Clause 24 (Iterators) Issues List (Rev. 2)

David Dodgson
dsd@tr.unisys.com
UNISYS

The following list contains the issues for Clause 24 on Iterators. The list is divided based upon the status of the issues. The status is either *active* - under discussion, *resolved* - resolution accepted but not yet in the working paper, *closed* - working paper updated, or *withdrawn* - issue withdrawn or rejected. They are numbered chronologically as entered in the list. Only the active and resolved issues are presented here. Those wishing a complete list may request one.

The proposed resolutions are my understanding of the consensus on the reflector.

1. Revision History

Revision 0 -	5/26/95	pre-Monterey	N0702/95-0102
Revision 1 -	9/25/95	pre-Tokyo	N0773/95-0173
Revision 2 -	11/30/95	pre-Santa Cruz	N0832/96-0014

2. *Active Issues*

Work Group: Library Clause 24
Issue Number: 24-003
Title: const operation for iterators
Section: 24.3
Status: active
Description:

24.3.1 p24-13 Box 108

Suggest that the operator *() for STL iterators be made into a const operation.

The function

```
void fn (const ReverseIterator & x) {
 ...
   y = x*;
 ...
}
```

shows that the operation * is not defined as const in the reverse_iterator (DRAFT 20 Sept 1994, 24.2.1.2). However, the body of the function does not modify the iterator object.

Of course, const Iterator is different from const_iterator and from const const_iterator.

This change was accepted in Monterey (see N740). However, in box 108, Corfield says it seems wrong to have const member functions return a reference or a pointer to non-const T. He believes this should be reconsidered for operator* and operator->.

It has been further suggested in public review that const should also be used the descriptions in 24.3.1.2.2 and 24.3.1.2.3. (Editorial if accepted.) Also, the same decisions should be made

for reverse_iterator in 24.3.1.3, 24.3.1.4.2, and 24.3.1.4.3.

Proposed Resolution:
Both base() and operator*() should be const.
Accepted in Monterey - N740

As stated above, there is a difference between const iterator and const_iterator. The template parameters must specify const if const T is desired.

Reverse_iterator should be treated the same as reverse_bidirectional_iterator.

Requestor: Bob Fraley <fraley@porter.hpl.hp.com>
David Olsen (public review comment #17)
Owner: David Dodgson (Iterators)
Emails: c++std-lib-3135
Papers: N740 - Small Changes

Work Group: Library Clause 24
Issue Number: 24-012
Title: Addition operators added to iterators
Section: 24.1
Status: active
Description:
24.1.3-24.1.5 p24-3 to 24-6:
Add addition and subtraction operators to forward, bidirectional and reverse iterators.

Alex Stepanov in lib-3611:
And if you reconsider the iterator requirements, you might as well reconsider the exclusion of + (and related operators) for non-random iterator categories. I really hate advance and distance templates. They are such a pain to use and they are really ugly. (To see what I mean, take a look at what we now need to do to implement, say, lower_bound algorithm. It is in algo.h in our implementation.)

Later discussions show that this should not include output iterators, and at most only - operations for input iterators.

Discussion at Monterey meeting:
The library subgroup was in favor of this change. It was felt that the added convenience of this is worthwhile. The cost is in making it unclear that '+' can be a linear operation. If operator+ is added to the base forward iterator in terms of using the advance template it should add little cost to existing implementations.

Proposed Resolution:
See paper N0739R1 in the post-Monterey mailing.

Requestor: Alex Stepanov
Owner: David Dodgson (Iterators)
Emails: lib 3611-3613
Papers: Operator+ and Operator- for Iterators, 95-0139/N0739,
David Dodgson, post-Monterey

Work Group: Library Clause 24
Issue Number: 24-013
Title: Const declaration of operator[]
Section: 24.3.1.3 [lib.reverse.iterator]
Status: active
Description:
24.3.1.3 p24-15.16: [Box 109] and
24.3.1.4.11 p24-19: [Box 110]
Should operator[] of reverse_iterator be specified as const?
Proposed Resolution:

Same resolution as issue 3 (Box 108 in lib.reverse.bidir.iter section 24.3.1.1 for reverse_bidirectional_iterator)

This was accepted and added to the working paper. Box 109 from Corfield states that he thinks it is wrong to return a non-const T from a const member function.

Again, this should be resolved as issue 3.

Resolution: specified as const - See N740
Requestor: Editorial box
Owner: David Dodgson (Iterators)
Emails:
Papers: Small Changes, 95-0140/N0740, David Dodgson, post-Monterey

---

Work Group: Library Clause 24
Issue Number: 24-015
Title: Char-oriented stream iterators
Section: 24.4.3 [lib.istreambuf.iterator]
Status: active
Description:

24.4.3 p24-23: [Box 111]

The istream_iterator and ostream_iterator are defined only for the char-oriented, but not the wchar_t-oriented or parameterized streams.

Resolution:
Requestor: Editorial Box
Owner: David Dodgson (Iterators)
Emails:
Papers:

Work Group: Library Clause 24
Issue Number: 24-017
Title: Exceptions in ostreambuf_iterator
Section: 24.4.4 [lib.ostreambuf.iterator]
Status: active
Description:

24.4.4.1 and 24.4.4.3:

Nathan Myers in message lib-3812:

As Plauger noted in some previous mail relating to locale, the ostreambuf_iterator used to decouple iostream from the locale facet interface provides no mechanism for reporting output errors. Changing the interface to allow the iterator to be compared against an "end" iterator doesn't help; again (as Plauger points out) the Output Iterator abstraction doesn't support comparison, and standard algorithms don't assume it.

** Discussion

Failures of abstraction in C++ are handled by throwing an exception. Output Iterators are, in general, allowed to throw if they cannot perform an operation; it is necessary only to specify that this is what ostream_iterator does.

ostream and locale need to have specified what happens in the event of an output error, so the failure can be handled according to ostream's policy without imposing knowledge of it upon all locale facets.

Another option is to note the error without throwing the exception. A member function can return a bool value to indicate if the operation failed.

Proposed Resolution

1. Section 24.4.4.1 [lib.ostreambuf.iter.cons], for operator=(charT c),

Add:
Effects: If no previous call has returned traits::eof(),
calls sbuf_->sputc(c).

2. Add a member:
bool operator failed() const throw();

Returns: Returns true if in any prior use of member operator=, the call to sbuf_->sputc() returned traits::eof(), or false otherwise.

3. Eliminate ostreambuf_iterator members equal(), in 24.4.4.2 [lib.ostreambuf.iter.ops], and global operators == and !=, in 24.4.4.3 [lib.ostreambuf.iterator.nonmembers]. No function that takes an iterator can use them anyway, so they only add clutter. (This does not imply any corresponding changes to istreambuf_iterator.)

Requestor: Nathan Myers
Owner: David Dodgson (Iterators)
Emails: lib-3809,3812
Papers: 95-0191/N0791, pre-Tokyo, "Streambuf Iterator Cleanup" Nathan Myers

---

Work Group: Library Clause 24
Issue Number: 24-018
Title: Cleanups in [io]streambuf_iterator
Section: 24.4.3 and 24.4.4
Status: active
Description:
24.4.3 [lib.istreambuf.iterator] and
24.4.4 [lib.ostreambuf.iterator]:

1. The typedefs declared in the streambuf iterators can lead to confusion because the have the same names as global typedefs. While this does not confuse compilers, it confuses readers, and is easily fixed.

2. The description of semantics of istreambuf_iterator member operators is too vague: "Advances the iterator" and "Extract one character" are subject to interpretation.

Proposed Resolution

1. In both 24.4.3 and 24.4.4, change the typedefs "streambuf", "istream", and "ostream" to "streambuf_type", "istream_type", and "ostream_type", respectively, to prevent confusion. In 24.4.3, replace the second and third constructors with:
istreambuf_iterator(istream_type& s) throw();
istreambuf_iterator(streambuf_type* s) throw();
In 24.4.4, replace the first three constructors with:
ostreambuf_iterator(ostream_type& s) throw();
ostreambuf_iterator(streambuf_type& s) throw();
In both 24.4.3 and 24.4.4 change the declaration of sbuf_ to:
streambuf_type* sbuf_; // exposition only

2. In 24.4.3.2 and 24.4.3.3, the descriptions of istreambuf_iterator operators have become unfortunately vague:

- operator*() should be documented to return (specifically) the result of calling (the equivalent of) sbuf_->sgetc().

- operator++() should be documented to:
Effects: sbuf_->sbumpc(); Returns: *this
(Or, if basic_streambuf<>::stossc() gets rehabilitated, then:
Effects: sbuf_->stossc(); Returns: *this)

- operator++(int) should be documented to return
proxy(sbuf_->sbumpc(), sbuf_).

- change the definition of member equal():
  Returns: true if both operatnds are at end-of-stream, or neither is end-of-stream, regardless of what stream they iterate over.

- change the definition of "istream_iterator<..>::proxy" to "istreambuf_iterator<..>::proxy".

Requestor: Nathan Myers
Owner: David Dodgson (Iterators)
Emails: lib-3812
Papers: 95-0191/N0791, pre-Tokyo, "Streambuf Iterator Cleanup" Nathan Myers

---

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

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>.

Proposed Resolution:
Move the stream iterators into a separate header.

Update Table 55 (pg 24-1) to include header <iositer> in 24.4.

Remove #include's for iosfwd, ios, and streambuf from 24.1.6 [lib.iterator.tags] Header <iterator> synopsis and tags for subclause 24.4. Create new header <iositer> in section 24.4 with #include's for iterator, iosfwd, ios, and streambuf and tags for section 24.4 from the <iterator> header.

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

Work Group: Library Clause 24
Issue Number: 24-023
Title: Bad description for istreambuf_iterator constructors
Section: 24.4.3.2 [lib.istreambuf.iterator.cons]
Status: active
Description:
24.4.3:
The header for istreambuf_iterator contains the constructor
istreambuf_iterator(streambuf* s);
but there is no description for this constructor in section 24.4.3.2 [lib.istreambuf.iterator.cons].

The description for constructor istreambuf_iterator(istream) states the effect as constructing 'istream_iterator' not 'istreambuf_iterator'.

Proposed Resolution:
Add the description for constructor istreambuf_iterator(streambuf* s) in section 24.4.3.2 [lib.istreambuf.iterator.cons] with effects of constructs the istreambuf_iterator pointing to s.

Change 'istream_iterator' on line 4 to 'istreambuf_iterator'.

Requestor: Library WG
Owner: David Dodgson (Iterators)
Emails:
Papers:

Work Group: Library Clause 24
Issue Number: 24-024
Title: Operator -> Issues for Iterators
Section: 24.1.3, 24.1.1
Status: active
Description:
24.1.1, 24.1.3 p24-2,4:
Should operator->* be added for iterators?
Section 14.3.3 [temp.opref] specifically allows operator-> to appear in a template where its return type cannot be dereferenced if it is not used. No such guarantee is made for operator->*. If operator->* is desired, the same guarantee should be made.

Does operator-> work correctly for input iterators? (*a can return an rvalue).

Resolution:
Requestor: Library WG
Owner: David Dodgson (Iterators)
Emails: lib-4301
Papers:

Work Group: Library Clause 24
Issue Number: 24-026
Title: Istream Iterator Interactive Input
Section: 24.4.1
Status: active
Description:
24.4.1 p24-22:

Bernd Eggink in lib-4007
> No, it does not. The code in HP implementation is:
>
> class istream_iterator : public input_iterator<T, Distance> {
> friend bool operator==(const istream_iterator<T, Distance>& x,
> const istream_iterator<T, Distance>& y);
> protected:
> istream* stream;
> T value;
> bool end_marker;
> void read() {
> end_marker = (*stream) ? true : false;
> if (end_marker) *stream >> value;
> end_marker = (*stream) ? true : false;
> }
> public:
> istream_iterator() : stream(&cin), end_marker(false) {}
> istream_iterator(istream& s) : stream(&s) { read(); }
> const T& operator*() const { return value; }
> istream_iterator<T, Distance>& operator++() {
> read();
> return *this;

```
>     }
>     istream_iterator<T, Distance> operator++(int)  {
>  istream_iterator<T, Distance> tmp = *this;
>  read();
>  return tmp;
>     }
> };
>
```

BTW, this implementation is practically unusable for interactive input because of the read() in the constructor (which could be easily eliminated
by introducing a bool member which tells whether or not the current element
has been read).

Nathan Myers in lib-4010
I agree that the above change should be made. Who will write up the WP changes? (The delta in the WP would be quite small: I believe it would involve two paragraphs.) This would not break code.

Proposed Resolution:

Changes to the semantics of input iterators accepted at the Tokyo meeting allow the read() in the constructor to be delayed until the iterator is referenced. This leaves us the choice of:

1) Leave as a quality-of-implementation issue
2) Add a bool member which identifies if the charT has been read
3) Mandate that istream iterators shall not perform the read during construction.

Note that the description in para. 1 of 24.4.1 [lib.istream. iterator] must be changed to say that the iterator need not read and store a value of T during construction.

A new bool member should not be needed now that input iterator semantics have been changed.

Requestor: Bernd Eggink
Owner: David Dodgson (Iterators)
Emails: lib-4007,4010
Papers:

Work Group: Library Clause 24
Issue Number: 24-027
Title: Istream Iterator Semantics
Section: 24.4
Status: active
Description:

24.4.3:

24.4.3.5 equal seems at variance with the standard definition. If istreambuf_iterator i = j; Then i == j should be true even if not at end-of-stream.

In general, the semantics of istream_iterator should conform to the semantics of input iterators in general. See issue 22 for a resolution to input iterator semantics. Once input iterator semantics are resolved, the semantics for istream iterators must be examined.

Also, the level of detail specified for istream iterators in the standard must be determined. To what extent should the details of istream iterator be defined? Should specific iostream calls be mandated? Must further explanation of items already defined by input iterator semantics be given. For example, does the 'proxy' class need to be specified or should that be left up to the individual implementation of input iterator?

Resolution: Dependent on issue 22
Requestor: David Dodgson
Owner: David Dodgson (Iterators)
Emails: lib-4065-4069
Papers:

Work Group: Library Clause 24
Issue Number: 24-028
Title: Const Attribute in Iterator Requirements
Section: 24.1
Status: active
Description:

24.1:

The tables in Clause 24 of Iterator Requirements include mutative operations such as ++ and =, but make no mention of constness. We should distinguish which operations require a non-const operand and which can be performed on a const operand. (e.g. is *a allowed on a const iterator?)

Proposed Resolution:

All operations found in the tables can be applied to const operands except: ++a a++ --a a-- a= a+= a-=.

Requestor: Nathan Myers
Owner: David Dodgson (Iterators)
Emails: lib-4172
Papers:

Work Group: Library Clause 24
Issue Number: 24-029
Title: Streambuf Iterator Issues
Section: 24.1.6 and 24.4
Status: active
Description:

24.1.6 header and 24.4 on streambuf iterators:

These issues are raised by P.J. Plauger in N0795:

24.1.6:
Class istreambuf_iterator should be declared with public base class input_iterator. There is then no need to add a special signature for iterator_category (which is missing from the <iterator> synopsis).

24.1.6:
Template operator==(istreambuf_iterator) should not have default template parameters.

24.1.6:
Template operator!=(istreambuf_iterator) is ambiguous in the presence of template operator!=(const T&, const T&). It should be struck.

24.1.6:
Class ostreambuf_iterator should be declared with public base class output_iterator. There is then no need to list a special signature for iterator_category.

24.1.6:
Template operator==(ostreambuf_iterator) and corresponding operator!= are nonsensical and unused. They should be struck.

24.4.3:
istreambuf_iterator should have a member ``bool fail() const'' that returns true if any extractions from the controlled basic_streambuf fail. This is desperately needed by istream to restore its original approved functionality when these iterators are used with facet num_get.

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(basic_istream s) should construct an end-of-stream iterator if s.rdbuf() is null. Otherwise, it should extract an element, as if by calling s->rdbuf()->sgetc(), and save it for future delivery by operator*(). (Lazy input, however, should be permitted.)

24.4.3.2:
istreambuf_iterator(basic_streambuf *) has no description

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

24.4.3.3:
istreambuf_iterator::operator*() should deliver a stored element, or call sgetc() on demand, then store the element. It should *not* extract a character, since this violates the input_iterator protocol.

24.4.3.4:
istreambuf_iterator::operator++() Effects should say that it alters the stored element as if by calling s->snextc(), where s is the stored basic_streambuf pointer.

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.

24.4.3.8:
template operator==(istreambuf_iterator&, istreambuf_iterator&) should have const operands.

24.4.3.8:
template operator!=(istreambuf_iterator&, istreambuf_iterator&) should have const operands. It also is ambiguous in the presence of template<class T> operator!=(T, T) (as are many operators in the library).

24.4.4:
ostreambuf_iterator::equal is silly, since output iterators cannot in general be compared. It should be struck.

24.4.4:
ostreambuf_iterator should remove all references to equal, operator==, and operator!=. Output iterators cannot be compared.

24.4.4:
ostreambuf_iterator should have a member ``bool fail() const'' that returns true if any insertions into the controlled basic_streambuf fail. This is desperately needed by ostream to restore its original approved functionality when these iterators are used with facet num_put.

24.4.4:
ostreambuf_iterator should add the member `bool failed() const', which returns true only if an earlier insertion failed. It is needed by num_put in 22.2.2.2 to communicate insertion failures to inserters in 27.6.1.2. With this change, I believe the following example inserter from basic_ostream satisfies all the exception-handling requirements in the current draft:

```
Mytype& operator<<(long X)
 {iostate stat = goodbit;
 if (opfx())
  {const Myfacet& fac = use_facet<Myfacet>(getloc());
  try {
   if (fac.put(Myiter(rdbuf()), Myiter(0), (ios_base&)*this,
    stat, X).failed()
    stat |= badbit; }
  catch (...) {
   setstate(badbit, Rethrow); }} // added argument
 osfx();
 setstate(stat);
 return (*this); }
```

24.4.4.1:
ostreambuf_iterator::ostreambuf_iterator() produces a useless object. It should be struck.

24.4.4.1:
ostreambuf_iterator;:ostreambuf_iterator(streambuf *) should require that s be not null, or define behavior if it is.

24.4.4.2:
ostreambuf_iterator::equal is not needed and should be struck.

24.4.4.3:
ostreambuf_iterator::operator== is silly, since output iterators cannot in general be compared. It should be struck.

24.4.4.3:
ostreambuf_iterator::operator!= is silly, since output iterators cannot in general be compared. It should be struck.

Resolution: Resolution as suggested in N0795
Requestor: Bill Plauger
Owner: David Dodgson (Iterators)
Emails: lib-4299,4404,4406-4407,4409-4412
Papers: pre-Tokyo N0795

Work Group: Library Clause 24
Issue Number: 24-030
Title: Distance Requirement
Section: 24.2.6
Status: active
Description:
24.2.6 p24-12 [lib.operator.operations]:

24.2.6:
Template function distance should have the requirement that last is reachable from first by incrementing first.

Resolution: As suggested
Requestor: Bill Plauger
Owner: David Dodgson (Iterators)
Emails:
Papers: pre-Tokyo N0795

Work Group: Library Clause 24
Issue Number: 24-032
Title: Insert Iterator Issues
Section: 24.3.2
Status: active
Description:
24.3.2 p24-18 [lib.insert.iterator]:

24.3.2.3:
Template class front_insert_iterator should not have a Returns clause.

24.3.2.5:
insert_iterator::operator++(int) returns a reference to *this, unlike in other classes. Otherwise, the update of iter by operator= gets lost.

24.3.2.6.5:
Declaration for template function inserter is missing second template argument, class Iterator. It is also missing second function argument, of type Iterator.

Resolution:
Requestor: Bill Plauger
Owner: David Dodgson (Iterators)
Emails:
Papers: pre-Tokyo N0795

Work Group: Library Clause 24
Issue Number: 24-033
Title: Iterator Category Defintion
Section: 24.1.6 [lib.iterator.tags]
Status: active
Description:
24.1.6:
Iterator tags could be related by inheritance. Doing so would allow a more generic solution to algorithms which are multiply defined based on iterator category. For example, it might be possible to define to versions of an algorithm, one based on output_iterator and one based on forward_iterator. Iterator categories which inherit from forward_iterator could use the second algorithm. If the categories are inherited, then the based classes should use inheritance.

It may also be desirable to provide a mechanism to indicate whether an iterator is constant or mutable. Different algorithms on iterators could be used if this information was available.

Resolution:
Requestor: Angelika Langer
Owner: David Dodgson (Iterators)
Emails: lib-4305,4308,4312,4315
Papers:

Work Group: Library Clause 24
Issue Number: 24-034
Title: Reverse Iterator Description [Box 107]
Section: 24.3.1 [lib.reverse.iterators]
Status: active
Description:
24.3.1 p24-13 p3:
Box 107 (from Corfield) states that the description for a reverse iterator return type should specify the return type, not a reference. The Reference and Pointer parameters include the appropriate type definitions.

Resolution:
This paragraph appears to be a holdover from before the parameters for the template were reworded. This paragraph should be reworded to conform to the new parameters.

Requestor: Editorial Box
Owner: David Dodgson (Iterators)
Emails:
Papers:

Work Group: Library Clause 24
Issue Number: 24-035
Title: Typos in 26 Sept. 95 Draft
Section: 24.1.6, 24.3.1.4.15
Status: active
Description:

24.1.6 p11:

After paragraph 11 the following title appears:

Header <iterator> synopsislib.iterator.synopsis

<- bold - - - - -><- normal - - - ->

Either the additional wording was added unintentionally or an attempt was made to add a header. Since the synopsis does not belong in the previous section (Iterator tags), a new header should be added. Other clauses seem to have a separate header before the synopsis, perhaps "Iterator classes" would serve?

24.3.1.4.15:

The header for 24.3.1.4.15 [lib.reverse.iter.opsum] states it is "operator==" when it should be "operator+". Operator== has already been defined and the code in this section is for operator+.

Resolution:

Add a new header before the synopsis.

Change the header for 24.3.1.4.15 to operator+.

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

Work Group: Library Clause 24
Issue Number: 24-036
Title: Distance Type for Input Iterators
Section: 24.
Status: active
Description:

24.p?.?:

From Angelika Langer in lib-4407:

>So, why is distance_type() defined for input_iterators?

There is no need for distance_type for input and output iterators.

Resolution:
Requestor: Angelika Langer
Owner: David Dodgson (Iterators)
Emails: lib-4407,4409-4412,4414-4415
Papers:

3. *Resolved Issues*

Work Group: Library Clause 24
Issue Number: 24-006
Title: Relaxing Requirement on Iterator++ Result
Section: 24.4.3
Status: resolved
Description:

24.4.3 p24-23

The return type of operator++ for istreambuf_iterator is listed as 'proxy'. This suggestion is to make the return type an object which is "convertible to const X&" rather than "X&".

Resolution: accepted in Austin
Requestor: Nathan Myers
Owner: David Dodgson (Iterators)
Emails:
Papers: 95-0021/N0621 (Pre-Austin mailing)

Work Group: Library Clause 24
Issue Number: 24-007
Title: Fixing istreambuf_iterator
Section: 24.4.3
Status: resolved
Description:
24.4.3 p24-23:
Proposes the addition to istreambuf_iterator of
inline istreambuf::proxy::operator istreambuf_iterator()
{ return sbuf_; }
to better conform to the Forward Iterator specification.
Resolution: accepted in Austin
Requestor: Nathan Myers
Owner: David Dodgson (Iterators)
Emails:
Papers: 95-0022/N0622 (Pre-Austin mailing)

Work Group: Library Clause 24
Issue Number: 24-022
Title: Input Iterator Semantics
Section: 24.1.1
Status: resolved
Description:
24.1.1 p24-2:
What are the semantics of input iterators in the following:

```
input_iterator i;
cout << *i;       // Object "a"
cout << *i;       // Continues to return object "a"?
 /* This seems to be implied by requirement a == b implies *a == *b.
    Therefore *a == *a should be true.
    This implies the input object is 'saved' in some fashion. */

input_iterator j = i;
cout << *j;       // Object "a"
++i;
cout << *i;       // Object "b"
cout << *j;       // Object "a", "b", or undefined?
 /* Returning "b" implies that all input iterators remain in
    lockstep and all point to the same item.  This is not how
    other iterators work.

    Undefined implies that changing a different iterator can
    affect the value of this iterator, even though no change
    has been made to this iterator.

    Returning "a" is how other iterators work.  It implies that
    the 'saved' object is not destroyed when an input occurs.
    Bill Plauger states that several STL algorithms depend on
    this behaviour.
 */

++j;              // What is the effect on j after i has been
                  // incremented; object "b", "c", or undefined?
```

Nathan Myers proposes a change to semantics of copying in 3943:
Copying an input iterator should invalidate the previous version of the iterator. This ensures that there is only one current version of the iterator usable for ++. It would prevent dereferencing of the copied version of the iterator. [See 3943 for a complete description].

Andy Koenig proposes a simplified scheme in 4114:
Copying an input iterator does not invalidate the previous version. Both may be dereferenced until one is incremented. It is undefined to dereference any copy other than the one incremented.

Summary
Various other messages discuss the relative merits of different semantics. There are three proposed methods.

-- Local Copy
This method requires the iterator to retain a separate copy of the object. Any copy of the iterator has a distinct copy of the object.

After a copy (b) is made of an iterator (a) then both a == b and *a == *b return true.

If a copy is incremented, the value returned by dereferencing a previous copy is well-defined and unchanged.

*a++ is valid and probably implemented by returning a temporary copy of the iterator for operator++.

-- Global Copy
This method allows the iterator to point to one particular copy of the object. Any copy of the iterator may point to the same copy of the object.

After a copy (b) is made of an iterator (a) then both a == b and *a == *b return true.

If a copy is incremented, it is undefined behaviour to dereference a previous copy.

*a++ is valid and probably implemented by returning a temporary instance of a proxy class (although an iterator implemented with local copy semantics will conform to global copy requirements).

-- Unique Copy
This method allows only one valid copy of an iterator to be dereferenced at a time.

After a copy (b) is made of an iterator (a) then a == b returns true but *a == *b is undefined, because *a is undefined.

It is undefined behaviour to increment any iterator other than the one which may be dereferenced.

*a++ is valid is probably implemented by returning a temporary instance of a proxy class.

Proposed Resolution:
Local copy is the status quo. Global copy has been proposed by Andy Koenig, unique copy by Nathan Myers. One specific method must be chosen. Consensus seems to be towards global copy but the issue is controversial.

Resolution:
Resolved as in N0787/95-0187 with changes as proposed by motion 31 as passed in Tokyo.

Requestor: Library WG
Owner: David Dodgson (Iterators)
Emails: lib-3938,3941-3950,3956-3959,4013-4050,4055-4059, 4068-4070,4074,4081,4084,4086-4088,4114-4118, 4122-4127,4132-4138,4141
Papers: 95-0187/N0787 "Input Iterators" pre-Tokyo

Work Group: Library Clause 24
Issue Number: 24-025
Title: Input Iterator Assignment
Sections: [lib.stream.iterators], [lib.istreambuf.iterator],
Status: resolved
Description:
24.1.1 p24-2 [lib.input.iterators]

From Nathan Myers:

This is what I want to say:

```
template <class T> void f(const T&);

// with assignment:
template <class InputIterator, class Pred>
void grab(InputIterator begin, InputIterator end, Pred const& pred)
{
  while ((begin = find(begin, end, pred)) != end)
    f(*begin++);
}
```

But without assignment, I can't. Using modern copy semantics, I can say:

```
// without assignment (using explicit construction)
template <class InputIterator, class Pred>
void grab(InputIterator begin, InputIterator end, Pred const& pred)
{
  while (1) {
    InputIterator t = find(begin, end, pred);
    if (t == end) break;
    f(*t);
    begin.~InputIterator();
    new(&begin) InputIterator(t);
    ++begin;
  }
}
```

But that's worse than ugly. Another alternative that works in this case is to use recursion to model the loop:

```
// without assignment (recursive):
template <class InputIterator, class Pred>
void grab(InputIterator begin, InputIterator end, Pred const& pred)
{
  InputIterator t = find(begin, end, pred);
  if (t != end) {
    f(*t);
    grab(++t, end, pred);
  }
}
```

... but you can't always use recursion this easily, and recursive functions don't expand inline very well. [and this consumes linear space]

I don't much like either of these alternatives. Do you have a better way to do this stuff, or should I just get used to using these methods? Or try to add assignment to the Input Iterator requirements?

Proposed Resolution:

Add the following to table 57-'Input Iterator requirements' and table 58-'Output iterator requirements' in sections 24.1.1 [lib.input.iterators] and 24.1.2 [lib.output.iterators]:

u = a	X&	post: u is a copy of a

Requestor: Nathan Myers
Owner: David Dodgson
Emails: lib-3936,3939-3940,3942-3943,4114,4116-4118
Papers: 95-0187/N0787 "Input Iterators", accepted in Tokyo