

Doc. no. J16/00-0047
WG21 N1270
Date: 26 Oct 2000
Project: Programming Language C++
Reply to: Matt Austern <austern@research.att.com>

C++ Standard Library Closed Issues List (Revision 16)

Reference ISO/IEC IS 14882:1998(E)

Also see:

- [Table of Contents](#) including both active and closed issues.
- [Index by Section](#) including both active and closed issues.
- [Index by Status](#) including both active and closed issues.
- [Library Active Issues List](#)
- [Library Defect Report List](#)

This document contains only library issues which have been closed by the Library Working Group as duplicates or not defects. That is, issues which have a status of [Dup](#) or [NAD](#). See "[C++ Standard Library Active Issues List](#)" for active issues and more information. See "[C++ Standard Library Defect Report List](#)" for issues considered defects. The introductory material in that document also applies to this document.

Revision History

- R16: post-Toronto mailing. Reflects committee actions taken in Toronto.
- R15: pre-Toronto mailing. Formatting changes only; no substantive changes.
- R14: post-Tokyo II mailing; reflects committee actions taken in Tokyo. (00-0020R1/N1243)
- R13: Unchanged from R12.
- R12: Add further rationale to issue [178](#).

Closed Issues

2. Auto_ptr conversions effects incorrect

Section: 20.4.5.3 [lib.auto_ptr.conv](#) **Status:** [NAD](#) **Submitter:** Nathan Myers **Date:** 4 Dec 97

Paragraph 1 in "Effects", says "Calls p->release()" where it clearly must be "Calls p.release()". (As it is, it seems to require using `auto_ptr<X>::operator->` to refer to `X::release`, assuming that exists.)

Original proposed resolution:

Change [lib.auto_ptr.conv](#) paragraph 1 Effects from "Calls p->release()" to "Calls p.release()".

Proposed resolution:

Not a defect: the proposed change is already found in the standard. [Originally classified as a defect, later reclassified.]

4. `basic_string` `size_type` and `difference_type` should be implementation defined

Section: 21.3 [lib.basic.string](#) **Status:** [NAD](#) **Submitter:** Beman Dawes **Date:** 16 Nov 97

In Morristown we changed the `size_type` and `difference_type` typedefs for all the other containers to implementation defined with a reference to [lib.container.requirements](#). This should probably also have been done for strings.

Proposed Resolution:

Not a defect. [Originally classified as a defect, later reclassified. See the rationale.]

Rationale:

`basic_string`, unlike the other standard library template containers, is severely constrained by its use of `char_traits`. Those types are dictated by the traits class, and are far from implementation defined.

6. File position not an offset unimplementable

Section: 27.4.3 [lib.fpos](#) **Status:** [NAD](#) **Submitter:** Matt Austern **Date:** 15 Dec 97

Table 88, in I/O, is too strict; it's unimplementable on systems where a file position isn't just an offset. It also never says just what `fpos<>` is really supposed to be. [Here's my summary, *which Jerry agrees is more or less accurate*. "I think I now know what the class really is, at this point: it's a magic cookie that encapsulates an `mbstate_t` and a file position (possibly represented as an `fpos_t`), it has syntactic support for pointer-like arithmetic, and implementors are required to have real, not just syntactic, support for arithmetic." This isn't standardese, of course.]

Rationale:

Not a defect. The LWG believes that the Standard is already clear, and that the above summary is what the Standard in effect says.

10. `codecvt<>::do_unclear`

Section: 22.2.1.5.2 [lib.locale.codecvt.virtuals](#) **Status:** [Dup](#) **Submitter:** Matt Austern **Date:** 14 Jan 98

Section 22.2.1.5.2 says that `codecvt<>::do_in` and `do_out` should return the value `noconv` if "no conversion was needed". However, I don't see anything anywhere that defines what it means for a conversion to be needed or not needed. I can think of several circumstances where one might plausibly think that a conversion is not "needed", but I don't know which one is intended here.

Rationale:

Duplicate. See [issue 19](#).

12. Way objects hold allocators unclear

Section: 20.1.5 [lib.allocator.requirements](#) **Status:** [NAD](#) **Submitter:** Angelika Langer **Date:** 23 Feb 98

I couldn't find a statement in the standard saying whether the allocator object held by a container is held as a copy of the constructor argument or whether a pointer of reference is maintained internal. There is an according statement for compare objects and how they are maintained by the associative containers, but I couldn't find anything regarding allocators.

Did I overlook it? Is it an open issue or known defect? Or is it deliberately left unspecified?

Rationale:

Not a defect. The LWG believes that the Standard is already clear. See 23.1 paragraph 8 [[lib.container.requirements](#)].

43. Locale table correction

Section: 22.2.1.5.2 [lib.locale.codecvt.virtuals](#) **Status:** [Dup](#) **Submitter:** Brendan Kehoe **Date:** 1 Jun 98

Rationale:

Duplicate. See [issue 33](#).

45. Stringstreams read/write pointers initial position unclear

Section: 27.7.3 [lib.ostringstream](#) **Status:** [NAD](#) **Submitter:** Matthias Mueller **Date:** 27 May 98

In a comp.lang.c++.moderated *Matthias Mueller* wrote:

"We are not sure how to interpret the CD2 (see [[lib.iostream.forward](#)], [[lib.ostringstream.cons](#)], [[lib.stringbuf.cons](#)]) with respect to the question as to what the correct initial positions of the write and read pointers of a stringstream should be."

"Is it the same to output two strings or to initialize the stringstream with the first and to output the second ?"

PJ Plauger, Bjarne Stroustrup, Randy Smithey, Sean Corfield, and Jerry Schwarz have all offered opinions; see reflector messages lib-6518, 6519, 6520, 6521, 6523, 6524.

Rationale:

The LWG believes the Standard is correct as written. The behavior of stringstreams is consistent with fstreams, and there is a constructor which can be used to obtain the desired effect. This behavior is known to be different from strstreams.

58. Extracting a char from a wide-oriented stream

Section: 27.6.1.2.3 [lib.istream::extractors](#) **Status:** [NAD](#) **Submitter:** Matt Austern **Date:** 1 Jul 98

27.6.1.2.3 has member functions for extraction of signed char and unsigned char, both singly and as strings. However, it doesn't say what it means to extract a char from a `basic_streambuf<charT, Traits>`.

`basic_streambuf`, after all, has no members to extract a char, so `basic_istream` must somehow convert from `charT` to signed char or unsigned char. The standard doesn't say how it is to perform that conversion.

Rationale:

The Standard is correct as written. There is no such extractor and this is the intent of the LWG..

65. Underspecification of `strstreambuf::seekoff`

Section: D.7.1.3 [depr.strstreambuf.virtuals](#) **Status:** [NAD](#) **Submitter:** Matt Austern **Date:** 18 Aug 98

The standard says how this member function affects the current stream position. (`gptr` or `pptr`) However, it does not say how this member function affects the beginning and end of the get/put area.

This is an issue when `seekoff` is used to position the get pointer beyond the end of the current read area. (Which is legal. This is implicit in the definition of *seekhigh* in D.7.1, paragraph 4.)

Rationale:

The LWG agrees that `seekoff()` is underspecified, but does not wish to invest effort in this deprecated feature.

67. `setw` useless for strings

Section: 21.3.7.9 [lib.string.io](#) **Status:** [Dup](#) **Submitter:** Steve Clamage **Date:** 9 Jul 98

In a `comp.std.c++` posting *Michel Michaud* wrote: What should be output by :

```
string text("Hello");
cout << '[' << setw(10) << right << text << '];'
```

Shouldn't it be:

```
[      Hello]
```

Another person replied: Actually, according to the FDIS, the width of the field should be the minimum of width and the length of the string, so the output shouldn't have any padding. I think that this is a typo, however, and that what is wanted is the maximum of the two. (As written, `setw` is useless for strings. If that had been the intent, one wouldn't expect them to have mentioned using its value.)

It's worth pointing out that this is a recent correction anyway; IIRC, earlier versions of the draft forgot to mention formatting parameters what soever.

Rationale:

Duplicate. See [issue 25](#).

72. Do_convert phantom member function

Section: 22.2.1.5 [lib.locale.codecvt](#) **Status:** [Dup](#) **Submitter:** Nathan Myers **Date:** 24 Aug 98

In 22.2.1.5 par 3 [lib.locale.codecvt](#), and in 22.2.1.5.2 par 8 [lib.locale.codecvt.virtuals](#), a nonexistent member function "do_convert" is mentioned. This member was replaced with "do_in" and "do_out", the proper referents in the contexts above.

Proposed Resolution:

Duplicate: see [issue 24](#).

73. is_open should be const

Section: 27.8.1 [lib.file.streams](#) **Status:** [NAD](#) **Submitter:** Matt Austern **Date:** 27 Aug 98

Classes `basic_ifstream`, `basic_ofstream`, and `basic_fstream` all have a member function `is_open`. It should be a `const` member function, since it does nothing but call one of `basic_filebuf`'s `const` member functions.

Rationale:

Not a defect. This is a deliberate feature; `const` streams would be meaningless.

77. Valarray operator[] const returning value

Section: 26.3.2.3 [lib.valarray.access](#) **Status:** [NAD Future](#) **Submitter:** Levente Farkas **Date:** 9 Sep 98

valarray:

```
T operator[] (size_t) const;
```

why not

```
const T& operator[] (size_t) const;
```

as in vector ???

One can't copy even from a `const` valarray eg:

```
memcpy(ptr, &v[0], v.size() * sizeof(double));
```

[I] find this bug in valarray is very difficult.

Rationale:

The LWG believes that the interface was deliberately designed that way. That is what valarray was designed to do; that's where the "value array" name comes from. LWG members further comment that "we don't want valarray to be a full STL container." 26.3.2.3 [lib.valarray.access](#) specifies properties that indicate "an absence of aliasing" for non-constant arrays; this allows optimizations, including special hardware optimizations, that are not otherwise possible.

81. Wrong declaration of slice operations

Section: 26.3.5 [lib.template.slice.array](#), 26.3.7 [lib.template.gslicing.array](#), 26.3.8, 26.3.9 **Status:** [NAD](#) **Submitter:** Nico Josuttis **Date:** 29 Sep 98

Isn't the definition of copy constructor and assignment operators wrong? Instead of

```
slice_array(const slice_array&);
slice_array& operator=(const slice_array&);
```

IMHO they have to be

```
slice_array(const slice_array<T>&);
slice_array& operator=(const slice_array<T>&);
```

Same for `gslicing_array`.

Rationale:

Not a defect. The Standard is correct as written.

82. Missing constant for set elements

Section: 23.1.2 [lib.associative.reqmts](#) **Status:** [NAD](#) **Submitter:** Nico Josuttis **Date:** 29 Sep 98

Paragraph 5 specifies:

For set and multiset the value type is the same as the key type. For map and multimap it is equal to `pair<const Key, T>`.

Strictly speaking, this is not correct because for set and multiset the value type is the same as the **constant** key type.

Rationale:

Not a defect. The Standard is correct as written; it uses a different mechanism (`const &`) for `set` and `multiset`. See issue [103](#) for a related issue.

84. Ambiguity with `string::insert()`

Section: 21.3.5 [lib.string.modifiers](#) **Status:** [NAD Future](#) **Submitter:** Nico Josuttis **Date:** 29 Sep 98

If I try

```
s.insert(0,1,' ');
```

I get an nasty ambiguity. It might be


```
s.insert((size_type)0,(size_type)1,(charT)' ');
```

which inserts 1 space character at position 0, or

```
s.insert((char*)0,(size_type)1,(charT)' ')
```

which inserts 1 space character at iterator/address 0 (bingo!), or

```
s.insert((char*)0, (InputIterator)1, (InputIterator)' ')
```

which normally inserts characters from iterator 1 to iterator '. But according to 23.1.1.9 (the "do the right thing" fix) it is equivalent to the second. However, it is still ambiguous, because of course I mean the first!

Rationale:

Not a defect. The LWG believes this is a "genetic misfortune" inherent in the design of string and thus not a defect in the Standard as such .

85. String char types

Section: 21 [lib.strings](#) **Status:** [NAD](#) **Submitter:** Nico Josuttis **Date:** 29 Sep 98

The standard seems not to require that charT is equivalent to traits::char_type. So, what happens if charT is not equivalent to traits::char_type ?

Rationale:

There is already wording in 21.1 paragraph 3 ([lib.char.traits](#)) that requires them to be the same.

87. Error in description of string::compare()

Section: 21.3.6.8 [lib.string::compare](#) **Status:** [Dup](#) **Submitter:** Nico Josuttis **Date:** 29 Sep 98

The following compare() description is obviously a bug:

```
int compare(size_type pos, size_type n1,
            charT *s, size_type n2 = npos) const;
```

because without passing n2 it should compare up to the end of the string instead of comparing npos characters (which throws an exception)

Rationale:

Duplicate; see [issue 5](#).

88. Inconsistency between string::insert() and string::append()

Section: 21.3.5.4 [lib.string::insert](#), 21.3.5.2 [lib.string::append](#) **Status:** [NAD Future](#) **Submitter:** Nico Josuttis **Date:** 29 Sep 98

Why does

```
template<class InputIterator>
    basic_string& append(InputIterator first, InputIterator last);
```

return a string, while

```
template<class InputIterator>
    void insert(iterator p, InputIterator first, InputIterator last);
```

returns nothing ?

Rationale:

The LWG believes this inconsistency is not sufficiently serious to constitute a defect.

89. Missing throw specification for string::insert() and string::replace()

Section: 21.3.5.4 [lib.string::insert](#), 21.3.5.6 [lib.string::replace](#) **Status:** [Dup](#) **Submitter:** Nico Josuttis **Date:** 29 Sep 1998

All insert() and replace() members for strings with an iterator as first argument lack a throw specification. The throw specification should probably be: length_error if size exceeds maximum.

Rationale:

Considered a duplicate because it will be solved by the resolution of [issue 83](#).

93. Incomplete Valarray Subset Definitions

Section: 26.3 [lib.numarray](#) **Status:** [NAD Future](#) **Submitter:** Nico Josuttis **Date:** 29 Sep 1998

You can easily create subsets, but you can't easily combine them with other subsets. Unfortunately, you almost always need an explicit type conversion to valarray. This is because the standard does not specify that valarray subsets provide the same operations as valarrays.

For example, to multiply two subsets and assign the result to a third subset, you can't write the following:

```
va[slice(0,4,3)] = va[slice(1,4,3)] * va[slice(2,4,3)];
```

Instead, you have to code as follows:

```
va[slice(0,4,3)] = static_cast<valarray<double>> >(va[slice(1,4,3)]) *
    static_cast<valarray<double>> >(va[slice(2,4,3)]);
```

This is tedious and error-prone. Even worse, it costs performance because each cast creates a temporary object, which could be avoided without the cast.

Proposed resolution:

Extend all valarray subset types so that they offer all valarray operations.

Rationale:

This is not a defect in the Standard; it is a request for an extension.

95. Members added by the implementation

Section: 17.4.4.4 [lib.member.functions](#) **Status:** [NAD](#) **Submitter:** AFNOR **Date:** 7 Oct 98

In 17.3.4.4/2 vs 17.3.4.7/0 there is a hole; an implementation could add virtual members a base class and break user derived classes.

Example:

```
// implementation code:
struct _Base { // _Base is in the implementer namespace
    virtual void foo ();
};
class vector : _Base // deriving from a class is allowed
{ ... };

// user code:
class vector_checking : public vector
{
    void foo (); // don't want to override _Base::foo () as the
                // user doesn't know about _Base::foo ()
};
```

Proposed Resolution:

Clarify the wording to make the example illegal.

Rationale:

This is not a defect in the Standard. The example is already illegal. See 17.4.4.4 [lib.member.functions](#) paragraph 2.

97. Insert inconsistent definition

Section: 23 [lib.containers](#) **Status:** [NAD Future](#) **Submitter:** AFNOR **Date:** 7 Oct 98

`insert(iterator, const value_type&)` is defined both on sequences and on set, with unrelated semantics: `insert` here (in sequences), and `insert with hint` (in associative containers). They should have different names (B.S. says: do not abuse overloading).

Rationale:

This is not a defect in the Standard. It is a genetic misfortune of the design, for better or for worse.

99. Reverse_iterator comparisons completely wrong

Section: 24.4.1.3.13 [lib.reverse.iter.op<](#), etc. **Status:** [NAD](#) **Submitter:** AFNOR **Date:** 7 Oct 98

The <, >, <=, >= comparison operator are wrong: they return the opposite of what they should.

Note: same problem in CD2, these were not even defined in CD1
SGI STL code is correct; this problem is known since the Morristown meeting but there it was too late

Rationale:

This is not a defect in the Standard. A careful reading shows the Standard is correct as written. A review of several implementations show that they implement exactly what the Standard says.

100. Insert iterators/ostream_iterators overconstrained

Section: 24.4.2 [lib.insert.iterators](#), 24.5.4 [lib.ostreambuf.iterator](#) **Status:** [NAD](#) **Submitter:** AFNOR **Date:** 7 Oct 98

Overspecified For an insert iterator it, the expression *it is required to return a reference to it. This is a simple possible implementation, but as the SGI STL documentation says, not the only one, and the user should not assume that this is the case.

Rationale:

The LWG believes this causes no harm and is not a defect in the standard. The only example anyone could come up with caused some incorrect code to work, rather than the other way around.

101. No way to free storage for vector and deque

Section: 23.2.4 [lib.vector](#), 23.2.1 [lib.deque](#) **Status:** [NAD](#) **Submitter:** AFNOR **Date:** 7 Oct 98

Reserve can not free storage, unlike string::reserve

Rationale:

This is not a defect in the Standard. The LWG has considered this issue in the past and sees no need to change the Standard. Deque has no reserve() member function. For vector, shrink-to-fit can be expressed in a single line of code (where v is vector<T>):

```
vector<T>(v).swap(v); // shrink-to-fit v
```

104. Description of basic_string::operator[] is unclear

Section: 21.3.4 [lib.string.access](#) **Status:** [NAD](#) **Submitter:** AFNOR **Date:** 7 Oct 98

It is not clear that undefined behavior applies when `pos == size()` for the non-const version.

Proposed Resolution:

Rewrite as: Otherwise, if `pos > size()` or `pos == size()` and the non-const version is used, then the behavior is undefined.

Rationale:

The Standard is correct. The proposed resolution already appears in the Standard.

105. `fstream` ctors argument types desired

Section: 27.8 [lib.file.streams](#) **Status:** [NAD Future](#) **Submitter:** AFNOR **Date:** 7 Oct 98

`fstream` ctors take a `const char*` instead of `string`.
`fstream` ctors can't take `wchar_t`

An extension to add a `const wchar_t*` to `fstream` would make the implementation non conforming.

Rationale:

This is not a defect in the Standard. It might be an interesting extension for the next Standard.

107. `Valarray` constructor is strange

Section: 26.3.2 [lib.template.valarray](#) **Status:** [NAD](#) **Submitter:** AFNOR **Date:** 7 Oct 98

The order of the arguments is `(elem, size)` instead of the normal `(size, elem)` in the rest of the library. Since `elem` often has an integral or floating point type, both types are convertible to each other and reversing them leads to a well formed program.

The suggested resolution was:

Inverting the arguments could silently break programs. Introduce the two signatures `(const T&, size_t)` and `(size_t, const T&)`, but make the one we do not want private so errors result in a diagnosed access violation. This technique can also be applied to STL containers.

Rationale:

The LWG believes that while the order of arguments is unfortunate, it does not constitute a defect in the standard. The LWG believes that the proposed solution will not work for `valarray<size_t>` and perhaps other cases.

113. Missing/extra `iostream` sync semantics

Section: 27.6.1.1 [lib.istream](#), 27.6.1.3 [lib.istream.unformatted](#), para 36 **Status:** [NAD](#) **Submitter:** Steve Clamage **Date:** 13 Oct 98

In 27.6.1.1, class `basic_istream` has a member function `sync`, described in 27.6.1.3, paragraph 36.

Following the chain of definitions, I find that the various `sync` functions have defined semantics for output streams, but no semantics for input streams. On the other hand, `basic_ostream` has no `sync` function.

The `sync` function should at minimum be added to `basic_ostream`, for internal consistency.

A larger question is whether `sync` should have assigned semantics for input streams.

Classic iostreams said `streambuf::sync` flushes pending output and attempts to return unread input characters to the source. It is a protected member function. The `filebuf` version (which is public) has that behavior (it backs up the read pointer). Class `strstreambuf` does not override `streambuf::sync`, and so `sync` can't be called on a `strstream`.

If we can add corresponding semantics to the various `sync` functions, we should. If not, we should remove `sync` from `basic_istream`.

Rationale:

A `sync` function is not needed in `basic_ostream` because the `flush` function provides the desired functionality.

As for the other points, the LWG finds the standard correct as written.

116. `bitset` cannot be constructed with a `const char*`

Section: 23.3.5 [lib.template.bitset](#) **Status:** [NAD Future](#) **Submitter:** Judy Ward **Date:** 6 Nov 1998

The following code does not compile *with the EDG compiler*:

```
#include <bitset>
using namespace std;
bitset<32> b( "11111111" );
```

If you cast the ctor argument to a string, i.e.:

```
bitset<32> b( string( "11111111" ) );
```

then it will compile. The reason is that `bitset` has the following templated constructor:

```
template <class charT, class traits, class Allocator>
explicit bitset (const basic_string<charT, traits, Allocator>& str,
...);
```

According to the compiler vendor, *Steve Adamcyk at EDG*, the user cannot pass this template constructor a `const char*` and expect a conversion to `basic_string`. The reason is "When you have a template constructor, it can get used in contexts where type deduction can be done. Type deduction basically comes up with exact matches, not ones involving conversions."

I don't think the intention when this constructor became templated was for construction from a `const char*` to no longer work.

Proposed Resolution:

Add to 23.3.5 [lib.template.bitset](#) a bitset constructor declaration

```
explicit bitset(const char*);
```

and in Section 23.3.5.1 [lib.bitset.cons](#) add:

```
explicit bitset(const char* str);
```

Effects:

```
Calls bitset((string) str, 0, string::npos);
```

Rationale:

Although the problem is real, the standard is designed that way so it is not a defect. Education is the immediate workaround. A future standard may wish to consider the Proposed Resolution as an extension.

121. Detailed definition for `ctype<wchar_t>` specialization missing

Section: 22.1.1.1.1 [lib.locale.category](#) **Status:** [NAD](#) **Submitter:** Judy Ward **Date:** 15 Dec 1998

Section 22.1.1.1.1 has the following listed in Table 51: `ctype<char>`, `ctype<wchar_t>`.

Also Section 22.2.1.1 [lib.locale.ctype](#) says:

The instantiations required in Table 51 (22.1.1.1.1) namely `ctype<char>` and `ctype<wchar_t>`, implement character classing appropriate to the implementation's native character set.

However, Section 22.2.1.3 [lib.facet.ctype.special](#) only has a detailed description of the `ctype<char>` specialization, not the `ctype<wchar_t>` specialization.

Proposed Resolution:

Add the `ctype<wchar_t>` detailed class description to Section 22.2.1.3 [lib.facet.ctype.special](#).

Rationale:

Specialization for `wchar_t` is not needed since the default is acceptable.

128. Need `open_mode()` function for file stream, string streams, file buffers, and string buffers

Section: 27.7 [lib.string.streams](#) and 27.8 [lib.file.streams](#) **Status:** [NAD Future](#) **Submitter:** Angelika Langer **Date:** 22 Feb 1999

The following question came from Thorsten Herlemann:

You can set a mode when constructing or opening a file-stream or filebuf, e.g. `ios::in`, `ios::out`, `ios::binary`, ... But how can I get that mode later on, e.g. in my own operator `<<` or operator `>>` or when I want to check whether a file-stream or file-buffer object passed as parameter is opened for input or output or binary? Is there no possibility? Is this a design-error in the standard C++ library?

It is indeed impossible to find out what a stream's or stream buffer's open mode is, and without that knowledge you don't know how certain operations behave. Just think of the append mode.

Both streams and stream buffers should have a `mode()` function that returns the current open mode setting.

Proposed Resolution:

For stream buffers, add a function to the base class as a non-virtual function qualified as `const` to 27.5.2 [lib.streambuf](#)

```
openmode mode() const;
```

Returns the current open mode.

With streams, I'm not sure what to suggest. In principle, the mode could already be returned by `ios_base`, but the mode is only initialized for file and string stream objects, unless I'm overlooking anything. For this reason it should be added to the most derived stream classes. Alternatively, it could be added to `basic_ios` and would be default initialized in `basic_ios<>::init()`.

Rationale:

This might be an interesting extension for some future, but it is not a defect in the current standard. The Proposed Resolution is retained for future reference.

130. Return type of `container::erase(iterator)` differs for associative containers

Section: 23.1.2 [lib.associative.reqmts](#), 23.1.1 [lib.sequence.reqmts](#) **Status:** [NAD Future](#) **Submitter:** Andrew Koenig **Date:** 2 Mar 1999

Table 67 (23.1.1) says that `container::erase(iterator)` returns an iterator. Table 69 (23.1.2) says that in addition to this requirement, associative containers also say that `container::erase(iterator)` returns void.

That's not an addition; it's a change to the requirements, which has the effect of making associative containers fail to meet the requirements for containers.

Rationale:

The LWG believes this was an explicit design decision by Alex Stepanov driven by complexity considerations. It has been previously discussed and reaffirmed, so this is not a defect in the current standard. A future standard may wish to reconsider this issue.

131. `list::splice` throws nothing

Section: 23.2.2.4 [lib.list.ops](#) **Status:** [NAD](#) **Submitter:** Howard Hinnant **Date:** 6 Mar 99

What happens if a splice operation causes the size() of a list to grow beyond max_size()?

Rationale:

Size() cannot grow beyond max_size().

135. basic_iostream doubly initialized

Section: 27.6.1.5.1 [lib.iostream.cons](#) **Status:** [NAD](#) **Submitter:** Howard Hinnant **Date:** 6 Mar 99

-1- Effects Constructs an object of class basic_iostream, assigning initial values to the base classes by calling basic_istream<charT,traits>(sb) (lib.istream) and basic_ostream<charT,traits>(sb) (lib.ostream)

The called for basic_istream and basic_ostream constructors call init(sb). This means that the basic_iostream's virtual base class is initialized twice.

Proposed Resolution:

Change 27.6.1.5.1, paragraph 1 to:

-1- Effects Constructs an object of class basic_iostream, assigning initial values to the base classes by calling basic_istream<charT,traits>(sb) (lib.istream).

Rationale:

The LWG agreed that the `init()` function is called twice, but said that this is harmless and so not a defect in the standard.

138. Class ctype_byname<char> redundant and misleading

Section: 22.2.1.4 [lib.locale.ctype.byname.special](#) **Status:** [NAD Future](#) **Submitter:** Angelika Langer **Date:** March 18, 1999

Section 22.2.1.4 [lib.locale.ctype.byname.special](#) specifies that ctype_byname<char> must be a specialization of the ctype_byname template.

It is common practice in the standard that specializations of class templates are only mentioned where the interface of the specialization deviates from the interface of the template that it is a specialization of. Otherwise, the fact whether or not a required instantiation is an actual instantiation or a specialization is left open as an implementation detail.

Clause 22.2.1.4 deviates from that practice and for that reason is misleading. The fact, that ctype_byname<char> is specified as a specialization suggests that there must be something "special" about it, but it has the exact same interface as the ctype_byname template. Clause 22.2.1.4 does not have any explanatory value, is at best redundant, at worst misleading - unless I am missing anything.

Naturally, an implementation will most likely implement ctype_byname<char> as a specialization, because the base class ctype<char> is a specialization with an interface different from the ctype template, but that's an implementation detail and need not be mentioned in the standard.

Rationale:

The standard as written is mildly misleading, but the correct fix is to deal with the underlying problem in the `ctype_byname` base class, not in the specialization. See issue [228](#).

140. `map<Key, T>::value_type` does not satisfy the assignable requirement

Section: 23.3.1 [lib.map](#) **Status:** [NAD Future](#) **Submitter:** Mark Mitchell **Date:** 14 Apr 99

[\[lib.container.requirements\]](#)

expression	return type	pre/post-condition
-----	-----	-----
<code>X::value_type</code>	<code>T</code>	<code>T</code> is assignable

[\[lib.map\]](#)

A map satisfies all the requirements of a container.

For a `map<Key, T> ...` the `value_type` is `pair<const Key, T>`.

There's a contradiction here. In particular, `'pair<const Key, T>'` is not assignable; the `'const Key'` cannot be assigned to. So, `map<Key, T>::value_type` does not satisfy the assignable requirement imposed by a container.

[See [103](#) for the slightly related issue of modification of set keys]

Rationale:

The LWG believes that the standard is inconsistent, but that this is a design problem rather than a strict defect. May wish to reconsider for the next standard.

143. C.h header wording unclear

Section: D.5 [depr.c.headers](#) **Status:** [NAD](#) **Submitter:** Christophe de Dinechin **Date:** 4 May 99

[\[depr.c.headers\]](#) paragraph 2 reads:

Each C header, whose name has the form `name.h`, behaves as if each name placed in the Standard library namespace by the corresponding `cname` header is also placed within the namespace scope of the namespace `std` and is followed by an explicit using-declaration (`_namespace.udecl_`)

I think it should mention the global name space somewhere... Currently, it indicates that name placed in `std` is also placed in `std`...

I don't know what is the correct wording. For instance, if `struct tm` is defined in `time.h`, `ctime` declares `std::tm`. However, the current wording seems ambiguous regarding which of the following would occur for use of both `ctime` and `time.h`:

```
// version 1:
namespace std {
    struct tm { ... };
}
```



```

    }
    using std::tm;

    // version 2:
    struct tm { ... };
    namespace std {
        using ::tm;
    }

    // version 3:
    struct tm { ... };
    namespace std {
        struct tm { ... };
    }

```

I think version 1 is intended.

[Kona: The LWG agreed that the wording is not clear. It also agreed that version 1 is intended, version 2 is not equivalent to version 1, and version 3 is clearly not intended. The example below was constructed by Nathan Myers to illustrate why version 2 is not equivalent to version 1.]

Although not equivalent, the LWG is unsure if (2) is enough of a problem to be prohibited. Points discussed in favor of allowing (2):

- *It may be a convenience to implementors.*
- *The only cases that fail are structs, of which the C library contains only a few.*

]

Example:

```

#include <time.h>
#include <utility>

int main() {
    std::tm * t;
    make_pair( t, t ); // okay with version 1 due to Koenig lookup
                      // fails with version 2; make_pair not found
    return 0;
}

```

The suggested resolution was:

Replace D.5 [depr.c.headers](#) paragraph 2 with:

Each C header, whose name has the form name.h, behaves as if each name placed in the Standard library namespace by the corresponding cname header is also placed within the namespace scope of the namespace std by name.h and is followed by an explicit using-declaration (`_namespace.udecl_`) in global scope.

Rationale:

The current wording in the standard is the result of a difficult compromise which averted delay of the standard. Based on discussions in Tokyo, it is clear that there is still no consensus on stricter wording, so the issue has been closed. It is suggested that users not write code that depends on Koenig lookup of C library functions.

145. adjustfield lacks default value

Section: 27.4.4.1 [lib.basic.ios.cons](#) **Status:** [NAD](#) **Submitter:** Angelika Langer **Date:** 12 May 99

There is no initial value for the adjustfield defined, although many people believe that the default adjustment were right. This is a common misunderstanding. The standard only defines that, if no adjustment is specified, all the predefined inserters must add fill characters before the actual value, which is "as if" the right flag were set. The flag itself need not be set.

When you implement a user-defined inserter you cannot rely on right being the default setting for the adjustfield. Instead, you must be prepared to find none of the flags set and must keep in mind that in this case you should make your inserter behave "as if" the right flag were set. This is surprising to many people and complicates matters more than necessary.

Unless there is a good reason why the adjustfield should not be initialized I would suggest to give it the default value that everybody expects anyway.

Rationale:

This is not a defect. It is deliberate that the default is no bits set. Consider Arabic or Hebrew, for example. See 22.2.2.2.2 [\[lib.facet.num.put.virtuals\]](#) paragraph 19, Table 61 - Fill padding.

149. Insert should return iterator to first element inserted

Section: 23.1.1 [lib.sequence.reqmts](#) **Status:** [NAD Future](#) **Submitter:** Andrew Koenig **Date:** 28 Jun 99

Suppose that *c* and *c1* are sequential containers and *i* is an iterator that refers to an element of *c*. Then I can insert a copy of *c1*'s elements into *c* ahead of element *i* by executing

```
c.insert(i, c1.begin(), c1.end());
```

If *c* is a vector, it is fairly easy for me to find out where the newly inserted elements are, even though *i* is now invalid:

```
size_t i_loc = i - c.begin();
c.insert(i, c1.begin(), c1.end());
```

and now the first inserted element is at *c.begin()+i_loc* and one past the last is at *c.begin()+i_loc+c1.size()*.

But what if *c* is a list? I can still find the location of one past the last inserted element, because *i* is still valid. To find the location of the first inserted element, though, I must execute something like

```
for (size_t n = c1.size(); n; --n)
    --i;
```

because *i* is now no longer a random-access iterator.

Alternatively, I might write something like

```
bool first = i == c.begin();
list<T>::iterator j = i;
if (!first) --j;
```



```

c.insert(i, cl.begin(), cl.end());
if (first)
    j = c.begin();
else
    ++j;

```

which, although wretched, requires less overhead.

But I think the right solution is to change the definition of `insert` so that instead of returning `void`, it returns an iterator that refers to the first element inserted, if any, and otherwise is a copy of its first argument.

Rationale:

The LWG believes this was an intentional design decision and so is not a defect. It may be worth revisiting for the next standard.

157. Meaningless error handling for `pword()` and `word()`

Section:: 27.4.2.5 [lib.ios.base.storage](#) **Status:** [Dup](#) **Submitter:** Dietmar Kühl **Date:** 20 Jul 1999

According to paragraphs 2 and 4 of 27.4.2.5 ([lib.ios.base.storage](#)), the functions `word()` and `pword()` "set the badbit (which might throw an exception)" on failure. ... but what does it mean for `ios_base` to set the badbit? The state facilities of the `IOStream` library are defined in `basic_ios`, a derived class! It would be possible to attempt a down cast but then it would be necessary to know the character type used...

Rationale:

Duplicate. See issue [41](#).

162. Really "formatted input functions"?

Section:: 27.6.1.2.3 [lib.istream::extractors](#) **Status:** [Dup](#) **Submitter:** Dietmar Kühl **Date:** 20 Jul 1999

It appears to be somewhat nonsensical to consider the functions defined in the paragraphs 1 to 5 to be "Formatted input function" but since these functions are defined in a section labeled "Formatted input functions" it is unclear to me whether these operators are considered formatted input functions which have to conform to the "common requirements" from 27.6.1.2.1 ([lib.istream.formatted.reqmts](#)): If this is the case, all manipulators, not just `ws`, would skip whitespace unless `noskipws` is set (... but setting `noskipws` using the manipulator syntax would also skip whitespace :-)

See also below for the same problem is [formatted output](#)

Rationale:

Duplicate. See issue [60](#).

163. Return of `gcount()` after a call to `gcount`

Section:: 27.6.1.3 [lib.istream.unformatted](#) **Status:** [Dup](#) **Submitter:** Dietmar Kühl **Date:** 20 Jul 99

It is not clear which functions are to be considered unformatted input functions. As written, it seems that all functions in 27.6.1.3 ([lib.istream.unformatted](#)) are unformatted input functions. However, it does not really make much sense to construct a sentry object for `gcount()`, `sync()`, ... Also it is unclear what happens to the `gcount()` if eg. `gcount()`, `putback()`, `unget()`, or `sync()` is called: These functions don't extract characters, some of them even "unextract" a character. Should this still be reflected in `gcount()`? Of course, it could be read as if after a call to `gcount()` `gcount()` return 0 (the last unformatted input function, `gcount()`, didn't extract any character) and after a call to `putback()` `gcount()` returns -1 (the last unformatted input function `putback()` did "extract" back into the stream). Correspondingly for `unget()`. Is this what is intended? If so, this should be clarified. Otherwise, a corresponding clarification should be used.

Rationale:

Duplicate. See issue [60](#).

166. Really "formatted output functions"?

Section:: 27.6.2.5.3 [lib ostream.inserters](#) **Status:** [Dup](#) **Submitter:** Dietmar Kühl **Date:** 20 Jul 1999

From 27.6.2.5.1 ([lib ostream.formatted.reqmts](#)) it appears that all the functions defined in 27.6.2.5.3 ([lib ostream.inserters](#)) have to construct a sentry object. Is this really intended?

This is basically the same problem as the corresponding defect report for [formatted input](#) but for output instead of input.

Rationale:

Duplicate. See issue [60](#).

177. Complex operators cannot be explicitly instantiated

Section: 26.2.6 [lib.complex.ops](#) **Status:** [NAD](#) **Submitter:** Judy Ward **Date:** 2 Jul 99

A user who tries to explicitly instantiate a complex non-member operator will get compilation errors. Below is a simplified example of the reason why. The problem is that `iterator_traits` cannot be instantiated on a non-pointer type like `float`, yet when the compiler is trying to decide which `operator+` needs to be instantiated it must instantiate the declaration to figure out the first argument type of a `reverse_iterator` operator.

```
namespace std {
template <class Iterator>
struct iterator_traits
{
    typedef typename Iterator::value_type value_type;
};

template <class T> class reverse_iterator;

// reverse_iterator operator+
template <class T>
reverse_iterator<T> operator+
```



```
(typename iterator_traits<T>::difference_type, const reverse_iterator<T>&);

template <class T> struct complex {};

// complex operator +
template <class T>
complex<T> operator+ (const T& lhs, const complex<T>& rhs)
{ return complex<T>(); }

// request for explicit instantiation
template std::complex<float> std::operator+<float>(const float&,
    const std::complex<float>&);
```

See also c++-stdlib reflector messages: lib-6814, 6815, 6816.

Rationale:

Implementors can make minor changes and the example will work. Users are not affected in any case.

According to John Spicer, It is possible to explicitly instantiate these operators using different syntax: change "std::operator+<float>" to "std::operator+".

The proposed resolution of issue 120 is that users will not be able to explicitly instantiate standard library templates. If that resolution is accepted then library implementors will be the only ones that will be affected by this problem, and they must use the indicated syntax.

178. Should clog and cerr initially be tied to cout?

Section: 27.3.1 [lib.narrow.stream.objects](#) **Status:** [NAD](#) **Submitter:** Judy Ward **Date:** 2 Jul 99

Section 27.3.1 says "After the object cerr is initialized, cerr.flags() & unitbuf is nonzero. Its state is otherwise the same as required for ios_base::init (lib.basic.ios.cons). It doesn't say anything about the state of clog. So this means that calling cerr.tie() and clog.tie() should return 0 (see Table 89 for ios_base::init effects).

Neither of the popular standard library implementations that I tried does this, they both tie cerr and clog to &cout. I would think that would be what users expect.

Rationale:

The standard is clear as written.

27.3.1/5 says that "After the object cerr is initialized, cerr.flags() & unitbuf is nonzero. Its state is otherwise the same as required for ios_base::init (27.4.4.1)." Table 89 in 27.4.4.1, which gives the postconditions of basic_ios::init(), says that tie() is 0. (Other issues correct ios_base::init to basic_ios::init().)

180. Container member iterator arguments constness has unintended consequences

Section: 23 [lib.containers](#) **Status:** [NAD Future](#) **Submitter:** Dave Abrahams **Date:** 1 Jul 99

It is the constness of the container which should control whether it can be modified through a member function such as `erase()`, not the constness of the iterators. The iterators only serve to give positioning information.

Here's a simple and typical example problem which is currently very difficult or impossible to solve without the change proposed below.

Wrap a standard container `C` in a class `W` which allows clients to find and read (but not modify) a subrange of `(C.begin(), C.end())`. The only modification clients are allowed to make to elements in this subrange is to erase them from `C` through the use of a member function of `W`.

The proposed resolution was:

Change all non-const iterator parameters of standard library container member functions to accept `const_iterator` parameters. Note that this change applies to all library clauses, including strings.

For example, in 21.3.5.5 change:

```
iterator erase(iterator p);
```

to:

```
iterator erase(const_iterator p);
```

Rationale:

The issue was discussed at length. It was generally agreed that 1) There is no major technical argument against the change (although there is a minor argument that some obscure programs may break), and 2) Such a change would not break const correctness. The concerns about making the change were 1) it is user detectable (although only in boundary cases), 2) it changes a large number of signatures, and 3) it seems more of a design issue than an out-and-out defect..

The LWG believes that this issue should be considered as part of a general review of const issues for the next revision of the standard. Also see issue [200](#).

188. valarray helpers missing augmented assignment operators

Section: 26.3.2.6 [lib.valarray.cassign](#) **Status:** [NAD Future](#) **Submitter:** Gabriel Dos Reis **Date:** 15 Aug 1999

26.3.2.6 defines augmented assignment operators `valarray<T>::op=(const T&)`, but fails to provide corresponding versions for the helper classes. Thus making the following illegal:

```
#include <valarray>

int main()
{
    std::valarray<double> v(3.14, 1999);

    v[99] *= 2.0; // Ok

    std::slice s(0, 50, 2);

    v[s] *= 2.0; // ERROR
}
```

I can't understand the intent of that omission. It makes the `valarray` library less intuitive and less useful.

Rationale:

Although perhaps an unfortunate design decision, the omission is not a defect in the current standard. A future standard may wish to add the missing operators.

190. min() and max() functions should be std::binary_functions

Section: 25.3.7 [lib.alg.min.max](#) **Status:** [NAD Future](#) **Submitter:** Mark Rintoul **Date:** 26 Aug 99

Both std::min and std::max are defined as template functions. This is very different than the definition of std::plus (and similar structs) which are defined as function objects which inherit std::binary_function.

This lack of inheritance leaves std::min and std::max somewhat useless in standard library algorithms which require a function object that inherits std::binary_function.

Rationale:

Although perhaps an unfortunate design decision, the omission is not a defect in the current standard. A future standard may wish to consider additional function objects.

191. Unclear complexity for algorithms such as binary search

Section: 25.3.3 [lib.alg.binary.search](#) **Status:** [NAD](#) **Submitter:** Nico Josuttis **Date:** 10 Oct 99

The complexity of binary_search() is stated as "At most log(last-first) + 2 comparisons", which seems to say that the algorithm has logarithmic complexity. However, this algorithm is defined for forward iterators. And for forward iterators, the need to step element-by-element results into linear complexity. But such a statement is missing in the standard. The same applies to lower_bound(), upper_bound(), and equal_range().

However, strictly speaking the standard contains no bug here. So this might be considered to be a clarification or improvement.

Rationale:

The complexity is expressed in terms of comparisons, and that complexity can be met even if the number of iterators accessed is linear. Paragraph 1 already says exactly what happens to iterators.

192. a.insert(p,t) is inefficient and overconstrained

Section: 23.1.2 [lib.associative.reqmts](#) **Status:** [NAD Future](#) **Submitter:** Ed Brey **Date:** 6 Jun 99

As defined in 23.1.2, paragraph 7 (table 69), a.insert(p,t) suffers from several problems:

expression	return type	pre/post-condition	complexity
------------	-------------	--------------------	------------

<code>a.insert(p,t)</code>	iterator	inserts t if and only if there is no element with key equivalent to the key of t in containers with unique keys; always inserts t in containers with equivalent keys. always returns the iterator pointing to the element with key equivalent to the key of t . iterator p is a hint pointing to where the insert should start to search.	logarithmic in general, but amortized constant if t is inserted right after p .
----------------------------	----------	---	---

1. For a container with unique keys, only logarithmic complexity is guaranteed if no element is inserted, even though constant complexity is always possible if p points to an element equivalent to t.
2. For a container with equivalent keys, the amortized constant complexity guarantee is only useful if no key equivalent to t exists in the container. Otherwise, the insertion could occur in one of multiple locations, at least one of which would not be right after p.
3. By guaranteeing amortized constant complexity only when t is inserted after p, it is impossible to guarantee constant complexity if t is inserted at the beginning of the container. Such a problem would not exist if amortized constant complexity was guaranteed if t is inserted before p, since there is always some p immediately before which an insert can take place.
4. For a container with equivalent keys, p does not allow specification of where to insert the element, but rather only acts as a hint for improving performance. This negates the added functionality that p would provide if it specified where within a sequence of equivalent keys the insertion should occur. Specifying the insert location provides more control to the user, while providing no disadvantage to the container implementation.

The resolution proposed was:

In 23.1.2 [lib.associative.reqmts](#) paragraph 7, replace the row in table 69 for `a.insert(p,t)` with the following two rows:

expression	return type	pre/post-condition	complexity
<code>a_uniq.insert(p,t)</code>	iterator	inserts t if and only if there is no element with key equivalent to the key of t. returns the iterator pointing to the element with key equivalent to the key of t.	logarithmic in general, but amortized constant if t is inserted right before p or p points to an element with key equivalent to t.
<code>a_eq.insert(p,t)</code>	iterator	inserts t and returns the iterator pointing to the newly inserted element. t is inserted right before p if doing so preserves the container ordering.	logarithmic in general, but amortized constant if t is inserted right before p.

Rationale:

Too big a change. Furthermore, implementors report checking both before p and after p, and don't want to change this behavior.

194. rdbuf() functions poorly specified

Section: 27.4.4 [lib.ios](#) **Status:** [NAD](#) **Submitter:** Steve Clamage **Date:** 7 Sep 99

In classic iostreams, base class ios had an rdbuf function that returned a pointer to the associated streambuf. Each derived class had its own rdbuf function that returned a pointer of a type reflecting the actual type derived from streambuf. Because in ARM C++, virtual function overrides had to have the same return type, rdbuf could not be virtual.

In standard iostreams, we retain the non-virtual rdbuf function design, and in addition have an overloaded rdbuf function that sets the buffer pointer. There is no need for the second function to be virtual nor to be implemented in derived classes.

Minor question: Was there a specific reason not to make the original rdbuf function virtual?

Major problem: Friendly compilers warn about functions in derived classes that hide base-class overloads. Any standard implementation of iostreams will result in such a warning on each of the istream classes, because of the ill-considered decision to overload rdbuf only in a base class.

In addition, users of the second rdbuf function must use explicit qualification or a cast to call it from derived classes. An explicit qualification or cast to basic_ios would prevent access to any later overriding version if there was one.

What I'd like to do in an implementation is add a using- declaration for the second rdbuf function in each derived class. It would eliminate warnings about hiding functions, and would enable access without using explicit qualification. Such a change I don't think would change the behavior of any valid program, but would allow invalid programs to compile:

```
filebuf mybuf;
fstream f;
f.rdbuf(mybuf); // should be an error, no visible rdbuf
```

I'd like to suggest this problem as a defect, with the proposed resolution to require the equivalent of a using-declaration for the rdbuf function that is not replaced in a later derived class. We could discuss whether replacing the function should be allowed.

Rationale:

For historical reasons, the standard is correct as written. There is a subtle difference between the base class `rdbuf ()` and derived class `rdbuf ()`. The derived class `rdbuf ()` always returns the original streambuf, whereas the base class `rdbuf ()` will return the "current streambuf" if that has been changed by the variant you mention.

Permission is not required to add such an extension. See 17.4.4.4 [\[lib.member.functions\]](#).

196. Placement new example has alignment problems

Section: 18.4.1.3 [lib.new.delete.placement](#) **Status:** [NAD](#) **Submitter:** Herb Sutter **Date:** 15 Dec 98

The example in 18.4.1.3 [\[lib.new.delete.placement\]](#) paragraph 4 reads:

[Example: This can be useful for constructing an object at a known address:

```
char place[sizeof(Something)];
Something* p = new (place) Something();
```

end example]

This example has potential alignment problems.

Rationale:

Duplicate: see issue [114](#)

203. `basic_istream::sentry::sentry()` is uninstantiable with `ctype<user-defined type>`

Section: 27.6.1.1.2 [lib.istream::sentry](#) **Status:** [NAD](#) **Submitter:** Matt McClure and Dietmar Kuehl **Date:** 1 Jan 2000

27.6.1.1.2 Paragraph 4 states:

To decide if the character `c` is a whitespace character, the constructor performs "as if" it executes the following code fragment:

```
const ctype<charT>& ctype = use_facet<ctype<charT> >(is.getloc());
if (ctype.is(ctype.space,c)!=0)
// c is a whitespace character.
```

But Table 51 in 22.1.1.1.1 only requires an implementation to provide specializations for `ctype<char>` and `ctype<wchar_t>`. If `sentry`'s constructor is implemented using `ctype`, it will be uninstantiable for a user-defined character type `charT`, unless the implementation has provided non-working (since it would be impossible to define a correct `ctype<charT>` specialization for an arbitrary `charT`) definitions of `ctype`'s virtual member functions.

It seems the intent the standard is that `sentry` should behave, in every respect, not just during execution, as if it were implemented using `ctype`, with the burden of providing a `ctype` specialization falling on the user. But as it is written, nothing requires the translation of `sentry`'s constructor to behave as if it used the above code, and it would seem therefore, that `sentry`'s constructor should be instantiable for all character types.

Note: If I have misinterpreted the intent of the standard with respect to `sentry`'s constructor's instantiability, then a note should be added to the following effect:

An implementation is forbidden from using the above code if it renders the constructor uninstantiable for an otherwise valid character type.

In any event, some clarification is needed.

Rationale:

It is possible but not easy to instantiate on types other than `char` or `wchar_t`; many things have to be done first. That is by intention and is not a defect.

204. `distance(first, last)` when "last" is before "first"

Section: 24.3.4 [lib.iterator.operations](#) **Status:** [NAD](#) **Submitter:** Rintala Matti **Date:** 28 Jan 00

Section 24.3.4 describes the function `distance(first, last)` (where `first` and `last` are iterators) which calculates "the number of increments or decrements needed to get from 'first' to 'last'".

The function should work for forward, bidirectional and random access iterators, and there is a requirement 24.3.4.5 which states that "'last' must be reachable from 'first'".

With random access iterators the function is easy to implement as "last - first".

With forward iterators it's clear that 'first' must point to a place before 'last', because otherwise 'last' would not be reachable from 'first'.

But what about bidirectional iterators? There 'last' is reachable from 'first' with the -- operator even if 'last' points to an earlier position than 'first'. However, I cannot see how the distance() function could be implemented if the implementation does not know which of the iterators points to an earlier position (you cannot use ++ or -- on either iterator if you don't know which direction is the "safe way to travel").

The paragraph 24.3.4.1 states that "for ... bidirectional iterators they use ++ to provide linear time implementations". However, the ++ operator is not mentioned in the reachability requirement. Furthermore 24.3.4.4 explicitly mentions that distance() returns the number of increments _or decrements_, suggesting that it could return a negative number also for bidirectional iterators when 'last' points to a position before 'first'.

Is a further requirement is needed to state that for forward and bidirectional iterators "'last' must be reachable from 'first' using the ++ operator". Maybe this requirement might also apply to random access iterators so that distance() would work the same way for every iterator category?

Rationale:

"Reachable" is defined in the standard in 24.1 paragraph 6 [[lib.iterator.requirements](#)]. The definition is only in terms of operator++(). The LWG sees no defect in the standard.

205. numeric_limits unclear on how to determine floating point types

Section: 18.2.1.2 [lib.numeric.limits.members](#) **Status:** [NAD](#) **Submitter:** Steve Cleary **Date:** 28 Jan 00

In several places in 18.2.1.2 [[lib.numeric.limits.members](#)], a member is described as "Meaningful for all floating point types." However, no clear method of determining a floating point type is provided.

In 18.2.1.5 [[lib.numeric.special](#)], paragraph 1 states "... (for example, epsilon() is only meaningful if is_integer is false). ." which suggests that a type is a floating point type if is_specialized is true and is_integer is false; however, this is unclear.

When clarifying this, please keep in mind this need of users: what exactly is the definition of floating point? Would a fixed point or rational representation be considered one? I guess my statement here is that there could also be types that are neither integer or (strictly) floating point.

Rationale:

It is up to the implementor of a user define type to decide if it is a floating point type.

206. operator new(size_t, nothrow) may become unlinked to ordinary operator new if ordinary version replaced

Section: 18.4.1.1 [lib.new.delete.single](#) **Status:** [NAD](#) **Submitter:** Howard Hinnant **Date:** 29 Aug 99

As specified, the implementation of the nothrow version of operator new does not necessarily call the ordinary operator new, but may instead simply call the same underlying allocator and return a null pointer instead of throwing an exception in case of failure.

Such an implementation breaks code that replaces the ordinary version of new, but not the nothrow version. If the ordinary version of new/delete is replaced, and if the replaced delete is not compatible with pointers returned from the library versions of new, then when the replaced delete receives a pointer allocated by the library new(nothrow), crash follows.

The fix appears to be that the lib version of new(nothrow) must call the ordinary new. Thus when the ordinary new gets replaced, the lib version will call the replaced ordinary new and things will continue to work.

An alternative would be to have the ordinary new call new(nothrow). This seems sub-optimal to me as the ordinary version of new is the version most commonly replaced in practice. So one would still need to replace both ordinary and nothrow versions if one wanted to replace the ordinary version.

Another alternative is to put in clear text that if one version is replaced, then the other must also be replaced to maintain compatibility. Then the proposed resolution below would just be a quality of implementation issue. There is already such text in paragraph 7 (under the new(nothrow) version). But this nuance is easily missed if one reads only the paragraphs relating to the ordinary new.

Rationale:

Yes, they may become unlinked, and that is by design. If a user replaces one, the user should also replace the other.

213. Math function overloads ambiguous

Section: 26.5 [lib.c.math](#) **Status:** [NAD](#) **Submitter:** Nico Josuttis **Date:** 26 Feb 00

Due to the additional overloaded versions of numeric functions for float and long double according to Section 26.5, calls such as `int x; std::pow(x, 4)` are ambiguous now in a standard conforming implementation. Current implementations solve this problem very different (overload for all types, don't overload for float and long double, use preprocessor, follow the standard and get ambiguities).

This behavior should be standardized or at least identified as implementation defined.

Rationale:

These math issues are an understood and accepted consequence of the design. They have been discussed several times in the past. Users must write casts or write floating point expressions as arguments.

215. Can a map's key_type be const?

Section: 23.1.2 [lib.associative.reqmts](#) **Status:** [NAD](#) **Submitter:** Judy Ward **Date:** 29 Feb 00

A user noticed that this doesn't compile with the Rogue Wave library because the `rb_tree` class declares a `key_allocator`, and `allocator<const int>` is not legal, I think:

```
map < const int, ... > // legal?
```


which made me wonder whether it is legal for a map's key_type to be const. In email from Matt Austern he said:

I'm not sure whether it's legal to declare a map with a const key type. I hadn't thought about that question until a couple weeks ago. My intuitive feeling is that it ought not to be allowed, and that the standard ought to say so. It does turn out to work in SGI's library, though, and someone in the compiler group even used it. Perhaps this deserves to be written up as an issue too.

Rationale:

The "key is assignable" requirement from table 69 in 23.1.2 [[lib.associative.reqmts](#)] already implies the key cannot be const.

216. setbase manipulator description flawed

Section: 27.6.3 [lib.std.manip](#) **Status:** [Dup](#) **Submitter:** Hyman Rosen **Date:** 29 Feb 00

27.6.3 [lib.std.manip](#) paragraph 5 says:

```
smanip setbase(int base);
```

Returns: An object s of unspecified type such that if out is an (instance of) basic_ostream then the expression out<<s behaves as if f(s) were called, in is an (instance of) basic_istream then the expression in>>s behaves as if f(s) were called. Where f can be defined as:

```
ios_base& f(ios_base& str, int base)
{
    // set basefield
    str.setf(n == 8 ? ios_base::oct :
            n == 10 ? ios_base::dec :
            n == 16 ? ios_base::hex :
            ios_base::fmtflags(0), ios_base::basefield);
    return str;
}
```

There are two problems here. First, f takes two parameters, so the description needs to say that out<<s and in>>s behave as if f(s,base) had been called. Second, f has a parameter named base, but is written as if the parameter was named n.

Actually, there's a third problem. The paragraph has grammatical errors. There needs to be an "and" after the first comma, and the "Where f" sentence fragment needs to be merged into its preceding sentence. You may also want to format the function a little better. The formatting above is more-or-less what the Standard contains.

Proposed Resolution:

The resolution of this defect is subsumed by the proposed resolution for issue [183](#).

[Tokyo: The LWG agrees that this is a defect and notes that it occurs additional places in the section, all requiring fixes.

218. Algorithms do not use binary predicate objects for default comparisons

Section: 25.3 [lib.alg.sorting](#), 25 [lib.algorithms](#) **Status:** [NAD](#) **Submitter:** Pablo Halpern **Date:** 6 Mar 00

Many of the algorithms take an argument, `pred`, of template parameter type `BinaryPredicate` or an argument `comp` of template parameter type `Compare`. These algorithms usually have an overloaded version that does not take the predicate argument. In these cases `pred` is usually replaced by the use of `operator==` and `comp` is replaced by the use of `operator<`.

This use of hard-coded operators is inconsistent with other parts of the library, particularly the containers library, where equality is established using `equal_to<>` and ordering is established using `less<>`. Worse, the use of `operator<`, would cause the following innocent-looking code to have undefined behavior:

```
vector<string*> vec;
sort(vec.begin(), vec.end());
```

The use of `operator<` is not defined for pointers to unrelated objects. If `std::sort` used `less<>` to compare elements, then the above code would be well-defined, since `less<>` is explicitly specialized to produce a total ordering of pointers.

Rationale:

This use of `operator==` and `operator<` was a very deliberate, conscious, and explicitly made design decision; these operators are often more efficient. The predicate forms are available for users who don't want to rely on `operator==` and `operator<`.

219. find algorithm missing version that takes a binary predicate argument

Section: 25.1.2 [lib.alg.find](#) **Status:** [NAD Future](#) **Submitter:** Pablo Halpern **Date:** 6 Mar 00

The `find` function always searches for a value using `operator==` to compare the value argument to each element in the input iterator range. This is inconsistent with other `find`-related functions such as `find_end` and `find_first_of`, which allow the caller to specify a binary predicate object to be used for determining equality. The fact that this can be accomplished using a combination of `find_if` and `bind_1st` or `bind_2nd` does not negate the desirability of a consistent, simple, alternative interface to `find`.

The resolution proposed by the submitter:

In section 25.1.2 [lib.alg.find](#), add a second prototype for `find` (between the existing prototype and the prototype for `find_if`), as follows:

```
template<class InputIterator, class T, class BinaryPredicate>
    InputIterator find(InputIterator first, InputIterator last,
                      const T& value, BinaryPredicate bin_pred);
```

Change the description of the return from:

Returns: The first iterator `i` in the range `[first, last)` for which the following corresponding conditions hold: `*i == value`, `pred(*i) != false`. Returns `last` if no such iterator is found.

to:

Returns: The first iterator `i` in the range `[first, last)` for which the following corresponding condition holds: `*i == value`, `bin_pred(*i,value) != false`, `pred(*) != false`. Return `last` if no such iterator is found.

Rationale:

This is request for a pure extension, so it is not a defect in the current standard. As the submitter pointed out, "this can be accomplished using a combination of `find_if` and `bind_1st` or `bind_2nd`".

236. `ctype<char>::is()` member modifies facet

Section: 24.2.1.3.2 [lib.facet.ctype.char.members](#) **Status:** [Dup](#) **Submitter:** Dietmar Kühl **Date:** 24 Apr 2000

The description of the `is()` member in paragraph 4 of [lib.facet.ctype.char.members](#) is broken: According to this description, the second form of the `is()` method modifies the masks in the `ctype` object. The correct semantics if, of course, to obtain an array of masks. The corresponding method in the general case, ie. the `do_is()` method as described in [lib-localees.html#lib.locale.ctype.virtuals](#) paragraph 1 does the right thing.

Proposed resolution:

Change paragraph 4 from

The second form, for all `*p` in the range `[low, high)`, assigns `vec[p-low]` to `table()[<unsigned char*>p]`.

to become

The second form, for all `*p` in the range `[low, high)`, assigns `table()[<unsigned char*>p]` to `vec[p-low]`.

Rationale:

Duplicate. See issue [28](#).

244. Must `find`'s third argument be CopyConstructible?

Section: 25.1.2 [lib.alg.find](#) **Status:** [NAD](#) **Submitter:** Andrew Koenig **Date:** 02 May 2000

Is the following implementation of `find` acceptable?

```
template<class Iter, class X>
Iter find(Iter begin, Iter end, const X& x)
{
    X x1 = x;           // this is the crucial statement
    while (begin != end && *begin != x1)
        ++begin;
    return begin;
}
```

If the answer is yes, then it is implementation-dependent as to whether the following fragment is well formed:

```
vector<string> v;

find(v.begin(), v.end(), "foo");
```

At issue is whether there is a requirement that the third argument of `find` be CopyConstructible. There may be no problem here, but analysis is necessary.

Rationale:

There is no indication in the standard that `find`'s third argument is required to be Copy Constructible. The LWG believes that no such requirement was intended. As noted above, there are times when a user might reasonably pass an argument that is not Copy Constructible.

245. Which operations on `istream_iterator` trigger input operations?

Section: 24.5.1 [lib.istream.iterator](#) **Status:** [NAD](#) **Submitter:** Andrew Koenig **Date:** 02 May 2000

I do not think the standard specifies what operation(s) on `istream` iterators trigger input operations. So, for example:

```
istream_iterator<int> i(cin);

int n = *i++;
```

I do not think it is specified how many integers have been read from `cin`. The number must be at least 1, of course, but can it be 2? More?

Rationale:

The standard is clear as written: the stream is read every time `operator++` is called, and it is also read either when the iterator is constructed or when `operator*` is called for the first time. In the example above, exactly two integers are read from `cin`.

There may be a problem with the interaction between `istream_iterator` and some STL algorithms, such as `find`. There are no guarantees about how many times `find` may invoke `operator++`.

246. `a.insert(p,t)` is incorrectly specified

Section: 23.1.2 [lib.associative.reqmts](#) **Status:** [Dup](#) **Submitter:** Mark Rodgers **Date:** 19 May 2000

Closed issue 192 raised several problems with the specification of this function, but was rejected as Not A Defect because it was too big a change with unacceptable impacts on existing implementations. However, issues remain that could be addressed with a smaller change and with little or no consequent impact.

1. The specification is inconsistent with the original proposal and with several implementations.

The initial implementation by Hewlett Packard only ever looked immediately *before* `p`, and I do not believe there was any intention to standardise anything other than this behaviour. Consequently, current implementations by several leading implementers also look immediately before `p`, and will only insert after `p` in logarithmic time. I am only aware of one implementation that does actually look after `p`, and it looks before `p` as well. It is therefore doubtful that existing code would be relying on the behaviour defined in the standard, and it would seem that fixing this defect as proposed below would standardise existing practice.

2. The specification is inconsistent with insertion for sequence containers.

This is difficult and confusing to teach to newcomers. All insert operations that specify an iterator as an insertion location should have a consistent meaning for the location represented by that iterator.

3. As specified, there is no way to hint that the insertion should occur at the beginning of the container, and the way to hint that it should occur at the end is long winded and unnatural.

For a container containing n elements, there are $n+1$ possible insertion locations and $n+1$ valid iterators. For there to be a one-to-one mapping between iterators and insertion locations, the iterator must represent an insertion location immediately before the iterator.

4. When appending sorted ranges using `insert_iterators`, insertions are guaranteed to be sub-optimal.

In such a situation, the optimum location for insertion is always immediately after the element previously inserted. The mechanics of the insert iterator guarantee that it will try and insert after the element after that, which will never be correct. However, if the container first tried to insert before the hint, all insertions would be performed in amortised constant time.

Proposed Resolution:

In 23.1.2 [lib.associative.reqmts] paragraph 7, table 69, make the following changes in the row for `a.insert(p,t)`:

assertion/note pre/post condition:
Change the last sentence from

"iterator `p` is a hint pointing to where the insert should start to search."

to

"iterator `p` is a hint indicating that immediately before `p` may be a correct location where the insertion could occur."

complexity:
Change the words "right after" to "immediately before".

Rationale:

Duplicate; see open issue [233](#).

249. Return Type of `auto_ptr::operator=`

Section: 20.4.5 [lib.auto_ptr](#) **Status:** [NAD](#) **Submitter:** Joseph Gottman <joegottman@worldnet.att.net> **Date:** 30 Jun 2000

According to section 20.4.5, the function `auto_ptr::operator=()` returns a reference to an `auto_ptr`. The reason that `operator=()` usually returns a reference is to facilitate code like

```
int x,y,z;
x = y = z = 1;
```

However, given analogous code for `auto_ptr`s,

```
auto_ptr<int> x, y, z;
z.reset(new int(1));
x = y = z;
```


the result would be that `z` and `y` would both be set to `NULL`, instead of all the `auto_ptr`s being set to the same value. This makes such cascading assignments useless and counterintuitive for `auto_ptr`s.

Proposed Resolution:

Change `auto_ptr::operator=()` to return `void` instead of an `auto_ptr` reference.

Rationale:

The return value has uses other than cascaded assignments: a user can call an `auto_ptr` member function, pass the `auto_ptr` to a function, etc. Removing the return value could break working user code.

255. Why do `basic_streambuf<>::pbump()` and `gbump()` take an `int`?

Section: 27.5.2 [lib.streambuf](#) **Status:** [NAD Future](#) **Submitter:** Martin Sebor **Date:** 12 Aug 2000

The `basic_streambuf` members `gbump()` and `pbump()` are specified to take an `int` argument. This requirement prevents the functions from effectively manipulating buffers larger than `std::numeric_limits<int>::max()` characters. It also makes the common use case for these functions somewhat difficult as many compilers will issue a warning when an argument of type larger than `int` (such as `ptrdiff_t` on LLP64 architectures) is passed to either of the function. Since it's often the result of the subtraction of two pointers that is passed to the functions, a cast is necessary to silence such warnings. Finally, the usage of a native type in the functions signatures is inconsistent with other member functions (such as `sgetn()` and `sputn()`) that manipulate the underlying character buffer. Those functions take a `streamsize` argument.

Proposed Resolution:

Change the signatures of these functions in the synopsis of template class `basic_streambuf` (27.5.2) and in their descriptions (27.5.2.3.1, p4 and 27.5.2.3.2, p4) to take a `streamsize` argument.

Although this change has the potential of changing the ABI of the library, the change will affect only platforms where `int` is different than the definition of `streamsize`. However, since both functions are typically inline (they are on all known implementations), even on such platforms the change will not affect any user code unless it explicitly relies on the existing type of the functions (e.g., by taking their address). Such a possibility is IMO quite remote.

Alternate Suggestion from Howard Hinnant, c++std-lib-7780:

This is something of a nit, but I'm wondering if `streamoff` wouldn't be a better choice than `streamsize`. The argument to `pbump` and `gbump` MUST be signed. But the standard has this to say about `streamsize` (27.4.1/2/Footnote):

[Footnote: `streamsize` is used in most places where ISO C would use `size_t`. Most of the uses of `streamsize` could use `size_t`, except for the `strstreambuf` constructors, which require negative values. It should probably be the signed type corresponding to `size_t` (which is what Posix.2 calls `ssize_t`). --- end footnote]

This seems a little weak for the argument to `pbump` and `gbump`. Should we ever really get rid of `strstream`, this footnote might go with it, along with the reason to make `streamsize` signed.

Rationale:

The LWG believes this change is too big for now. We may wish to reconsider this for a future revision of the standard. One possibility is overloading pbump, rather than changing the signature.

----- End of document -----