Nor do output iterators - see issue 42). The requirements are such that *i++ must work. The actual type returned should be any that satisfy the requirements. This suggests that the implementer be given some latitude in the definition. All other instances of operator++(int) in Clause 24 return a value of the iterator type. The proposal is to have istreambuf_iterator::operator++(int) return a type which is implementation defined.

A. (use implementation defined)
24.5.3 synopsis
remove 'class proxy' and 'istreambuf_iterator(const proxy& p)'
change 'proxy operator++(int)' to 'implementation_defined operator++(int)'
remove 24.5.3.1
remove istreambuf_iterator(const proxy& p) from 24.5.3.2

B. (make proxy a class for exposition only)
change all occurrences of proxy in 24.5.3 to boldface
remove the code portion of 24.5.3.1, change proxy to boldface
change proxy to boldface in 24.5.3.2

Requester: David Dodgson
Owner: David Dodgson (Iterators)
Emails:
Papers: N0795, Updated Issues List for Library, pre-Tokyo
N0833, Proposed Iterators Changes, pre-Santa Cruz

Work Group: Library Clause 24
Issue Number: 24-042
Title: Return type for operator++(int)
Section: 24.3.2 24.4.2 24.4.4
Status: active
Description:
24.:

>From Judy Ward (j_ward@decc.enet.dec.com):

operator++(int) for:

back_insert_iterator
front_insert_iterator
insert_iterator
ostream_iterator
[Note: ostreambuf_iterator is also affected]

are all currently specified in the standard as:

```
insert_iterator<Container> operator++(int);
```

I was wondering why the HP implementation has them as:

```
insert_iterator<Container>& operator++(int);
```

The reason is that if the user tries something like:

```
*i++ = 0;
```

where i is an insert_iterator, an insert_iterator<Container> copy ctor would automatically be called under the current specification. I don't think you want this to happen, especially in the HP implementation where the private data members are of type Container& and Container::iterator.

So my proposal is to return by reference in each of the postfix ++ operators.

See also issue 32 regarding the return type of insert_iterator:: operator++(int).

Discussion:

In general, the result of operator++(int) is a temporary which is needed only for the duration of the expression. The iterators described in Clause 24 are described uniformly in this regard. However, the iterators specified in this issue are all output iterators. For them there is no need to return a temporary (usually (*this) is returned). The standard could be changed to return a reference for these items.

The specifications for output iterators (and input iterators) do not require the return result for operator++(int) to be of the same class. The specifications are therefore somewhat open-ended. However, some return value must be specified in the iterators described in this section. One possibility is to change the return types to references, another is to leave them as they are but provide additional discussion in the introduction stating that any return type which meets the specifications is conforming. It may be argued that a reference return type meets an 'as-is' requirement for the iterators. A third possibility is to make them implementation-defined.

Resolution:
Update the return type for operator++(int) in
24.4.2.1 [lib.back.insert.iterator], 24.4.2.2.4,
24.4.2.3 [lib.front.insert.iterator], 24.4.2.4.4,
24.4.2.5 [lib.insert.iterator], 24.4.2.6.4,
24.5.2 [lib.ostream.iterator],
24.5.4 [lib.ostreambuf.iterator], 24.5.4.2

Requester: Judy Ward
Owner: David Dodgson (Iterators)
Emails:
Papers:

Work Group: Library Clause 24
Issue Number: 24-044
Title: Simplification of reverse iterator adapters
Section: 24.2 24.4.1
Status: active
Description:
24.4.1 [lib.reverse.iterators]:
Previous changes to iterators allow reverse_bidirectional_iterators to be combined with reverse_iterators. The bidirectional case could be eliminated as a separate class, only reverse_iterators would be needed.

An additional change could be made to the iterator_traits and iterator templates. This change would include the Reference and Pointer types in the traits. Reference is the type returned for a reference for the value_type, Pointer for a pointer to the value_type. Currently these are parameters for the reverse_ iterators only. Adding them would make them available for all iterators. It would require uses of the iterator template to possibly specify 5 parameters instead of 3 (default arguments would allow fewer arguments to be specified in many cases). It would also allow only the base iterator to be needed as an argument to the reverse_iterator template.

Question: Currently an output iterator is defined using:

```
class out_iter : public iterator<output_iterator_tag, void> { };
```

Will this code be legal if this change is made (because the default for Reference would use void&). If not, can a specialization be defined to make it work?

Proposed Resolution:

A. Eliminate reverse_bidirectional Iterators

Previous changes to iterators make reverse_bidirectional_iterator superfluous. The reverse_iterator template can be written to handle both random access and bidirectional iterators.

Remove sections 24.4.1.1 and 24.4.1.2

B. Include the Pointer and Reference typedefs in iterator<>

Including these types would make iterator adapters easier to write.

Changes to the WP are in N0910 / 96-0092 with these updates:

3.3 bullet 2:
the base class for reverse_iterator can be
iterator_traits<Iterator>

3.3 bullet 5:
the penultimate word should be "const_iterator" not
"reverse_iterator"

Requester: Matt Austern, Angelika Langer, Alex Stepanov
Owner: David Dodgson (Iterators)
Emails: lib-4826-27,4833,4836,4847,4855
Papers: 96-0092 / N0910, "Simplification of reverse iterator adapters",
pre-Stockholm

Work Group: Library Clause 24
Issue Number: 24-045
Title: Descriptions of stream iterators
Section: 24.5.1 and 24.5.2
Status: active
Description:
24.5.1 and 24.5.2
[lib.istream.iterator] and [lib.ostream.iterator]

All other iterators in this section have a description of the semantics of each individual member function. The istream_ and ostream_ iterators do not. There is simply a listing of the headers with no following descriptions.

Proposed Resolution:

Add the following protected members in 24.5.1

```
protected:
  basic_istream<charT,traits>* in_stream;
  T value;
```

Add the following descriptions:
24.5.1.1 istream_iterator constructors and destructor

istream_iterator();

Effects: Constructs the end-of-stream iterator.

istream_iterator(istream_type& s);

Effects: Initializes in_stream with s. value may be initialized during construction or the first time it is referenced.

istream_iterator(const istream_iterator<T,Distance>& x);

Effects: Constructs a copy of x.

~istream_iterator();

Effects: The destructor for value is performed.

24.5.1.2 istream_iterator operations

const T& operator*() const;

Returns: value

const T* operator->() const;

Returns: &(operator*())

istream_iterator<T,Distance>& operator++();

Effects: *in_stream >> value
Returns: *this

istream_iterator<T,Distance> operator++(int);

Effects:

```
   istream_iterator<T,Distance> tmp = *this;
   *in_stream >> value;
   return (tmp);
```

```
template <class T, class Distance>
  bool operator==(const istream_iterator<T,Distance>& x,
                const istream_iterator<T,Distance>& y);
```

Returns: (x.in_stream == y.in_stream)

Add the following protected members to 24.5.2

```
protected:
  basic_ostream<charT, traits> out_stream;
  const char* delim;
```

Add the following descriptions:

24.5.2.1 ostream_iterator constuctors and destructor

```
ostream_iterator(ostream_type& s);
```

Effects: Initializes out_stream with s and delim with null.

```
ostream_iterator(ostream_type& s, const charT* delimiter);
```

Effects: Initializes out_stream with s and delim with delimiter.

```
ostream_iterator(const ostream_iterator<T>& x);
```

Effects: Constructs a copy of x.

```
~ostream_iterator();
```

Effects: The iterator is destroyed.

24.5.2.2 ostream_iterator operations

```
ostream_iterator<T>& operator=(const T& value);
```

Effects:

```
   *out_stream << value;
   if (delim != 0) *out_stream << *delim;
   return (*this);
```

```
ostream_iterator<T>& operator*();
```

Returns: *this

```
ostream_iterator<T>& operator++();
ostream_iterator<T>  operator++(int);
```

Returns: *this

Requester: David Dodgson
Owner: David Dodgson (Iterators)
Emails:
Papers: