Clause 24 (Iterators) Issues List (Rev. 4)

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discussion, *resolved* - resolution accepted but not yet in the working paper, *closed* - working paper updated, or *withdrawn* - issue withdrawn or rejected.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
Revision 3 -	5/23/96	pre-Stockholm	N0915/96-0097
Revision 4 -	9/22/96	pre-Hawaii	N0988/96-0170

2. *Active 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>.

Proposed Resolution:
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.

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

Work Group: Library Clause 24
Issue Number: 24-038
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 implementor 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 of istreambuf_iterator. Additionally, wording similar to the following should be included in 24.1 (possibly in 24.1.6):

> For the purposes of exposition, the return type of operator++(int) in the header <iterator> is specified as a value of that iterator. However, a different type may be used for input and output iterators providing the iterator meets the requirements specified for its category.

An additional editorial consideration may be to put the return types in boldface italic font as are other items for exposition.
Requestor: 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 specifiactions 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.

Resolution:
Requestor: Judy Ward
Owner: David Dodgson (Iterators)
Emails:
Papers:

Work Group: Library Clause 24
Issue Number: 24-043
Title: Distance Type in istreambuf_iterator
Section: 24.5.3 [lib.istreambuf.iterator]
Status: active
Description:
24.5.3 24-22:
During discussions prior to and during the Santa Cruz meeting it was suggested that input iterators do not need a Distance type. At the same time it was suggested that istreambuf_iterator, as an input iterator, did not need a Distance type. The working group decided against removing Distance from input iterators in general, but removing it from istreambuf_iterator was accepted as part of N0845.

Should istreambuf_iterator have a Distance type?

Proposed Resolution:
It would be more consistent for istreambuf_iterator to conform to have a Distance type as other input iterators.
Requestor: David Dodgson
Owner: David Dodgson (Iterators)
Emails:
Papers: N0845, Make Library Member Typedefs Consistent, pre-Santa Cruz

----

Work Group: Library Clause 24
Issue Number: 24-044
Title: Simplification of reverse iterator adaptors
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.

Resolution:
Requestor: Matt Austern, Angelika Langer, Alex Stepanov
Owner: David Dodgson (Iterators)
Emails:
Papers: 96-0092/N0910, "Simplification of reverse iterator adaptors", 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 iterator is destroyed.

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`

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

3. *Resolved Issues*

Work Group: Library Clause 24
Issue Number: 24-032
Title: Insert Iterator Issues
Status: resolved
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: N0833R1 - Proposed Iterators Changes, post Santa Cruz

----

Work Group: Library Clause 24

Issue Number: 24-033
Title: Iterator Category Defintion
Section: 24.1.6 [lib.iterator.tags]
Status: resolved
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: Inheritance in iterator tags accepted in N833R1
accepted in Santa Cruz.
input -> forward -> bidirectional -> random
Requestor: Angelika Langer
Owner: David Dodgson (Iterators)
Emails: lib-4305,4308,4312,4315
Papers: N0833, Proposed Iterators Changes, pre-Santa Cruz

Work Group: Library Clause 24
Issue Number: 24-037
Title: Iterator Traits
Section: 24.
Status: resolved
Description:
24.:

Define the types governing iterators in an iterator_traits class.

```
template <class Iterator>  struct iterator_traits {
   typedef Iterator::distance_type      distance_type;
   typedef Iterator::value_type         value_type;
   typedef Iterator::iterator_category  iterator_category; }
```

The types for any iterator could then be referenced as:

```
iterator_traits<Iter>::distance_type ...;
```

Partial specialization would be used for pointer types:

```
template <class T>  struct iterator_traits<T*> {
   typedef ptrdiff_t  distance_type;
   typedef T  value_type;
   typedef random_access_iterator_tag  iterator_category; }
```

Additionally, the current base classes for iterators would be replaced by:

```
template <class Category, class T, class Distance=ptrdiff_t >
struct iterator {
   typedef Distance distance_type;
   typedef T        value_type;
   typedef Category iterator_category; }
```

which would be used as:

```
class MyIter:public iterator<bidirectional_iterator_tag,
                             double, long>  { ... }
```

Resolution: Accepted in Santa Cruz
Requestor: Bjarne Stroustrup, Alex Stepanov, Matt Austern
Owner: David Dodgson (Iterators)
Emails:
Papers: N847,Bring Back the Obvious Definition of Count, pre-Santa Cruz