WG21/N0833
X3J16/96-0015

Proposed Iterators Changes

David Dodgson
dsd@tr.unisys.com
UNISYS

Introduction

The open issues regarding iterators can be grouped together under several larger subjects. The following groupings discuss the general issues involved and give proposed WP changes. See the "Clause 24 Issues List (Rev. 2)" for a list of the individual issues. The following table gives a cross-reference between the general topic and the individual issues.

Topic	Issue
const in member functions	3, 13, 28
operator->* in iterators	24
distance	30, 36
pre-defined iterators	31,32,34
stream iterators	15, 17, 18, 21, 23, 26, 27, 29
iterator categories	33
operator+ in iterators	12
editorial	35

1. Const in Member Functions

The member functions `base(), operator*(), operator->(),` and `operator[]()` specified in the reverse and reverse_bidirectional iterators should be uniformly const or not const. Originally the const-ness of the functions was not specified. We have agreed (motion 34 in Monterey, see N0740) to make these functions const. There should be a uniform treatment of these member functions.

In editorial boxes 108, 109, and 110, Sean Corfield suggests that a const member function returning a reference to non-const T is wrong. An alternative to what is in the WP is to have separate classes `const_reverse_bidirectional_iterator` and `const_reverse_iterator.` Specification of const in the template parameters of the current templates provides the same basic functionality but additional classes would make this clearer.

A further consideration (see issue 28) is whether the Iterator requirement tables should specify which operations require const. In other words, should we specify which operations must be available for const iterators?

The proposed changes make the changes for the operator functions uniformly const. Any new classes may be separately proposed.

Section	Changes
24.3.1.2.2 [lib.reverse.bidir.iter.conv] through 24.3.1.2.4 [lib.reverse.bidir.iter.opref]	should be const
24.3.1.3 [lib.reverse.iterator]	functions `base, operator*, and operator->` should be const
24.3.1.4.2 [lib.reverse.iter.conv] through 24.3.1.4.4 [lib.reverse.iter.opref]	should be const
24.4.3 [lib.istreambuf.iterator] and 24.4.3.3 [lib.istreambuf.iterator::op*]	operator* should be const

Table 1 - Changes for const member functions

2. Operator->* in Iterators

Motion 32 accepted in Monterey added operator-> for iterators. Since we have decided to allow `operator->` why not allow `operator->*`? (See N0738 for the previous changes).

The table defining forward iterator (24.1.3) would be updated as with `a->m`. There is an outstanding issue asking whether `a->m` should be allowed for input iterators (24.1.1) since an input iterator may return an rvalue. If so, that table should also be updated.

Changes would be required to add the operator function in reverse iterators, 24.3.1. However the new code would differ from the code for operator-> because `operator->` is treated like a unary overloading (see 13.5.6 [over.ref]). `operator->*` is strictly a binary op and would require the type of the second operand to be considered. The change could be done with a member template function:

```
      template< class U >
      U operator->*( U T::* p ) { return &(operator*()).*p; };
```

Although this template works with data members I am unsure what changes would be needed for pointer-to-member functions.

Section	Changes
24.1.1 [lib.input.iterators]	add `a->*m` similarly to `a->m`
24.1.3 [lib.forward.iterators]	add `a->*m` similarly to `a->m`

24.3.1.1 [lib.reverse.bidir.iter] and 24.3.1.3 [lib.reverse.iter]	after operator-> add:
	`template< class U >`
	`U operator->*( U T::* p )`

In 24.3.1.2 [lib.reverse.bidir.iter.ops] and 24.3.1.4 [lib.reverse.iter.ops]	add a section for the description of the function:
	`template< class U >`
	`U operator->*( U T::* p )`
	Effects:
	`return &(operator*()).*p;`

Table 2A - Changes for operator->*

If the proposal to change `operator->*` in N0831/96-0013 passes, the changes in the iterators will be simplified.

Section	Changes
24.1.1 [lib.input.iterators]	add `a->*m` similarly to `a->m`
24.1.3 [lib.forward.iterators]	add `a->*m` similarly to `a->m`

Section	Changes
24.3.1.1 [lib.reverse.bidir.iter] and 24.3.1.3 [lib.reverse.iter]	after operator-> add: `Pointer operator->*() const`

In 24.3.1.2 [lib.reverse.bidir.iter.ops] and 24.3.1.4 [lib.reverse.iter.ops]	add a section for the description of the function:
	`Pointer operator->*() const`
	Effects:
	`return &(operator*());`

Table 2B - Changes for operator->* if N0831 passes

3. Distance in Iterators

The `distance` function in 24.2.6 does not have a requirement constraining `last` to be reachable from `first`. (Issue 30)

Section	Changes
24.2.6 [lib.iterator.operations]	Add the following for `distance`: **Requires:** `last` must be reachable from `first`

Table 3A - Changes for iterator operations

Input iterators are written to require a distance type. Since input iterators do not represent a sequence there is no distance between instances. The distance could be used to count the number of increments which have been done on an iterator, but it is not a reproducable distance. Removing the distance type from input iterators would make this similarity with output iterators clearer. If removed, should the `advance` function still be defined to work with input iterators (and output iterators)?

The proposed changes remove the distance class for input iterators.

Section	Changes
24.1 [lib.iterator.requirements] paragraph 1	The last sentence should change “for which equality is defined” to “which defines a reproducible sequence”

Header <iterator> synopsis	Change to:
	`template <class T> struct input_iterator {};`
	`template <class T> input_iterator_tag iterator_category (const input_iterator<T>&);`
	`template <class T> t* value_type (const input_iterator<T>&);`
	Remove `distance_type` for input iterator;
	Change `InputIterator` to `ForwardIterator` for `advance` and `distance;`
	Change `istream_iterator` to remove `Distance`

Section	Changes
24.2.2 [lib.basic.iterators], 24.2.3 [lib.iterator.category], and 24.2.5 [lib.distance.type]	Remove the `Distance` template parameter for `input_iterator`
24.2.6 [lib.iterator.operations	Replace the references to `InputIterator` with `ForwardIterator`
24.4.1 [lib.istream.iterator]	Remove all uses of the `Distance` parameter

Table 3B - Changes to remove distance from input iterators

4. Pre-Defined Iterators

The following are miscellaneous small changes in the reverse and input iterators.

Section	Changes
24.3.1 [lib.reverse.iterators]	Update paragraph 3 as suggested in Box 107
24.3.2.3 [lib.front.insert.iterator]	The template class should not have a Returns clause
24.3.2.6.1 [lib.insert.iter.cons]	Type `Iterator` should be `typename Container::iterator`
24.3.2.6.5 [lib.inserter]	The template should have a second parameter, `Inserter`, (see 24.3.2.5) and the function should have a second parameter: `Inserter i`.

Table 4 - Changes to pre-defined iterators

5. Stream Iterators

There are a substantial number of issues regarding the stream iterators in 24.4. Some changes to the input iterators are needed to conform to the input iterator resolution passed in Tokyo. There have been a number of proposed changes to the iterators, some of which are conflicting. The following proposals are an attempt to provide a consistent structure.

Character-Orientation of Istream and Ostream

Editorial box 111 states that the `istream_iterator` and `ostream_iterator` classes are defined only for the `char`-oriented streams. They should be usable by any stream.

Section	Changes
Header <iterator> synopsis	Add template parameters as below for `istream_iterator` and `ostream_iterator`

24.4.1 [lib.istream.iterator]	Add template parameters:
	`charT, class traits = ios_traits<charT>`
	Add:
	`typedef basic_istream<charT,traits> istream_type;`

Section	Changes
	Use `istream_type` in the appropriate constructor

24.4.2 [lib.ostream.iterator]	Add template parameters:
	`charT, class traits = ios_traits<charT>`
	
	Add:
	`typedef basic_ostream<charT,traits> ostream_type;`
	Use `ostream_type` in the appropriate constructors
	
	Change 2 parameter constructor to use:
	`const charT* delimiter`

Table 5A - Changes for character orientation

Error Reporting

The streambuf iterators should be able to report errors directly. One possibility is to throw an exception. However, this is not normally done for I/O. Better is to have a bool member function which reports if the operation failed.

Section	Changes
24.4.3 [lib.istreambuf.iterator] and 24.4.4 [lib.ostreambuf.iterator]	Add the member function: `bool fail() const;`
Add to 24.4.3	**istreambuf_iterator::fail** `bool fail() const;` **Returns:** true if an operation using the iterator has failed; otherwise, false
Add to 24.4.4	**ostreambuf_iterator::fail** `bool fail() const;` **Returns:** true if an operation using the iterator has failed; otherwise, false

Table 5B - Changes for error reporting

Small Changes

There are several small changes required to maintain consistency and correct small errors.

Section	Changes
24.4.3 [lib.istreambuf.iterator]	Publicly derive from `input_iterator<charT>`
24.4.3 [lib.istreambuf.iterator] and 24.4.4 [lib.ostreambuf.iterator	Change the typedefs `streambuf`, `istream`, and `ostream` to `streambuf_type`, `istream_type`, and `ostream_type`.
24.4.3.1 [lib.istreambuf.iterator::proxy]	Change `istream_iterator` on line 3 to `istreambuf_iterator`
24.4.3.2 [lib.istreambuf.iterator.cons]	Line 4 should read: Constructs the `istreambuf_iterator` pointing to the `basic_streambuf` object `*(s.rdbuf())` if not null; otherwise the end-of-

Section	Changes
	stream iterator. Add: `istreambuf_iterator( basic_streambuf<charT, traits> s);` **Effects:** Constructs the `istreambuf_iterator` pointing to the `basic_streambuf` object `s.` An end-of-stream iterator is constructed if `s` is null.
24.4.3.6 [lib.iterator.category.i]	Remove this section.
Header <iterator> synopsis and 24.4.3.7 [lib.istreambuf.iterator::op==]	Template operator== should not have default template parameters and the function parameters should be const
Header <iterator> synopsis and 24.4.3.8 [lib.istreambuf.iterator::op!=]	This operator is ambiguous with operator!=(const T&, const T&) and should be removed.
24.4.4 [lib.ostreambuf.iterator]	Publicly derive from `output_iterator`
Header <iterator> synopsis, 24.4.4 [lib.ostreambuf.iterator], 24.4.4.2 [lib.ostreambuf.iter.ops], and 24.4.4.3 [lib.ostreambuf.iterator.nonmembers]	Member function `equal, operator==` and `operator !=` are not used for output iterators and should be removed.
24.4.4 [lib.ostreambuf.iterator] and 24.4.4.1 [lib.ostreambuf.iter.cons]	Remove constructor `ostreambuf_iterator()`
24.4.4.4.1 [lib.ostreambuf.iter.cons]	The constructor for `streambuf` should add: **Requires:** *`s`* shall not be null
24.4.3.3 [lib.istreambuf.iterator::op*]	Replace the word "Extract" with "Returns" and add "as if calling ***sbuf_****->sgetc()*".
24.4.3.4 [lib.istreambuf.iterator::op++]	Add "as if calling ***sbuf_****->snextc()*".
24.4.3.5 [lib.istreambuf.iterator::equal]	**Returns:** `true` if both iterators are at end-of-stream, or if neither is at end-of-stream, regardless of what stream they iterate over.

Table 5C - Small changes

Remove `proxy` class

A streambuf iterator may be constructed which conforms to input iterator semantics but does not require a proxy class. It should not be mandatory that a proxy class be used.

Section	Changes
24.4.3 [lib.istreambuf.iterator] and 24.4.3.2 [lib.istreambuf.iterator.cons]	Remove constructor for class proxy

24.4.3 [lib.istreambuf.iterator] and 24.4.3.4 [lib.istreambuf.iterator::op++]	Change return type of operator++(int) to `istreambuf_iterator<charT,traits>`
	Change to:
	Effects: `{`
	`istreambuf_iterator<charT,traits> tmp = *this;`
	`++(*this);`

Section	Changes
	`return tmp;  }`
24.4.3.1 [lib.istreambuf.iterator::proxy]	Remove this section

Table 5D - Changes to remove proxy class

Remove stream iterators from `<iterators>`

The stream iterators require the inclusion of the headers `iosfwd, ios,` and `streambuf`. This is a large amount of material. Any time the `iterators` header is used to define an iterator will bring in the I/O headers. It should be possible, even desirable, to use the `iterators` header whenever a new iterator is defined. It is, however, almost certain that the I/O headers will be needed when using the stream iterators. Therefore the stream iterators could be moved into the I/O headers in clause 27.

Section	Changes
Header <iterator> synopsis	Remove inclusion of `iosfwd, ios,` and `streambuf` Move definition of `istream_iterator` to `<istream>`. Move definition of `ostream_iterator` to `<ostream>`. Move definitions of `istreambuf_iterator` and `ostreambuf_iterator` to `<streambuf>.`
24.4.1 [lib.istream.iterator]	Move to 27.6.1 [lib.input.streams]
24.4.2 [lib.ostream.iterator]	Move to 27.6.2 [lib.output.streams]
24.4.3 [lib.istreambuf.iterator] and 24.4.4 [lib.ostreambuf.iterator]	Move to 27.5 [lib.stream.buffers]

27.2 [lib.iostream.forward]	Include:
	`template<class charT> class istreambuf_iterator;`
	`template <class charT> class ostreambuf_iterator;`
	`template <class T, class charT> class istream_iterator;`
	`template <class T, class charT> class ostream_iterator;`

27.5 [lib.stream.buffers] and 27.6 [lib.iostream.format]	`#include <iterator>`

Table 5E - Changes to move stream iterators

6. Iterator Categories

The various iterator categories as currently defined are distinct. However, a random access iterator is a bidirectional iterator, and a bidirectional iterator is a forward iterator. The iterator tags and base classes for these iterators could be related through inheritance. This could make the definition of algorithms easier. For example, an algorithm that tested for an iterator category of forward could be passed a random access iterator without problem. It may also be desirable to provide a mechanism to indicate whether an iterator is constant or mutable (see issue 33). No such mechanism is proposed here.

Section	Changes
24.2.1 [lib.std.iterator.tags]	`bidirectional_iterator_tag` should inherit from `forward_iterator_tag` and `random_access_iterator_tag` should inherit from `bidirectional_iterator_tag`
24.2.2 [lib.basic.iterators]	`bidirectional_iterator` should inherit from `forward_iterator` and `random_access_iterator` should inherit from `bidirectional_iterator`

Table 6 - Changes for iterator categories

7. Addition and Subtraction in Iterators

Forward iterators use increment to move through the sequence. It is possible to define operator+ to perform a sequence of increments to move through the sequence (i.e. use the `advance` function). It is also possible to use operator- in bidirectional iterators to move backward through the sequence. Defining these operators would make it simpler to update operators to move through the sequence. It would, however, remove the assurance that an operation on an iterator is performed in constant time.

Section	Changes
24.1 [lib.iterator.requirements] para. 8	Change the first sentence to include, “except addition operators for forward iterator and addition and subtraction operators for bidirectional iterator”.

24.1.3 [lib.forward.iterators]	r += n X& { advance(r,n); return r; }
	a + n X { X tmp = a; return tmp += n; }
	n + a a + n == n + a.
	a [n] Convertible to T *(a + n)

24.1.4 [lib.bidirectional.iterators]	r -= n X& return r += -n;
	a - n X { X tmp = a; return tmp -= n;}

24.1.5 [lib.random.access.iterators]	Remove first four rows
24.2.6 [lib.iterator.operations]	Change the first sentence to, “The library provides two template functions `advance` and `distance.″`

Table 7 - Changes for addition and subtraction

8. Editorial Changes

These are typographical errors or simple editorial changes.

Section	Changes
24.1.6 [lib.iterator.tags] para. 11	The synopsis should not be part of this section. It should be a separate section.
24.3.1.4.15 [lib.reverse.iter.opsum]	The heading should be `operator+=`.

Table 8 - Editorial changes