Doc. no. J16/99-0047
WG21 N1223
Date: 3 November 1999
Project: Programming Language C++

C++ Standard Library Closed Issues List (Revision 11)

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.

Closed Issues

---

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. "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 issue19.

---

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:** **Date:** 27 May 98

In a a comp.lang.c++.moderated :

"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 ?"

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 : 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.80. Global Operators of complex declared twice

---

81. Wrong declaration of slice operations

Section: 26.3.5 lib.template.slice.array, 26.3.7 lib.template.gslice.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 gslice_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 98

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 98

You can easily create subsets, but you can't easily combine them with other subsets. Unfortunately, you almost always needs 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 objects, which could be avoided without the cast.

Proposed resolution:

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

Ratinale:

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.

---

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.

---

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.

Rationale:

The LWG believes that while the order of arguments is unfortunate, it does not constitute a defect in the standard.

---

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:

```
#include <bitset>
using namespace std;
bitset<32> b("111111111");
```

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

```
bitset<32> b(string("111111111"));
```

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

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

According to the compiler vendor, 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 templatized 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.

.

---

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:** February 22, 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 99

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.

---

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 |
> |---|---|---|
> | X::value_type | T | T 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.

---

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, c1.begin(), c1.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 `iword()`

Section:: 27.4.2.5 lib.ios.base.storage **Status:** Dup **Submitter:** Dietmar Kühl **Date:** 20 Jul 99

According to paragraphs 2 and 4 of 27.4.2.5 (lib.ios.base.storage), the functions `iword()` 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 99

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 funcion `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 99

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.

---

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 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.

---

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 99

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 algorithms 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 apllies to lower_bound(), upper_bound(), and equal_range().

However, strictly speaking the standard contains no bug here. So this might 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
`a.insert (p,t)`	`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.

Proposed Resolution:

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
`a_uniq.insert (p,t)`	`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.

`a_eq.insert (p,t)`	`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 iostream 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].

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