Doc. no. J16 00-0003
WG21 N1226
Date: 18 Feb 2000
Project: Programming Language C++

C++ Standard Library Active Issues List (Revision 12)

Reference ISO/IEC IS 14882:1998(E)

Also see:

- Table of Contents for all library issues.
- Index by Section for all library issues.
- Index by Status for all library issues.
- Library Defect Report List
- Library Closed Issues List
- How to prepare and submit an issue.

The purpose of this document is to record the status of issues which have come before the Library Working Group (LWG) of the ANSI (J16) and ISO (WG21) C++ Standards Committee. Issues represent potential defects in the ISO/IEC IS 14882:1998(E) document. Issues are not to be used to request new features or other extensions.

This document contains only library issues which are actively being considered by the Library Working Group. That is, issues which have a status of New, Open, Review, and Ready. See "C++ Standard Library Defect Report List" for issues considered defects and "C++ Standard Library Closed Issues List" for issues considered closed.

The issues in these lists are not necessarily formal ISO Defect Reports (DR's). While some issues will eventually be elevated to official Defect Report status, other issues will be disposed of in other ways. See Issue Status.

This document is in an experimental format designed for both viewing via a world-wide web browser and hard-copy printing. It is available as an HTML file for browsing or PDF file for printing.

This issues list exists in two slightly different versions; the Committee Version and the Public Version. The Committee Version is the master copy, while the Public Version is an extract with certain names, email addresses, action items, and internal committee comments removed. A line of text reading "Committee Version" following the title above identifies the Committee Version

For the most current public version of this document see http://www.dkuug.dk/jtc1/sc22/wg21. Requests for further information about this document should include the document number above, reference ISO/IEC 14882:1998(E), and be submitted to Information Technology Industry Council (ITI), 1250 Eye Street NW, Washington, DC 20005.

Public information as to how to obtain a copy of the C++ Standard, join the standards committee, submit an issue, or comment on an issue can be found in the C++ FAQ at http://reality.sgi.com/austern_mti/std-c++/faq.html. Public discussion of C++ Standard related issues occurs on news:comp.std.c++.

For committee members, files available on the committee's private web site include the HTML version of the Standard itself. HTML hyperlinks from this issues list to those files will only work for committee members who have downloaded them into the same disk directory as the issues list files.

Revision history

- R12: pre-Tokyo II: Added issues 199 to 211.
- R11: post-Kona: Updated to reflect LWG and full committee actions in Kona (99-0048/N1224). Note changed

resolution of issues 4 and 38. Added issues 196 to 198. Closed issues list split into "defects" and "closed" documents.
- R10: pre-Kona updated. Added proposed resolutions 83, 86, 91, 92, 109. Added issues 190 to 195. (99-0033/D1209, 14 Oct 99)
- R9: pre-Kona mailing. Added issues 140 to 189. Issues list split into separate "active" and "closed" documents. (99-0030/N1206, 25 Aug 99)
- R8: post-Dublin mailing. Updated to reflect LWG and full committee actions in Dublin. (99-0016/N1193, 21 Apr 99)
- R7: pre-Dublin updated: Added issues 130, 131, 132, 133, 134, 135, 136, 137, 138, 139 (31 Mar 99)
- R6: pre-Dublin mailing. Added issues 127, 128, and 129. (99-0007/N1194, 22 Feb 99)
- R5: update issues 103, 112; added issues 114 to 126. Format revisions to prepare for making list public. (30 Dec 98)
- R4: post-Santa Cruz II updated: Issues 110, 111, 112, 113 added, several issues corrected. (22 Oct 98)
- R3: post-Santa Cruz II: Issues 94 to 109 added, many issues updated to reflect LWG consensus (12 Oct 98)
- R2: pre-Santa Cruz II: Issues 73 to 93 added, issue 17 updated. (29 Sep 98)
- R1: Correction to issue 55 resolution, 60 code format, 64 title. (17 Sep 98)

Issue Status

New - The issue has not yet been reviewed by the LWG. Any **Proposed Resolution** is purely a suggestion from the issue submitter, and should not be construed as the view of LWG.

Open - The LWG has discussed the issue but is not yet ready to move the issue forward. There are several possible reasons for open status:

- Consensus may have not yet have been reached as to how to deal with the issue.
- Informal consensus may have been reached, but the LWG awaits exact **Proposed Resolution** wording for review.
- The LWG wishes to consult additional technical experts before proceeding.
- The issue may require further study.

A **Proposed Resolution** for an open issue is still not be construed as the view of LWG. Comments on the current state of discussions are often given at the end of open issues in an italic font. Such comments are for information only and should not be given undue importance. They do not appear in the public version.

Dup - The LWG has reached consensus that the issue is a duplicate of another issue, and will not be further dealt with. A **Rationale** identities the duplicated issue's issue number.

NAD - The LWG has reached consensus that the issue is not a defect in the Standard, and the issue is ready to forward to the full committee as a proposed record of response. A **Rationale** discusses the LWG's reasoning.

Review - Exact wording of a **Proposed Resolution** is now available for review on an issue for which the LWG previously reached informal consensus.

Ready - The LWG has reached consensus that the issue is a defect in the Standard, the **Proposed Resolution** is correct, and the issue is ready to forward to the full committee for further action as a Defect Report (DR).

DR - (Defect Report) - The full J16 committee has voted to forward the issue to the Project Editor to be processed as a Potential Defect Report. The Project Editor reviews the issue, and then forwards it to the WG21 Convenor, who returns it to the full committee for final disposition. This issues list accords the status of DR to all these Defect Reports regardless of where they are in that process.

TC - (Technical Corrigenda) - The full WG21 committee has voted to accept the Defect Report's Proposed Resolution as a Technical Corrigenda. Action on this issue is thus complete and no further action is possible under ISO rules.

RR - (Record of Response) - The full WG21 committee has determined that this issue is not a defect in the Standard.

Action on this issue is thus complete and no further action is possible under ISO rules.

Future - In addition to the regular status, the LWG believes that this issue should be revisited at the next revision of the standard. It is usually paired with NAD.

Issues are always given the status of New when they first appear on the issues list. They may progress to Open or Review while the LWG is actively working on them. When the LWG has reached consensus on the disposition of an issue, the status will then change to Dup, NAD, or Ready as appropriate. Once the full J16 committee votes to forward Ready issues to the Project Editor, they are given the status of Defect Report (DR). These in turn may become the basis for Technical Corrigenda (TC), or are closed without action other than a Record of Response (RR). The intent of this LWG process is that only issues which are truly defects in the Standard move to the formal ISO DR status.

Active Issues

---

3. Atexit registration during atexit() call is not described

Section: 18.3 lib.support.start.term **Status**: Open **Submitter:** Steve Clamage **Date:** 12 Dec 97 **Msg:** lib-6500

We appear not to have covered all the possibilities of exit processing with respect to atexit registration.

Example 1: (C and C++)

```
#include <stdlib.h>
void f1() { }
void f2() { atexit(f1); }

int main()
{
    atexit(f2); // the only use of f2
    return 0; // for C compatibility
}
```

At program exit, f2 gets called due to its registration in main. Running f2 causes f1 to be newly registered during the exit processing. Is this a valid program? If so, what are its semantics?

Interestingly, neither the C standard, nor the C++ draft standard nor the forthcoming C9X Committee Draft says directly whether you can register a function with atexit during exit processing.

All 3 standards say that functions are run in reverse order of their registration. Since f1 is registered last, it ought to be run first, but by the time it is registered, it is too late to be first.

If the program is valid, the standards are self-contradictory about its semantics.

Example 2: (C++ only)

```
void F() { static T t; } // type T has a destructor

int main()
{
    atexit(F); // the only use of F
}
```

Function F registered with atexit has a local static variable t, and F is called for the first time during exit processing. A local static object is initialized the first time control flow passes through its definition, and all static objects are destroyed during exit processing. Is the code valid? If so, what are its semantics?

Section 18.3 "Start and termination" says that if a function F is registered with atexit before a static object t is

initialized, F will not be called until after t's destructor completes.

In example 2, function F is registered with atexit before its local static object O could possibly be initialized. On that basis, it must not be called by exit processing until after O's destructor completes. But the destructor cannot be run until after F is called, since otherwise the object could not be constructed in the first place.

If the program is valid, the standard is self-contradictory about its semantics.

I plan to submit Example 1 as a public comment on the C9X CD, with a recommendation that the results be undefined. (Alternative: make it unspecified. I don't think it is worthwhile to specify the case where f1 itself registers additional functions, each of which registers still more functions.)

I think we should resolve the situation in the whatever way the C committee decides.

For Example 2, I recommend we declare the results undefined.

Proposed Resolution:

---

8. Locale::global lacks guarantee

Section: 22.1.1.5 lib.locale.statics **Status**: Review **Submitter:** Matt Austern **Date:** 24 Dec 97

It appears there's an important guarantee missing from clause 22. We're told that invoking locale::global(L) sets the C locale if L has a name. However, we're not told whether or not invoking setlocale(s) sets the global C++ locale.

The intent, I think, is that it should not, but I can't find any such words anywhere.

Proposed Resolution:

Add a sentence at the end of 22.1.1.5 [lib.locale.statics], paragraph 2:

> No library function other than `locale::global()` shall affect the value returned by `locale()`.

---

9. Operator new(0) calls should not yield the same pointer

Section: 18.4.1 lib.new.delete **Status**: Open **Submitter:** Steve Clamage **Date:** 4 Jan 98

comp.std.c++ posting: I just noticed that section 3.7.3.1 of CD2 seems to allow for the possibility that all calls to operator new(0) yield the same pointer, an implementation technique specifically prohibited by ARM 5.3.3.Was this prohibition really lifted? Does the FDIS agree with CD2 inthe regard? [Issues list maintainer's note: the IS is the same.]

Proposed Resolution:

[Kona: After initial discussion, Steve drafted an analysis, concluding that the choices are:

1. *A request for zero size never fails, meaning values need not be distinct.*
2. *A request for zero size returns a distinct pointer value if it succeeds, but is allowed to fail.*

After much further discussion, there was agreement that choice 2 is the desired behavior. Steve will draft wording.]

---

17. Bad bool parsing

Section: 22.2.2.1.2 lib.facet.num.get.virtuals **Status**: Ready **Submitter:** Nathan Myers **Date:** 6 Aug 98

This section describes the process of parsing a text boolean value from the input stream. It does not say it recognizes either of the sequences "true" or "false" and returns the corresponding bool value; instead, it says it recognizes only one of those sequences, and chooses which according to the received value of a reference argument intended for returning the result, and reports an error if the other sequence is found. (!) Furthermore, it claims to get the names from the ctype<> facet rather than the numpunct<> facet, and it examines the "boolalpha" flag wrongly; it doesn't define the value "loc"; and finally, it computes wrongly whether to use numeric or "alpha" parsing.

I believe the correct algorithm is "as if":

```
// in, err, val, and str are arguments.
err = 0;
const numpunct<charT>& np = use_facet<numpunct<charT> >(str.getloc());
const string_type t = np.truename(), f = np.falsename();
bool tm = true, fm = true;
size_t pos = 0;
while (tm && pos < t.size() || fm && pos < f.size()) {
  if (in == end) { err = str.eofbit; }
  bool matched = false;
  if (tm && pos < t.size()) {
    if (!err && t[pos] == *in) matched = true;
    else tm = false;
  }
  if (fm && pos < f.size()) {
    if (!err && f[pos] == *in) matched = true;
    else fm = false;
  }
  if (matched) { ++in; ++pos; }
  if (pos > t.size()) tm = false;
  if (pos > f.size()) fm = false;
}
if (tm == fm || pos == 0) { err |= str.failbit; }
else                      { val = tm; }
return in;
```

Notice this works reasonably when the candidate strings are both empty, or equal, or when one is a substring of the other. The proposed text below captures the logic of the code above.

Proposed Resolution:

In 22.2.2.1.2 [lib.facet.num.get.virtuals], in the first line of paragraph 14, change "&&" to "&".

Then, replace paragraphs 15 and 16 as follows:

> Otherwise target sequences are determined "as if" by calling the members `falsename()` and `truename()` of the facet obtained by `use_facet<>(str.getloc())`. Successive characters in the range `[in,end)` (see [lib.sequence.reqmts]) are obtained and matched against corresponding positions in the target sequences only as necessary to identify a unique match. The input iterator `in` is compared to `end` only when necessary to obtain a character. If and only if a target sequence is uniquely matched, `val` is set to the corresponding value.
>
> The `in` iterator is always left pointing one position beyond the last character successfully matched. If `val` is set, then err is set to `str.goodbit`; or to `str.eofbit` if, when seeking another character to match, it is found that `(in==end)`. If `val` is not set, then _err_ is set to `str.failbit`; or to `(str.failbit|str.eofbit)`if the reason for the failure was that `(in==end)`. [Example: for targets `true`:"a" and `false`:"abb", the input sequence "a" yields `val==true` and `err==str.eofbit`; the

input sequence "abc" yields `err=str.failbit`, with `in` ending at the 'c' element. For targets `true`:"1" and `false`:"0", the input sequence "1" yields `val==true` and `err=str.goodbit`. For empty targets (""), any input sequence yields `err==str.failbit`. --end example]

---

19. "Noconv" definition too vague

Section: 22.2.1.5.2 lib.locale.codecvt.virtuals **Status**: Review **Submitter:** Nathan Myers **Date:** 6 Aug 98

In the definitions of codecvt<>::do_out and do_in, they are specified to return noconv if "no conversion is needed". This definition is too vague, and does not say normatively what is done with the buffers.

Proposed Resolution:

Change the entry for noconv in the table under paragraph 4 in section 22.2.1.5.2 [lib.locale.codecvt.virtuals] to read:

> `noconv`: `internT` and `externT` are the same type, and input sequence is identical to converted sequence.

Change the Note in paragraph 2 to normative text as follows:

> If returns `noconv`, `internT` and `externT` are the same type and the converted sequence is identical to the input sequence `[from,from_next)`. `to_next` is set equal to `to`, the value of `state` is unchanged, and there are no changes to the values in `[to, to_limit)`.

---

21. Codecvt_byname<> instantiations

Section: 22.1.1.1.1 lib.locale.category **Status**: Ready **Submitter:** Nathan Myers **Date:** 6 Aug 98

In the second table in the section, captioned "Required instantiations", the instantiations for codecvt_byname<> have been omitted. These are necessary to allow users to construct a locale by name from facets.

Proposed Resolution:

Add in 22.1.1.1.1 [lib.locale.category] to the table captioned "Required instantiations", in the category "ctype" the lines

```
codecvt_byname<char,char,mbstate_t>,
codecvt_byname<wchar_t,char,mbstate_t>
```

---

26. Bad sentry example

Section: 27.6.1.1.2 lib.istream::sentry **Status**: Open **Submitter:** Nathan Myers **Date:** 6 Aug 98

In paragraph 6, the code in the example:

```
template <class charT, class traits = char_traits<charT> >
basic_istream<charT,traits>::sentry(
         basic_istream<charT,traits>& is, bool noskipws = false) {
    ...
    int_type c;
    typedef ctype<charT> ctype_type;
    const ctype_type& ctype = use_facet<ctype_type>(is.getloc());
```

```
      while ((c = is.rdbuf()->snextc()) != traits::eof()) {
        if (ctype.is(ctype.space,c)==0) {
          is.rdbuf()->sputbackc (c);
          break;
        }
      }
      ...
  }
```

fails to demonstrate correct use of the facilities described. In particular, it fails to use traits operators, and specifies incorrect semantics. (E.g. it specifies skipping over the first character in the sequence without examining it.)

Proposed Resolution:

Replace the example with better code, as follows:

```
template <class charT, class traits>
basic_istream<charT,traits>::sentry::sentry(
    basic_istream<charT,traits>& is, bool noskipws)
{
  typedef ctype<charT> ctype_type;
  const ctype_type& ct = use_facet<ctype_type>(is.getloc());
  for (int_type c = is.rdbuf()->sgetc();
       !traits::eq_int_type(c,traits::eof()) && ct.is(ct.space,c);
       c = is.rdbuf()->snextc())
    {}
}
```

---

31. Immutable locale values

Section: 22.1.1 [lib.locale] **Status**: Open **Submitter:** Nathan Myers **Date:** 6 Aug 98

Paragraph 6, says "An instance of _locale_ is *immutable*; once a facet reference is obtained from it, ...". This has caused some confusion, because locale variables are manifestly assignable.

Proposed Resolution:

In 22.1.1 [lib.locale] replace paragraph 6,

> An instance of locale is immutable; once a facet reference is obtained from it, that reference remains usable as long as the locale value itself exists.

with

> A locale value is immutable. This means that once a facet reference is obtained from a locale object by calling use_facet<>, that reference remains usable, and the results from member functions of it may be cached and re-used, until the locale object is assigned to or destroyed.

---

32. Pbackfail description inconsistent

Section: 27.5.2.4.4 lib.streambuf.virt.pback **Status**: Ready **Submitter:** Nathan Myers **Date:** 6 Aug 98

The description of the required state before calling virtual member basic_streambuf<>::pbackfail requirements is inconsistent with the conditions described in 27.5.2.2.4 [lib.streambuf.pub.pback] where member sputbackc calls it. Specifically, the latter says it calls pbackfail if:

traits::eq(c,gptr()[-1]) is false

where pbackfail claims to require:

traits::eq(*gptr(),traits::to_char_type(c)) returns false

It appears that the pbackfail description is wrong.

Proposed Resolution:

In 27.5.2.4.4 [lib.streambuf.virt.pback], paragraph 1, change:

"traits::eq(*gptr(),traits::to_char_type(c))"

to

"traits::eq(traits::to_char_type(c),gptr()[-1])"

Rationale:

Note deliberate reordering of arguments for clarity in addition to the correction of the argument value.

---

41. Ios_base needs clear(), exceptions()

Section: 27.4.2 [lib.ios.base] **Status**: Ready **Submitter:** Nathan Myers **Date:** 6 Aug 98

The description of ios_base::iword() and pword() in 27.4.2.4 [lib.ios.members.static], say that if they fail, they "set badbit, which may throw an exception". However, ios_base offers no interface to set or to test badbit; those interfaces are defined in basic_ios<>.

Proposed Resolution:

Change the description in 27.4.2.5 [lib.ios.members.storage] in paragraph 2, and also in paragraph 4, as follows. Replace

If the function fails it sets badbit, which may throw an exception.

with

If the function fails, and `*this` is a base sub-object of a `basic_ios<>` object or sub-object, the effect is equivalent to calling `basic_ios<>::setstate(badbit)` on the derived object (which may throw `failure`).

---

42. String ctors specify wrong default allocator

Section: 21.3 [lib.basic.string] **Status**: Ready **Submitter:** Nathan Myers **Date:** 6 Aug 98

The basic_string<> copy constructor:

```
basic_string(const basic_string& str, size_type pos = 0,
             size_type n = npos, const Allocator& a = Allocator());
```

specifies an Allocator argument default value that is counter-intuitive. The natural choice for a the allocator to copy from is str.get_allocator(). Though this cannot be expressed in default-argument notation, overloading suffices.

Alternatively, the other containers in Clause 23 (deque, list, vector) do not have this form of constructor, so it is inconsistent, and an evident source of confusion, for basic_string<> to have it, so it might better be removed.

Proposed Resolution:

In 21.3 [lib.basic.string], replace the declaration of the copy constructor as follows:

```
basic_string(const basic_string& str);
basic_string(const basic_string& str, size_type pos, size_type n = npos,
             const Allocator& a = Allocator());
```

In 21.3.1 [lib.string.cons], replace the copy constructor declaration as above. Add to paragraph 5, Effects:

> In the first form, the Allocator value used is copied from `str.get_allocator()`.

Rationale:

The LWG believes the constructor is actually broken, rather than just an unfortunate design choice.

The LWG considered two other possible resolutions:

A. In 21.3 [lib.basic.string], replace the declaration of the copy constructor as follows:

```
basic_string(const basic_string& str, size_type pos = 0,
             size_type n = npos);
basic_string(const basic_string& str, size_type pos,
             size_type n, const Allocator& a);
```

In 21.3.1 [lib.string.cons], replace the copy constructor declaration as above. Add to paragraph 5, Effects:

> When no `Allocator` argument is provided, the string is constructed using the value `str.get_allocator()`.

B. In 21.3 [lib.basic.string], and also in 21.3.1 [lib.string.cons], replace the declaration of the copy constructor as follows:

```
basic_string(const basic_string& str, size_type pos = 0,
             size_type n = npos);
```

The proposed resolution reflects the original intent of the LWG. It was also noted that this fix "will cause a small amount of existing code to now work correctly."

---

44. Iostreams use operator== on int_type values

Section: 27 [lib.input.output] **Status**: Open **Submitter:** Nathan Myers **Date:** 6 Aug 98

Many of the specifications for iostreams specify that character values or their int_type equivalents are compared using

operators == or !=, though in other places traits::eq() or traits::eq_int_type is specified to be used throughout. This is an inconsistency; we should change uses of == and != to use the traits members instead.

Proposed Resolution:

---

49. Underspecification of ios_base::sync_with_stdio

Section: 27.4.2.4 lib.ios.members.static **Status**: Open **Submitter:** Matt Austern **Date:** 21 Jun 98

Two problems.

(1) 27.4.2.4 doesn't say what ios_base::sync_with_stdio(f) returns. Does it return f, or does it return the previous synchronization state? My guess is the latter, but the standard doesn't say so.

(2) 27.4.2.4 doesn't say what it means for streams to be synchronized with stdio. Again, of course, I can make some guesses. (And I'm unhappy about the performance implications of those guesses, but that's another matter.)

Proposed Resolution:

Change the following sentenance in 27.4.2.4 lib.ios.members.static returns clause from:

> `true` if the standard iostream objects (27.3) are synchronized and otherwise returns `false`.

to:

> `true` if the previous state of the standard iostream objects (27.3) was synchronized and otherwise returns `false`.

---

50. Copy constructor and assignment operator of ios_base

Section: 27.4.2 lib.ios.base **Status**: Ready **Submitter:** Matt Austern **Date:** 21 Jun 98

As written, ios_base has a copy constructor and an assignment operator. (Nothing in the standard says it doesn't have one, and all classes have copy constructors and assingment operators unless you take specific steps to avoid them.) However, nothin in 27.4.2 says what the copy constructor and assignment operator do.

My guess is that this was an oversight, that ios_base is, like basic_ios, not supposed to have a copy constructor or an assignment operator.

A LWG member comments: Yes, its an oversight, but in the opposite sense to what you're suggesting. At one point there was a definite intention that you could copy ios_base. It's an easy way to save the entire state of a stream for future use. As you note, to carry out that intention would have required a explicit description of the semantics (e.g. what happens to the iarray and parray stuff).

Proposed Resolution:

In 27.4.2 lib.ios.base, class ios_base, specify the copy constructor and operator= members as being private.

Rationale:

The LWG believes the difficulty of specifying correct semantics outweighs any benefit of allowing ios_base objects to be

copyable.

---

53. Basic_ios destructor unspecified

Section: 27.4.4.1 lib.basic.ios.cons, 27.4.4.2 lib.basic.ios.members **Status**: Ready **Submitter:** Matt Austern **Date:** 23 Jun 98

There's nothing in 27.4.4 saying what basic_ios's destructor does.

The important question is whether basic_ios::~basic_ios() destroys rdbuf().

Proposed Resolution:

Add after 27.4.4.1 lib.basic.ios.cons paragraph 2:

```
virtual ~basic_ios();
```

Notes: The destructor does not destroy `rdbuf()`.

Rationale:

The LWG reviewed the additional question of whether or not `rdbuf(0)` may set `badbit`. The answer is clearly yes; it may be set via `clear()`. See 27.4.4.2 lib.basic.ios.members, paragraph 6.

---

60. What is a formatted input function?

Section: 27.6.1.2.1 lib.istream.formatted.reqmts **Status**: Ready **Submitter:** Matt Austern **Date:**3 Aug 98

Paragraph 1 of 27.6.1.2.1 contains general requirements for all formatted input functions. Some of the functions defined in section 27.6.1.2 explicitly say that those requirements apply ("Behaves like a formatted input member (as described in 27.6.1.2.1)"), but others don't. The question: is 27.6.1.2.1 supposed to apply to everything in 27.6.1.2, or only to those member functions that explicitly say "behaves like a formatted input member"? Or to put it differently: are we to assume that everything that appears in a section called "Formatted input functions" really is a formatted input function? I assume that 27.6.1.2.1 is intended to apply to the arithmetic extractors (27.6.1.2.2), but I assume that it is not intended to apply to extractors like

```
basic_istream& operator>>(basic_istream& (*pf)(basic_istream&));
```

and

```
basic_istream& operator>>(basic_streammbuf*);
```

There is a similar ambiguity for unformatted input, formatted output, and unformatted output.

Comments : It seems like the problem is that the basic_istream and basic_ostream operator <<()'s that are used for the manipulators and streambuf* are in the wrong section and should have their own separate section or be modified to make it clear that the "Common requirements" listed in section 27.6.1.2.1 (for basic_istream) and section 27.6.2.5.1 (for basic_ostream) do not apply to them.

Additional comments : It appears to be somewhat nonsensical to consider the functions defined in 27.6.1.2.3 lib.istream::extractors 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 :-)

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.

Proposed Resolution:

Change the standard as specified in J16/99-0043==WG21/N1219, Proposed Resolution to Library Issue 60, section "VI Wording", by Judy Ward and Matt Austern.

---

61. Ambiguity in iostreams exception policy

Section: 27.6.1.3 lib.istream.unformatted **Status**: Open **Submitter:** Matt Austern **Date:**6 Aug 98

The introduction to the section on unformatted input (27.6.1.3) says that every unformatted input function catches all exceptions that were thrown during input, sets badbit, and then conditionally rethrows the exception. That seems clear enough. Several of the specific functions, however, such as get() and read(), are documented in some circumstances as setting eofbit and/or failbit. (The standard notes, correctly, that setting eofbit or failbit can sometimes result in an exception being thrown.) The question: if one of these functions throws an exception triggered by setting failbit, is this an exception "thrown during input" and hence covered by 27.6.1.3, or does 27.6.1.3 only refer to a limited class of exceptions? Just to make this concrete, suppose you have the following snippet.

```
char buffer[N];
istream is;
...
is.exceptions(istream::failbit); // Throw on failbit but not on badbit.
is.read(buffer, N);
```

Now suppose we reach EOF before we've read N characters. What iostate bits can we expect to be set, and what exception (if any) will be thrown?

Proposed Resolution:

Alternative A:
In 27.6.1.3, paragraph 1, change "If an exception is thrown during input then..." to "If, during input, an exception is thrown by one of rdbuf()'s virtual members or by a locale or a locale facet, then...".

Alternative B:
In 27.6.1.3, paragraph 1, after the sentence that begins "If an exception is thrown...", add the following parenthetical comment: "(Exceptions thrown from `basic_ios<>::clear()` are not caught or rethrown.)"

---

63. Exception-handling policy for unformatted output

Section: 27.6.2.6 lib.ostream.unformatted **Status**: Review **Submitter:** Matt Austern **Date:**11 Aug 98

Clause 27 details an exception-handling policy for formatted input, unformatted input, and formatted output. It says nothing for unformatted output (27.6.2.6). 27.6.2.6 should either include the same kind of exception-handling policy as in the other three places, or else it should have a footnote saying that the omission is deliberate.

Proposed Resolution:

In 27.6.2.6, paragraph 1, replace the last sentence ("In any case, the unformatted output function ends by destroying the sentry object, then returning the value specified for the formatted output function.") with the following text:

> If an exception is thrown during output, then `ios::badbit` is turned on [Footnote: without causing an `ios::failure` to be thrown.] in `*this`'s error state. If `(exception() & badbit) != 0` then the exception is rethrown. In any case, the unformatted output function ends by destroying the sentry object, then, if no exception was thrown, returning the value specified for the formatted output function.

---

76. Can a `codecvt` facet always convert one internal character at a time?

Section: 22.2.1.5 lib.locale.codecvt **Status**: Open **Submitter:** Matt Austern **Date:** 25 Sep 98

This issue concerns the requirements on classes derived from `codecvt`, including user-defined classes. What are the restrictions on the conversion from external characters (e.g. `char`) to internal characters (e.g. `wchar_t`)? Or, alternatively, what assumptions about `codecvt` facets can the I/O library make?

The question is whether it's possible to convert from internal characters to external characters one internal character at a time, and whether, given a valid sequence of external characters, it's possible to pick off internal characters one at a time. Or, to put it differently: given a sequence of external characters and the corresponding sequence of internal characters, does a position in the internal sequence correspond to some position in the external sequence?

To make this concrete, suppose that `[first, last)` is a sequence of *M* external characters and that `[ifirst, ilast)` is the corresponding sequence of *N* internal characters, where *N* > *1*. That is, `my_encoding.in()`, applied to `[first, last)`, yields `[ifirst, ilast)`. Now the question: does there necessarily exist a subsequence of external characters, `[first, last_1)`, such that the corresponding sequence of internal characters is the single character `*ifirst`?

(What a "no" answer would mean is that `my_encoding` translates sequences only as blocks. There's a sequence of *M* external characters that maps to a sequence of *N* internal characters, but that external sequence has no subsequence that maps to *N-1* internal characters.)

Some of the wording in the standard, such as the description of `codecvt::do_max_length` (22.2.1.5.2, paragraph 11) and `basic_filebuf::underflow` (27.8.1.4, paragraph 3) suggests that it must always be possible to pick off internal characters one at a time from a sequence of external characters. However, this is never explicitly stated one way or the other.

This issue seems (and is) quite technical, but it is important if we expect users to provide their own encoding facets. This is an area where the standard library calls user-supplied code, so a well-defined set of requirements for the user-supplied code is crucial. Users must be aware of the assumptions that the library makes. This issue affects positioning operations on `basic_filebuf`, unbuffered input, and several of `codecvt`'s member functions.

Proposed Resolution:

---

83. String::npos vs. string::max_size()

Section: 21.3 lib.basic.string **Status**: Ready **Submitter:** Nico Josuttis **Date:** 29 Sep 98

Many string member functions throw if size is getting or exceeding npos. However, I wonder why they don't throw if size is getting or exceeding max_size() instead of npos. May be npos is known at compile time, while max_size() is known at runtime. However, what happens if size exceeds max_size() but not npos, then ? It seems the standard lacks some clarifications here.

Proposed Resolution:

After 21.3 [lib.basic.string] paragraph 4 ("The functions described in this clause...") add a new paragraph:

> For any string operation, if as a result of the operation, `size()` would exceed `max_size()` then the operation throws `length_error`.

---

86. String constructors don't describe exceptions

Section: 21.3.1 lib.string.cons **Status**: Review **Submitter:** Nico Josuttis **Date:** 29 Sep 98

The constructor from a range:

```
template<class InputIterator>
        basic_string(InputIterator begin, InputIterator end,
                     const Allocator& a = Allocator());
```

lacks a throw specification. However, I would expect that it throws according to the other constructors if the numbers of characters in the range equals npos (or exceeds max_size(), see above).

Proposed resolution:

At the beginning of 21.3.1 [lib.string.cons] paragraph 15 add:

> Throws: `length_error` if `distance(begin,end)` equals or exceeds `npos` (see [lib.iterator.operations] for `distance()`).

---

91. Description of operator>> and getline() for string<> might cause endless loop

Section: 21.3.7.9 lib.string.io **Status**: Review **Submitter:** Nico Josuttis **Date:** 29 Sep 98

Operator >> and getline() for strings read until eof() in the input stream is true. However, this might never happen, if the stream can't read anymore without reaching EOF. So shouldn't it be changed into that it reads until !good() ?

Proposed resolution:

In 21.3.7.9 [lib.string.io], paragraph 1, last sentence "Characters are extracted and appended until any of the following occurs:...", replace:

- end-of-file occurs on the input sequence;

with:

- an attempt to extract a character fails;

In 21.3.7.9 [lib.string.io], paragraph 5, last sentence, replace :

- end-of-file occurs on the input sequence (in which case, the getline function calls is.setstate(ios_base::eofbit)).

with:

- an attempt to extract a character fails

In 23.3.5.3 [lib.bitset.operators], paragraph 5, last sentence, replace:

- end-of-file occurs on the input sequence;

with:

- an attempt to extract a character fails;

---

92. Incomplete Algorithm Requirements

Section: 25 lib.algorithms **Status**: Open **Submitter:** Nico Josuttis **Date:** 29 Sep 98

The standard does not state, how often a function object is copied, called, or the order of calls inside an algorithm. This may lead to suprising/buggy behavior. Consider the following example:

```
class Nth {    // function object that returns true for the nth element
  private:
    int nth;     // element to return true for
    int count;   // element counter
  public:
    Nth (int n) : nth(n), count(0) {
    }
    bool operator() (int) {
        return ++count == nth;
    }
};
....
// remove third element
    list<int>::iterator pos;
    pos = remove_if(coll.begin(),coll.end(),  // range
                    Nth(3)),                  // remove criterion
    coll.erase(pos,coll.end());
```

This call, in fact removes the 3rd **AND the 6th** element. This happens because the usual implementation of the algorithm copies the function object internally:

```
template <class ForwIter, class Predicate>
ForwIter std::remove_if(ForwIter beg, ForwIter end, Predicate op)
{
    beg = find_if(beg, end, op);
    if (beg == end) {
        return beg;
    }
    else {
        ForwIter next = beg;
        return remove_copy_if(++next, end, beg, op);
    }
}
```

The algorithm uses find_if() to find the first element that should be removed. However, it then uses a copy of the passed function object to process the resulting elements (if any). Here, Nth is used again and removes also the sixth element. This behavior compromises the advantage of function objects being able to have a state. Without any cost it could be avoided (just implement it directly instead of calling find_if()).

Proposed resolution:

The standard should specify that this kind of implementation is a bug. Something like "it is guaranteed that an algorithm uses the same object for all calls of passed function objects (however, it may be a copy)".

25

---

94. May library implementors add template parameters to Standard Library classes?

Section: 17.4.4 lib.conforming **Status**: Open **Submitter:** Matt Austern **Date:** 22 Jan 98

Is it a permitted extension for library implementors to add template parameters to standard library classes, provided that those extra parameters have defaults? For example, instead of defining `template <class T, class Alloc = allocator<T> > class vector;` defining it as `template <class T, class Alloc = allocator<T>, int N = 1> class vector;`

The standard may well already allow this (I can't think of any way that this extension could break a conforming program, considering that users are not permitted to forward-declare standard library components), but it ought to be explicitly permitted or forbidden.

Proposed Resolution:

Add a new subclause [presumably 17.4.4.9] following 17.4.4.8 [lib.res.on.exception.handling]:

> 17.4.4.9 Template Parameters
>
> A specialization of a template class described in the C++ Standard Library behaves the same as if the implementation declares no additional template parameters.
>
> Footnote/ Additional template parameters with default values are thus permitted.

Add "template parameters" to the list of subclauses at the end of 17.4.4 paragraph 1 [lib.conforming].

---

96. Vector<bool> is not a container

Section: 23.2.5 lib.vector.bool **Status**: Open **Submitter:** AFNOR **Date:** 7 Oct 98

`vector<bool>` is not a container as its reference and pointer types are not references and pointers.

Also it forces everyone to have a space optimization instead of a speed one.

See also: 99-0008 == N1185 Vector<bool> is Nonconforming, Forces Optimization Choice.

Proposed Resolution:

---

98. Input iterator requirements are badly written

Section: 24.1.1 lib.input.iterators **Status**: Open **Submitter:** AFNOR **Date:** 7 Oct 98

Table 72 in 24.1.1 (lib.input.iterators) specifies semantics for `*r++` of:

```
{ T tmp = *r; ++r; return tmp; }
```

This does not work for pointers and overconstrains implementors.

Proposed Resolution:

Add for *r++: “To call the copy constructor for the type T is allowed but not required.”

---

102. Bug in insert range in associative containers

Section: 23.1.2 lib.associative.reqmts **Status**: Open **Submitter:** AFNOR **Date:** 7 Oct 98

Table 69 of Containers say that a.insert(i,j) is linear if [i, j) is ordered. It seems impossible to implement, as it means that if [i, j) = [x], insert in an associative container is O(1)!

Proposed Resolution:

N+log (size()) if [i,j) is sorted according to value_comp()

---

103. set::iterator is required to be modifiable, but this allows modification of keys

Section: 23.1.2 lib.associative.reqmts, 23.3.3 lib.set, 23.3.4 lib.mutliset **Status**: Open **Submitter:** AFNOR **Date:** 7 Oct 98

Set::iterator is described as implementation-defined with a reference to the container requirement; the container requirement says that const_iterator is an iterator pointing to const T and iterator an iterator pointing to T.

23.1.2 paragraph 2 implies that the keys should not be modified to break the ordering of elements. But that is not clearly specified. Especially considering that the current standard requires that iterator for associative containers be different from const_iterator. Set, for example, has the following:

```
typedef implementation defined iterator;
       // See _lib.container.requirements_
```

23.1 lib.container.requirements actually requires that iterator type pointing to T (table 65). Disallowing user modification of keys by changing the standard to require an iterator for associative container to be the same as const_iterator would be overkill since that will unnecessarily significantly restrict the usage of associative container. A class to be used as elements of set, for example, can no longer be modified easily without either redesigning the class (using mutable on fields that have nothing to do with ordering), or using const_cast, which defeats requiring iterator to be const_iterator. The proposed solution goes in line with trusting user knows what he is doing.

Proposed Resolution:

Option A. In 23.1.2 lib.associative.reqmts, paragraph 2, after first sentence, and before "In addition,...", add one line:

> Modification of keys shall not change their strict weak ordering.

Option B. Add three new sentences to 23.1.2 lib.associative.reqmts:

> At the end of paragraph 5: "Keys in an associative container are immutable." At the end of paragraph 6: "For associative containers where the value type is the same as the key type, both `iterator` and `const_iterator` are constant iterators. It is unspecified whether or not `iterator` and `const_iterator` are the same type."

Option C. To 23.1.2 lib.associative.reqmts, paragraph 3, which currently reads:

> The phrase ``equivalence of keys" means the equivalence relation imposed by the comparison and not the operator== on keys. That is, two keys k1 and k2 in the same container are considered to be equivalent if for the comparison object comp, comp(k1, k2) == false && comp(k2, k1) == false.

add the following:

> For any two keys k1 and k2 in the same container, comp(k1, k2) shall return the same value whenever it is evaluated. [Note: If k2 is removed from the container and later reinserted, comp(k1, k2) must still return a consistent value but this value may be different than it was the first time k1 and k2 were in the same container. This is intended to allow usage like a string key that contains a filename, where comp compares file contents; if k2 is removed, the file is changed, and the same k2 (filename) is reinserted, comp(k1, k2) must again return a consistent value but this value may be different than it was the previous time k2 was in the container.]

Rationale:

Simply requiring that keys be immutable is not sufficient, because the comparison object may indirectly (via pointers) operate on values outside of the keys. Furthermore, requiring that keys be immutable places undue restrictions on `set` for structures where only a portion of the structure participates in the comparison.

---

108. Lifetime of exception::what() return unspecified

Section: 18.6.1 lib.exception para 8, 9 **Status**: Review **Submitter:** AFNOR **Date:** 7 Oct 98

The lifetime of the return value of exception::what() is left unspecified. This issue has implications with exception safety of exception handling: some exceptions should not throw bad_alloc.

Proposed Resolution:

Add to 18.6.1 lib.exception paragraph 9 (exception::what notes clause) the sentence:

> The return value remains valid until the exception object from which it is obtained is destroyed or a non-const member function of the exception object is called.

---

109. Missing binders for non-const sequence elements

Section: 20.3.6 lib.binders **Status**: Open **Submitter:** Bjarne Stroustrup **Date:** 7 Oct 98

There are no versions of binders that apply to non-const elements of a sequence. This makes examples like for_each() using bind2nd() on page 521 of "The C++ Programming Language (3rd)" non-conforming. Suitable versions of the binders need to be added.

What is probably meant here is shown in the following example:

```
class Elem {
  public:
    void print (int i) const { }
    void modify (int i) { }
};

int main()
{
    vector<Elem> coll(2);
    for_each (coll.begin(), coll.end(), bind2nd(mem_fun_ref(&Elem::print),42));    // OK
    for_each (coll.begin(), coll.end(), bind2nd(mem_fun_ref(&Elem::modify),42));   // ERRO
}
```

The error results from the fact that bind2nd() passes its first argument (the argument of the sequence) as constant reference. See the following typical implementation:

```
template <class Operation>
class binder2nd
  : public unary_function<typename Operation::first_argument_type,
                          typename Operation::result_type> {
protected:
  Operation op;
  typename Operation::second_argument_type value;
public:
  binder2nd(const Operation& o,
            const typename Operation::second_argument_type& v)
      : op(o), value(v) {}

 typename Operation::result_type
  operator()(const typename Operation::first_argument_type& x) const {
    return op(x, value);
  }
};
```

The solution is to overload operator () of bind2nd for non-constant arguments:

```
template <class Operation>
class binder2nd
  : public unary_function<typename Operation::first_argument_type,
                          typename Operation::result_type> {
protected:
  Operation op;
  typename Operation::second_argument_type value;
public:
  binder2nd(const Operation& o,
            const typename Operation::second_argument_type& v)
      : op(o), value(v) {}

 typename Operation::result_type
  operator()(const typename Operation::first_argument_type& x) const {
    return op(x, value);
  }
  typename Operation::result_type
  operator()(typename Operation::first_argument_type& x) const {
    return op(x, value);
  }
};
```

Proposed Resolution:

In 20.3.6.1 [lib.binders.1st] in the declaration of binder1st after:

```
typename Operation::result_type
 operator()(const typename Operation::second_argument_type& x) const;
```

insert:

```
typename Operation::result_type
 operator()(typename Operation::second_argument_type& x) const;
```

In 20.3.6.3 [lib.binders.2nd] in the declaration of binder2nd after:

```
typename Operation::result_type
 operator()(const typename Operation::first_argument_type& x) const;
```

insert:

```
typename Operation::result_type
 operator()(typename Operation::first_argument_type& x) const;
```

---

111. istreambuf_iterator::equal overspecified, inefficient

Section: 24.5.3.5 [lib.istreambuf.iterator::equal] **Status**: Open **Submitter:** Nathan Myers **Date:** 15 Oct 98

The member istreambuf_iterator<>::equal is specified to be unnecessarily inefficient. While this does not affect the efficiency of conforming implementations of iostreams, because they can "reach into" the iterators and bypass this function, it does affect users who use istreambuf_iterators.

The inefficiency results from a too-scrupulous definition, which requires a "true" result if neither iterator is at eof. In practice these iterators can only usefully be compared with the "eof" value, so the extra test implied provides no benefit, but slows down users' code.

The solution is to weaken the requirement on the function to return true only if both iterators are at eof.

Proposed Resolution:

Replace 24.5.3.5 [lib.istreambuf.iterator::equal], paragraph 1,

> -1- Returns: true if and only if both iterators are at end-of-stream, or neither is at end-of-stream, regardless of what streambuf object they use.

with

> -1- Returns: true if and only if both iterators are at end-of-stream, regardless of what streambuf object they use.

---

112. Minor typo in `ostreambuf_iterator` constructor

Section: 24.5.4.1 lib.ostreambuf.iter.cons **Status**: Review **Submitter:** Matt Austern **Date:** 20 Oct 98

The **requires** clause for `ostreambuf_iterator`'s constructor from an `ostream_type` (24.5.4.1, paragraph 1) reads "*s* is not null". However, *s* is a reference, and references can't be null.

Proposed Resolution:

In 24.5.4.1 lib.ostreambuf.iter.cons:

Move the current paragraph 1, which reads "Requires: s is not null.", from the first constructor to the second constructor.

Insert a new paragraph 1 Requires clause for the first constructor reading:

> **Requires**: `s.rdbuf()` is not null.

---

114. Placement forms example in error twice

Section: 18.4.1.3 [lib.new.delete.placement] **Status**: Open **Submitter:** Steve Clamage **Date:** 28 Oct 1998

Section 18.4.1.3 contains the following example:

```
[Example: This can be useful for constructing an object at a known address:
        char place[sizeof(Something)];
        Something* p = new (place) Something();
 -end example]
```

First code line: "place" need not have any special alignment, and the following constructor could fail due to misaligned data.

Second code line: Aren't the parens on Something() incorrect? [Dublin: the LWG believes the () are correct.]

Examples are not normative, but nevertheless should not show code that is invalid or likey to fail.

Proposed Resolution:

Replace the first line of code in the example in 18.4.1.3 [lib.new.delete.placement] with:

```
void* place = operator new(sizeof(Something));
```

---

115. Typo in strstream constructors

Section: D.7.4.1 [depr.strstream.cons] **Status**: Review **Submitter:** Steve Clamage **Date:** 2 Nov 1998

D.7.4.1 strstream constructors paragraph 2 says:

> Effects: Constructs an object of class strstream, initializing the base class with iostream(& sb) and initializing sbwith one of the two constructors:
>
> - If mode&app==0, then s shall designate the first element of an array of n elements. The constructor is strstreambuf(s, n, s).

- If mode&app==0, then s shall designate the first element of an array of n elements that contains an NTBS whose first element is designated by s. The constructor is strstreambuf(s, n, s+std::strlen(s)).

Notice the second condition is the same as the first. I think the second condition should be "If mode&app==app", or "mode&app!=0", meaning that the append bit is set.

Proposed Resolution:

In D.7.3.1 [depr.ostrstream.cons] paragraph 2 and D.7.4.1 [depr.strstream.cons] paragraph 2, change the first condition to `(mode&app)==0` and the second condition to `(mode&app)!=0`.

---

117. `basic_ostream` uses nonexistent `num_put` member functions

Section: 27.6.2.5.2 lib.ostream.inserters.arithmetic **Status**: Review **Submitter:** Matt Austern **Date:** 20 Nov 98

The **effects** clause for numeric inserters says that insertion of a value `x`, whose type is either `bool`, `short`, `unsigned short`, `int`, `unsigned int`, `long`, `unsigned long`, `float`, `double`, `long double`, or `const void*`, is delegated to `num_put`, and that insertion is performed as if through the following code fragment:

```
bool failed = use_facet<
   num_put<charT,ostreambuf_iterator<charT,traits> >
   >(getloc()).put(*this, *this, fill(), val). failed();
```

This doesn't work, because `num_put<>`::put is only overloaded for the types `bool`, `long`, `unsigned long`, `double`, `long double`, and `const void*`. That is, the code fragment in the standard is incorrect (it is diagnosed as ambiguous at compile time) for the types `short`, `unsigned short`, `int`, `unsigned int`, and `float`.

We must either add new member functions to `num_put`, or else change the description in `ostream` so that it only calls functions that are actually there. I prefer the latter.

Proposed Resolution:

Replace 27.6.2.5.2, paragraph 1 with the following:

> The classes `num_get<>` and `num_put<>` handle localedependent numeric formatting and parsing. These inserter functions use the imbued `locale` value to perform numeric formatting. When `val` is of type `bool`, `long`, `unsigned long`, `double`, `long double`, or `const void*`, the formatting conversion occurs as if it performed the following code fragment:
>
> ```
> bool failed = use_facet<
>    num_put<charT,ostreambuf_iterator<charT,traits> >
>    >(getloc()).put(*this, *this, fill(), val). failed();
> ```
>
> When `val` is of type `short` or `int` the formatting conversion occurs as if it performed the following code fragment:
>
> ```
> bool failed = use_facet<
>    num_put<charT,ostreambuf_iterator<charT,traits> >
>    >(getloc()).put(*this, *this, fill(), static_cast<long>(val)). failed();
> ```
>
> When `val` is of type `unsigned short` or `unsigned int` the formatting conversion occurs as if it performed the following code fragment:
>
> ```
> bool failed = use_facet<
> ```

```
num_put<charT,ostreambuf_iterator<charT,traits> >
>(getloc()).put(*this, *this, fill(), static_cast<unsigned long>(val)). failed();
```

When `val` is of type `float` the formatting conversion occurs as if it performed the following code fragment:

```
bool failed = use_facet<
   num_put<charT,ostreambuf_iterator<charT,traits> >
   >(getloc()).put(*this, *this, fill(), static_cast<double>(val)). failed();
```

---

118. `basic_istream` uses nonexistent `num_get` member functions

Section: 27.6.1.2.2 lib.istream.formatted.arithmetic **Status**: Open **Submitter:** Matt Austern **Date:** 20 Nov 98

Formatted input is defined for the types `short`, `unsigned short`, `int`, `unsigned int`, `long`, `unsigned long`, `float`, `double`, `long double`, `bool`, and `void*`. According to section 27.6.1.2.2, formatted input of a value `x` is done as if by the following code fragment:

```
typedef num_get< charT,istreambuf_iterator<charT,traits> > numget;
iostate err = 0;
use_facet< numget >(loc).get(*this, 0, *this, err, val);
setstate(err);
```

According to section 22.2.2.1.1 lib.facet.num.get.members, however, `num_get<>::get()` is only overloaded for the types `bool`, `long`, `unsigned short`, `unsigned int`, `unsigned long`, `unsigned long`, `float`, `double`, `long double`, and `void*`. Comparing the lists from the two sections, we find that 27.6.1.2.2 is using a nonexistent function for types `short` and `int`.

Proposed Resolution:

Add `short` and `int` overloads for `num_get<>::get()`

---

119. Should virtual functions be allowed to strengthen the exception specification?

Section: 17.4.4.8 lib.res.on.exception.handling **Status:** Ready **Submitter:** Judy Ward **Date:** 15 Dec 1998

Section 17.4.4.8 lib.res.on.exception.handling states:

"An implementation may strengthen the exception-specification for a function by removing listed exceptions."

The problem is that if an implementation is allowed to do this for virtual functions, then a library user cannot write a class that portably derives from that class.

For example, this would not compile if ios_base::failure::~failure had an empty exception specification:

```
#include <ios>
#include <string>

class D : public std::ios_base::failure {
public:
        D(const std::string&);
        ~D(); // error - exception specification must be compatible with
              // overridden virtual function ios_base::failure::~failure()
};
```

Proposed Resolution:

Change Section 17.4.4.8 lib.res.on.exception.handling from:

"may strengthen the exception-speciification for a function"

to:

"may strengthen the exception-specification for a non-virtual function".

---

120. Can an implementor add specializations?

Section: 17.4.3.1 lib.reserved.names **Status:** Open **Submitter:** Judy Ward **Date:** 15 Dec 1998

Section 17.4.3.1 says:

> It is undefined for a C++ program to add declarations or definitions to namespace std or namespaces within namespace std unless otherwise specified. A program may add template specializations for any standard library template to namespace std. Such a specialization (complete or partial) of a standard library template results in undefined behavior unless the declaration depends on a user-defined name of external linkage and unless the specialization meets the standard library requirements for the original template...

This implies that it is ok for library users to add specializations, but not implementors. A user program can actually detect this, for example, the following manual instantiation will not compile if the implementor has made ctype<wchar_t> a specialization:

```
#include <locale>
#include <wchar.h>

template class std::ctype<wchar_t>; // can't be specialization
```

Lib-7047 comments:

The status quo is unclear, and probably contradictory. This issue applies both the explicit instantiations and to specializations, since it is not permitted to provide both a specialization and an explicit instantiation.

The specialization issue is actually more serious than the instantiation one. One could argue that there is a consistent status quo as far as instantiations go, but one can't argue that in the case of specializations. The standard must either (1) give library implementors license to provide explicit specializations of any library template; or (2) give a complete list of exactly which specializations must be provided, and forbid library implementors from providing any specializations not on that list. At present the standard does neither.

Proposed Resolution:

Add to 17.4.4 lib.conforming a section called Specializations with wording:

> An implementation can define additional specializations for any of the template classes or functions in the standard library if a use of any of these classes or functions behaves as if the implementation did not define them.

---

121. Detailed definition for ctype<wchar_t> specialization missing

Section: 22.1.1.1.1 lib.locale.category **Status:** Open **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.

---

122. streambuf/wstreambuf description should not say they are specializations

Section: 27.5.2 lib.streambuf **Status:** Open **Submitter:** Judy Ward **Date:** 15 Dec 1998

Section 27.5.2 describes the streambuf classes this way:

> The class streambuf is a specialization of the template class basic_streambuf specialized for the type char.
>
> The class wstreambuf is a specialization of the template class basic_streambuf specialized for the type wchar_t.

This implies that these classes must be template specializations, not typedefs.

It doesn't seem this was intended, since Section 27.5 has them declared as typedefs.

Proposed Resolution:

Remove 27.5.2 lib.streambuf paragraphs 2 and 3 (the two above sentences).

Rationale:

The `streambuf` synopsis already has a declaration for the typedefs.

---

123. Should valarray helper arrays fill functions be const?

Section: 26.3.5.4 lib.slice.arr.fill, 26.3.7.4 lib.gslice.array.fill, 26.3.8.4 lib.mask.array.fill, 26.3.9.4 lib.indirect.array..fill **Status:** Open **Submitter:** Judy Ward **Date:** 15 Dec 1998

One of the operator= in the valarray helper arrays is const and one is not. For example, look at slice_array. This operator= in Section 26.3.5.2 lib.slice.arr.assign is const:

```
void operator=(const valarray<T>&) const;
```

but this one in Section 26.3.5.4 lib.slice.arr.fill, is not:

```
void operator=(const T&);
```

The description of the semantics for these two functions is similar.

Proposed Resolution:

Make the `operator=(const T&)` versions of slice_array, gslice_array, indirect_array, and mask_array `const` member functions.

---

127. auto_ptr<> conversion issues

Section: 20.4.5 lib.auto.ptr **Status**: Open **Submitter:** Greg Colvin **Date:** 17 Feb 99

There are two problems with the current `auto_ptr` wording in the standard:

First, the `auto_ptr_ref` definition cannot be nested because `auto_ptr<Derived>::auto_ptr_ref` is unrelated to `auto_ptr<Base>::auto_ptr_ref`.

Second, there is no `auto_ptr` assignment operator taking an `auto_ptr_ref` argument.

I have discussed these problems with my proposal coauthor, Bill Gibbons, and with some compiler and library implementers, and we believe that these problems are not desired or desirable implications of the standard.

25 Aug 99: The proposed resolution now reflects changes; 1) changed "assignment operator" to "public assignment operator", 2) changed effects to specify use of release(), 3) made the conversion to auto_ptr_ref const.

2 Feb 00: Lisa Lippincott comments: [The resolution of] this issue states that the conversion from auto_ptr to auto_ptr_ref should be const. This is not acceptable, because it would allow initialization and assignment from _any_ const auto_ptr! It also introduces an implementation difficulty in writing this conversion function -- namely, somewhere along the line, a const_cast will be necessary to remove that const so that release() may be called. This may result in undefined behavior > [7.1.5.1/4]. The conversion operator does not have to be const, because a non-const implicit object parameter may be bound to an rvalue [13.3.3.1.4/3] [13.3.1/5].

Proposed Resolution:

In 20.4.5 lib.auto.ptr, paragraph 2, move the `auto_ptr_ref` definition to namespace scope.

In 20.4.5 lib.auto.ptr, paragraph 2, add a public assignment operator to the `auto_ptr` definition:

```
auto_ptr& operator=(auto_ptr_ref<X> r) throw();
```

Also add the assignment operator to 20.4.5.3 lib.auto.ptr.conv:

```
auto_ptr& operator=(auto_ptr_ref<X> r) throw()

Effects: Calls reset(p.release()) for the auto_ptr p that r holds a reference to.
Returns: *this.
```

In 20.4.5 lib.auto.ptr, paragraph 2, and 20.4.5.3 lib.auto.ptr.conv, paragraph 2, make the conversion to auto_ptr_ref const:

```
template<class Y> operator auto_ptr_ref<Y>() const throw();
```

---

129. Need error indication from seekp() and seekg()

Section: 27.6.1.3 lib.istream.unformatted and 27.6.2.4 lib.istream.seeks **Status**: Review **Submitter:** Angelika Langer **Date:** February 22, 1999

Currently, the standard does not specify how seekg() and seekp() indicate failure. They are not required to set failbit, and they can't return an error indication because they must return *this, i.e. the stream. Hence, it is undefined what happens if they fail. And they _can_ fail, for instance, when a file stream is disconnected from the underlying file (is_open()==false) or when a wide charaacter file stream must perform a state-dependent code conversion, etc.

The stream functions seekg() and seekp() should set failbit in the stream state in case of failure.

Proposed Resolution:

Add to the Effects: clause of seekg() in 27.6.1.3 lib.istream.unformatted and to the Effects: clause of seekp() in 27.6.2.4 lib.istream.seeks:

> In case of failure, the function calls `setstate(failbit)` (which may throw `ios_base::failure`).

---

134. vector and deque constructors over specified

Section: 23.2.4.1 lib.vector.cons **Status:** Open **Submitter:** Howard Hinnant **Date:** 6 Mar 99

The complexity description says: "It does at most 2N calls to the copy constructor of T and logN reallocations if they are just input iterators ...".

This appears to be overly restrictive, dictating the precise memory/performance tradeoff for the implementor.

Proposed Resolution:

Change 23.2.1.1, paragraph 6 to:

-6- Complexity: If the iterators first and last are forward iterators, bidirectional iterators, or random access iterators the constructor makes only N calls to the copy constructor, and performs no reallocations, where N is last - first. It makes order N calls to the copy constructor of T and order log N reallocations if they are input iterators.*

And change 23.2.4.1, paragraph 1 to:

-1- Complexity: The constructor template <class InputIterator> vector(InputIterator first, InputIterator last) makes only N calls to the copy constructor of T (where N is the distance between first and last) and no reallocations if iterators first and last are of forward, bidirectional, or random access categories. It makes order N calls to the copy constructor of T and order logN reallocations if they are just input iterators, since it is impossible to determine the distance between first and last and then do copying.

---

136. seekp, seekg setting wrong streams?

Section: 27.6.1.3 lib.istream.unformatted **Status:** Open **Submitter:** Howard Hinnant **Date:** 6 Mar 99

I may be misunderstanding the intent, but should not seekg set only the input stream and seekp set only the output stream? The description seems to say that each should set both input and output streams. If that's really the intent, I withdraw this proposal.

Proposed Resolution:

In section 27.6.1.3 change:

```
basic_istream<charT,traits>& seekg(pos_type pos);
Effects: If fail() != true, executes rdbuf()->pubseekpos(pos).
```

To:

```
basic_istream<charT,traits>& seekg(pos_type pos);
Effects: If fail() != true, executes rdbuf()->pubseekpos(pos, ios_base::in).
```

In section 27.6.1.3 change:

```
basic_istream<charT,traits>& seekg(off_type& off, ios_base::seekdir dir);
Effects: If fail() != true, executes rdbuf()->pubseekoff(off, dir).
```

To:

```
basic_istream<charT,traits>& seekg(off_type& off, ios_base::seekdir dir);
Effects: If fail() != true, executes rdbuf()->pubseekoff(off, dir, ios_base::in).
```

In section 27.6.2.4, paragraph 2 change:

```
-2- Effects: If fail() != true, executes rdbuf()->pubseekpos(pos).
```

To:

```
-2- Effects: If fail() != true, executes rdbuf()->pubseekpos(pos, ios_base::out).
```

In section 27.6.2.4, paragraph 4 change:

```
-4- Effects: If fail() != true, executes rdbuf()->pubseekoff(off, dir).
```

To:

```
-4- Effects: If fail() != true, executes rdbuf()->pubseekoff(off, dir, ios_base::out).
```

---

137. Do use_facet and has_facet look in the global locale?

Section: 22.1.1 lib.locale **Status**: Open **Submitter:** Angelika Langer **Date:** March 17, 1999

Section 22.1.1 lib.locale says:

-4- In the call to use_facet<Facet>(loc), the type argument chooses a facet, making available all members of the named type. If Facet is not present in a locale (or, failing that, in the global locale), it throws the standard exception bad_cast. A C++ program can check if a locale implements a particular facet with the template function has_facet<Facet>().

This contradicts the specification given in section 22.1.2 lib.locale.global.templates:

template <class Facet> const Facet& use_facet(const locale& loc);

-1- Get a reference to a facet of a locale.
-2- Returns: a reference to the corresponding facet of loc, if present.
-3- Throws: bad_cast if has_facet<Facet>(loc) is false.
-4- Notes: The reference returned remains valid at least as long as any copy of loc exists

Proposed Resolution:

If there's consensus that section 22.1.2 reflects the intent, then the phrase:

(or, failing that, in the global locale)

should be removed from section 22.1.1.

---

138. Class ctype_byname<char> redundant and misleading

Section: 22.2.1.4 lib.locale.ctype.byname.special **Status**: Open **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.

Proposed Resolution:

Delete section 22.2.1.4 lib.locale.ctype.byname.special

---

141. basic_string::find_last_of, find_last_not_of say pos instead of xpos

Section: 21.3.6.4 lib.string::find.last.of, 21.3.6.6 lib.string::find.last.not.of **Status**: Ready **Submitter:** Arch Robison **Date:** 28 Apr 99

Sections 21.3.6.4 paragraph 1 and 21.3.6.6 paragraph 1 surely have misprints where they say:

— `xpos <= pos` and `pos < size();`

Surely the document meant to say ``xpos < size()'' in both places.

Proposed Resolution:

Change Sections 21.3.6.4 paragraph 1 and 21.3.6.6 paragraph 1, the line which says:

— `xpos <= pos` and `pos < size();`

to:

— `xpos <= pos` and `xpos < size();`

---

142. lexicographical_compare complexity wrong

Section: 25.3.8 lib.alg.lex.comparison **Status**: Review **Submitter:** Howard Hinnant **Date:** 20 Jun 99

The lexicographical_compare complexity is specified as:

> "At most min((last1 - first1), (last2 - first2)) applications of the corresponding comparison."

The best I can do is twice that expensive.

Proposed Resolution:

Change 25.3.8 [lib.alg.lex.comparison] complexity to:

> At most `2*min((last1 - first1), (last2 - first2))` applications of the corresponding comparison.

Change the example at the end of paragraph 3 to read:

> [Example:
>
> ```
> for ( ; first1 != last1 && first2 != last2 ; ++first1, ++first2) {
>   if (*first1 < *first2) return true;
>   if (*first2 < *first1) return false;
> }
> return first1 == last1 && first2 != last2;
> ```
>
> --end example]

---

143. C .h header wording unclear

Section: D.5 depr.c.headers **Status**: Open **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.

-
-

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;
}
```

Proposed Resolution:

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.

---

144. Deque constructor complexity wrong

Section: 23.2.1.1 lib.deque.cons **Status**: Ready **Submitter:** Herb Sutter **Date:** 9 May 99

In 23.2.1.1 paragraph 6, the deque ctor that takes an iterator range appears to have complexity requirements which are incorrect, and which contradict the complexity requirements for insert(). I suspect that the text in question, below, was

taken from vector:

> Complexity: If the iterators first and last are forward iterators, bidirectional iterators, or random access iterators the constructor makes only N calls to the copy constructor, and performs no reallocations, where N is last - first.

The word "reallocations" does not really apply to deque. Further, all of the following appears to be spurious:

> It makes at most 2N calls to the copy constructor of T and log N reallocations if they are input iterators.1)
>
> 1) The complexity is greater in the case of input iterators because each element must be added individually: it is impossible to determine the distance between first abd last before doing the copying.

This makes perfect sense for vector, but not for deque. Why should deque gain an efficiency advantage from knowing in advance the number of elements to insert?

Proposed Resolution:

In 23.2.1.1 paragraph 6, replace the Complexity description, including the footnote, with the following text (which also corrects the "abd" typo):

> Complexity: Makes last - first calls to the copy constructor of T.

---

146. complex<T> Inserter and Extractor need sentries

Section: 26.2.6 lib.complex.ops **Status**: Review **Submitter:** Angelika Langer **Date:**12 May 99

The extractor for complex numbers is specified as:

> template<class T, class charT, class traits>
> basic_istream<charT, traits>&
> operator>>(basic_istream<charT, traits>& is, complex<T>& x);
>
> Effects: Extracts a complex number x of the form: u, (u), or (u,v), where u is the real part and v is the imaginary part (lib.istream.formatted).
> Requires: The input values be convertible to T. If bad input is encountered, calls is.setstate(ios::failbit) (which may throw ios::failure (lib.iostate.flags).
> Returns: is .

Is it intended that the extractor for complex numbers does not skip whitespace, unlike all other extractors in the standard library do? Shouldn't a sentry be used?

The inserter for complex numbers is specified as:

> template<class T, class charT, class traits>
> basic_ostream<charT, traits>&
> operator<<(basic_ostream<charT, traits>& o, const complex<T>& x);
>
> Effects: inserts the complex number x onto the stream o as if it were implemented as follows:
>
> template<class T, class charT, class traits>
> basic_ostream<charT, traits>&
> operator<<(basic_ostream<charT, traits>& o, const complex<T>& x)
> {

```
basic_ostringstream<charT, traits> s;
s.flags(o.flags());
s.imbue(o.getloc());
s.precision(o.precision());
s << '(' << x.real() << "," << x.imag() << ')';
return o << s.str();
}
```

Is it intended that the inserter for complex numbers ignores the field width and does not do any padding? If, with the suggested implementation above, the field width were set in the stream then the opening parentheses would be adjusted, but the rest not, because the field width is reset to zero after each insertion.

I think that both operations should use sentries, for sake of consistency with the other inserters and extractors in the library. Regarding the issue of padding in the inserter, I don't know what the intent was.

Proposed Resolution:

After 26.2.6 lib.complex.ops paragraph 14 (operator>>), add a Notes clause:

> Notes: This extraction is performed as a series of simpler extractions. Therefore, the skipping of whitespace is specified to be the same for each of the simpler extractions.

Rationale:

For extractors, the note is added to make it clear that skipping whitespace follows an "all-or-none" rule.

For inserters, the LWG believes there is no defect; the standard is correct as written.

---

147. Library Intro refers to global functions that aren't global

Section: 17.4.4.3 lib.global.functions **Status**: Review **Submitter:** Lois Goldthwaite **Date:** 4 Jun 99

The library had many global functions until 17.4.1.1 [lib.contents] paragraph 2 was added:

> All library entities except macros, operator new and operator delete are defined within the namespace std or namespaces nested within namespace std.

It appears "global function" was never updated in the following:

> 17.4.4.3 - Global functions [lib.global.functions]
>
> -1- It is unspecified whether any global functions in the C++ Standard Library are defined as inline (dcl.fct.spec).
>
> -2- A call to a global function signature described in Clauses lib.language.support through lib.input.output behaves the same as if the implementation declares no additional global function signatures.*
>
> [Footnote: A valid C++ program always calls the expected library global function. An implementation may also define additional global functions that would otherwise not be called by a valid C++ program. --- end footnote]
>
> -3- A global function cannot be declared by the implementation as taking additional default arguments.

17.4.4.4 - Member functions [lib.member.functions]

-2- An implementation can declare additional non-virtual member function signatures within a class:

-- by adding arguments with default values to a member function signature; The same latitude does not extend to the implementation of virtual or global functions, however.

Proposed Resolution:

Change "global" to "global or non-member" in:

17.4.4.3 [lib.global.functions] section title,
17.4.4.3 [lib.global.functions] para 1,
17.4.4.3 [lib.global.functions] para 2 in 2 places plus 2 places in the footnote,
17.4.4.3 [lib.global.functions] para 3,
17.4.4.4 [lib.member.functions] para 2

---

148. Functions in the example facet BoolNames should be const

Section: 22.2.8 lib.facets.examples **Status**: Ready **Submitter:** Jeremy Siek **Date:** 3 Jun 99

In 22.2.8 [lib.facets.examples] paragraph 13, the do_truename() and do_falsename() functions in the example facet BoolNames should be const. The functions they are overriding in numpunct_byname<char> are const.

Proposed Resolution:

In 22.2.8 [lib.facets.examples] paragraph 13, insert "const" in two places:

```
string do_truename() const { return "Oui Oui!"; }
string do_falsename() const { return "Mais Non!"; }
```

---

150. Find_first_of says integer instead of iterator

Section: 25.1.4 lib.alg.find.first.of **Status**: Ready **Submitter:** Matt McClure **Date:** 30 Jun 99

Proposed Resolution:

Change 25.1.4 lib.alg.find.first.of paragraph 2 from:

Returns: The first iterator i in the range [first1, last1) such that for some integer j in the range [first2, last2) ...

to:

Returns: The first iterator i in the range [first1, last1) such that for some iterator j in the range [first2, last2) ...

---

151. Can't currently clear() empty container

Section: 23.1.1 lib.sequence.reqmts **Status**: Ready **Submitter:** Ed Brey **Date:** 21 Jun 99

For both sequences and associative containers, a.clear() has the semantics of erase(a.begin(),a.end()), which is undefined for an empty container since erase(q1,q2) requires that q1 be dereferenceable (23.1.1,3 and 23.1.2,7). When the container is empty, a.begin() is not dereferenceable.

The requirement that q1 be unconditionally dereferenceable causes many operations to be intuitively undefined, of which clearing an empty container is probably the most dire.

Since q1 and q2 are only referenced in the range [q1, q2), and [q1, q2) is required to be a valid range, stating that q1 and q2 must be iterators or certain kinds of iterators is unnecessary.

Proposed Resolution:

In 23.1.1, paragraph 3, change:

> p and q2 denote valid iterators to a, q and q1 denote valid dereferenceable iterators to a, [q1, q2) denotes a valid range

to:

> p denotes a valid iterator to a, q denotes a valid dereferenceable iterator to a, [q1, q2) denotes a valid range in a

In 23.1.2, paragraph 7, change:

> p and q2 are valid iterators to a, q and q1 are valid dereferenceable iterators to a, [q1, q2) is a valid range

to

> p is a valid iterator to a, q is a valid dereferenceable iterator to a, [q1, q2) is a valid range into a

---

152. Typo in `scan_is()` semantics

Section:: 22.2.1.1.2 lib.locale.ctype.virtuals **Status:** Ready **Submitter:** Dietmar Kühl **Date:** 20 Jul 99

The semantics of `scan_is()` (paragraphs 4 and 6) is not exactly described because there is no function `is()` which only takes a character as argument. Also, in the effects clause (paragraph 3), the semantic is also kept vague.

Proposed resolution:

In 22.2.1.1.2 lib.locale.ctype.virtuals paragraphs 4 and 6, change the returns clause from:

> "... such that `is(*p)` would..."

to: "... such that `is(m, *p)` would...."

---

153. Typo in `narrow()` semantics

Section:: 22.2.1.3.2 lib.facet.ctype.char.members **Status:** Open **Submitter:** Dietmar Kühl **Date:** 20 Jul 99

The description of the array version of `narrow()` (in paragraph 11) is flawed: There is no member `do_narrow()` which takes only there arguments because in addition to the range a default character is needed.

Proposed resolution:

Change 22.2.1.3.2 lib.facet.ctype.char.members `narrow()` (in paragraph 10) by removing the comments around **`dfault`** (2 places).

Change 22.2.1.3.2 lib.facet.ctype.char.members `narrow()` (in paragraph 11) returns clause to:

> Returns: `do_narrow(low, high, dfault, to)`

---

154. Missing `double` specifier for `do_get()`

Section:: 22.2.2.1.2 lib.facet.num.get.virtuals **Status:** Ready **Submitter:** Dietmar Kühl **Date:** 20 Jul 99

The table in paragraph 7 for the length modifier does not list the length modifier `l` to be applied if the type is `double`. Thus, the standard asks the implementation to do undefined things when using `scanf()` (the missing length modifier for `scanf()` when scanning `doubles` is actually a problem I found quite often in production code, too).

Proposed resolution:

In 22.2.2.1.2 lib.facet.num.get.virtuals, paragraph 7, add a row in the Length Modifier table to say that for `double` a length modifier `l` is to be used.

Rationale:

The standard mades an embarrassing beginner's mistake.

---

155. Typo in naming the class defining the class `Init`

Section:: 27.3 lib.iostream.objects **Status:** Ready **Submitter:** Dietmar Kühl **Date:** 20 Jul 99

There are conflicting statements about where the class `Init` is defined. According to 27.3 (lib.iostream.objects) paragraph 2 it is defined as `basic_ios::Init`, according to 27.4.2 (lib.ios.base) it is defined as `ios_base::Init`.

Proposed resolution:

Change 27.3 (lib.iostream.objects) paragraph 2 from "`basic_ios::Init`" to "`ios_base::Init`".

Rationale:

Although not strictly wrong, the standard was misleading enough to warrant the change.

---

156. Typo in `imbue()` description

Section:: 27.4.2.3 lib.ios.base.locales **Status:** Ready **Submitter:** Dietmar Kühl **Date:** 20 Jul 99

There is a small discrepancy between the declarations of `imbue()`: in 27.4.2 (lib.ios.base) the argument is passed as `locale const&` (correct), in 27.4.2.3 (lib.ios.base.locales) it is passed as `locale const` (wrong).

Proposed resolution:

In 27.4.2.3 (lib.ios.base.locales) change the `imbue` argument from `"locale const" to "locale const&"`.

---

158. Underspecified semantics for `setbuf()`

Section:: 27.5.2.4.2 lib.streambuf.virt.buffer **Status:** Ready **Submitter:** Dietmar Kühl **Date:** 20 Jul 99

The default behavior of `setbuf()` is described only for the situation that `gptr() != 0 && gptr() != egptr()`: namely to do nothing. What has to be done in other situations is not described although there is actually only one reasonable approach, namely to do nothing, too.

Since changing the buffer would almost certainly mess up most buffer management of derived classes unless these classes do it themselves, the default behavior of `setbuf()` should always be to do nothing.

Proposed resolution:

Change 27.5.2.4.2 lib.streambuf.virt.buffer, paragraph 3, Default behavior, to: "Default behavior: Does nothing. Returns this."

---

159. Strange use of `underflow()`

Section:: 27.5.2.4.3 lib.streambuf.virt.get **Status:** Review **Submitter:** Dietmar Kühl **Date:** 20 Jul 99

The description of the meaning of the result of `showmanyc()` seems to be rather strange: It uses calls to `underflow()`. Using `underflow()` is strange because this function only reads the current character but does not extract it, `uflow()` would extract the current character. This should be fixed to use `sbumpc()` instead.

Proposed resolution:

Change 27.5.2.4.3 lib.streambuf.virt.get paragraph 1, `showmanyc()`returns clause, by replacing the word "supplied" with the words "extracted from the stream".

---

160. Typo: Use of non-existing function `exception()`

Section:: 27.6.1.1 lib.istream **Status:** Ready **Submitter:** Dietmar Kühl **Date:** 20 Jul 99

The paragraph 4 refers to the function `exception()` which is not defined. Probably, the referred function is `basic_ios::exceptions()`.

Proposed resolution:

In 27.6.1.1 lib.istream change `"exception()" to "basic_ios::exceptions()"`.

---

161. Typo: `istream_iterator` vs. `istreambuf_iterator`

Section:: 27.6.1.2.2 lib.istream.formatted.arithmetic **Status:** Ready **Submitter:** Dietmar Kühl **Date:** 20 Jul 99

The note in the second paragraph pretends that the first argument is an object of type `istream_iterator`. This is wrong: It is an object of type `istreambuf_iterator`.

Proposed resolution:

Change 27.6.1.2.2 lib.istream.formatted.arithmetic from:

> The first argument provides an object of the istream_iterator class...

to

> The first argument provides an object of the istreambuf_iterator class...

---

164. do_put() has apparently unused fill argument

Section:: 22.2.5.3.2 lib.locale.time.put.virtuals **Status:** Review **Submitter:** Angelika Langer **Date:** 23 Jul 99

In [lib.locale.time.put.virtuals] the do_put() function is specified as taking a fill character as an argument, but the description of the function does not say whether the character is used at all and, if so, in which way. The same holds for any format control parameters that are accessible through the ios_base& argument, such as the adjustment or the field width. Is strftime() supposed to use the fill character in any way? In any case, the specification of time_put.do_put() looks inconsistent to me.

Is the signature of do_put() wrong, or is the effects clause incomplete?

Proposed resolution:

Add the following note after 22.2.5.3.2 lib.locale.time.put.virtuals paragraph 2:

> [Note: the `fill` argument may be used in the implementation-defined formats, or by derivations. A space character is a reasonable default for this argument. --end Note]

Rationale:

The LWG felt that while the normative text was correct, users need some guidance on what to pass for the `fill` argument since the standard doesn't say how it's used.

---

165. `xsputn()`, `pubsync()` never called by `basic_ostream` members?

Section:: 27.6.2.1 lib.ostream **Status:** Open **Submitter:** Dietmar Kühl **Date:** 20 Jul 99

Paragraph 2 explicitly states that none of the `basic_ostream` functions falling into one of the groups "formatted output functions" and "unformatted output functions" calls any stream buffer function which might call a virtual function other than `overflow()`. Basically this is fine but this implies that `sputn()` (this function would call the

virtual function `xsputn()`) is never called by any of the standard output functions. Is this really intended? At minimum it would be convenient to call `xsputn()` for strings... Also, the statement that `overflow()` is the only virtual member of `basic_streambuf` called is in conflict with the definition of `flush()` which calls `rdbuf()->pubsync()` and thereby the virtual function `sync()` (`flush()` is listed under "unformatted output functions").

In addition, I guess that the sentence starting with "They may use other public members of `basic_ostream` ..." probably was intended to start with "They may use other public members of `basic_streamuf`..." although the problem with the virtual members exists in both cases.

Proposed resolution:

I see two obvious resolutions:

1. state in a footnote that this means that `xsputn()` will never be called by any ostream member and that this is intended.
2. relax the restriction and allow calling `overflow()` and `xsputn()`. Of course, the problem with `flush()` has to be resolved in some way.

---

167. Improper use of `traits_type::length()`

Section:: 27.6.2.5.4 lib.ostream.inserters.character **Status:** Open **Submitter:** Dietmar Kühl **Date:** 20 Jul 99

Paragraph 4 states that the length is determined using `traits::length(s)`. Unfortunately, this function is not defined for example if the character type is `wchar_t` and the type of `s` is `char const*`. Similar problems exist if the character type is `char` and the type of `s` is either `signed char const*` or `unsigned char const*`.

Proposed resolution:

Make the case where `s` is of type a different type than `typename traits::char_type const*` a special case, where eg. `std::char_traits<...>::length()` is used (with the ... replaced by the correct type, of course) However, this resolution would require that `char_traits` is specialized for `signed char` and `unsigned char` which is currently not the case, I think.

---

168. Type: formatted vs. unformatted

Section:: 27.6.2.6 lib.ostream.unformatted **Status:** Ready **Submitter:** Dietmar Kühl **Date:** 20 Jul 99

The first paragraph begins with a descriptions what has to be done in *formatted* output functions. Probably this is a typo and the paragraph really want to describe unformatted output functions...

Proposed resolution:

In 27.6.2.6 lib.ostream.unformatted paragraph 1, the first and last sentences, change the word "formatted" to "unfomatted":

> "Each **unformatted** output function begins ..."
> "... value specified for the **unformatted** output function."

---

169. Bad efficiency of `overflow()` mandated

Section:: 27.7.1.3 lib.stringbuf.virtuals **Status:** Ready **Submitter:** Dietmar Kühl **Date:** 20 Jul 99

Paragraph 8, Notes, of this section seems to mandate an extremely inefficient way of buffer handling for `basic_stringbuf`, especially in view of the restriction that `basic_ostream` member functions are not allowed to use `xsputn()` (see 27.6.2.1 lib.ostream): For each character to be inserted, a new buffer is to be created.

Of course, the resolution below requires some handling of simultaneous input and output since it is no longer possible to update `egptr()` whenever `epptr()` is changed. A possible solution is to handle this in `underflow()`.

Proposed resolution:

In 27.7.1.3 lib.stringbuf.virtuals paragraph 8, Notes, insert the words "at least" as in the following:

> To make a write position available, the function reallocates (or initially allocates) an array object with a sufficient number of elements to hold the current array object (if any), plus **at least** one additional write position.

---

170. Inconsistent definition of `traits_type`

Section:: 27.7.4 lib.stringstream **Status:** Review **Submitter:** Dietmar Kühl **Date:** 20 Jul 99

The classes `basic_stringstream` (27.7.4, lib.stringstream), `basic_istringstream` (27.7.2, lib.istringstream), and `basic_ostringstream` (27.7.3, lib.ostringstream) are inconsistent in their definition of the type `traits_type`: For `istringstream`, this type is defined, for the other two it is not. This should be consistent.

Proposed resolution:

To the declarations of `basic_ostringstream` (27.7.3, lib.ostringstream) and `basic_stringstream` (27.7.4, lib.stringstream) add:

```
typedef traits traits_type;
```

---

171. Strange `seekpos()` semantics due to joint position

Section:: 27.8.1.4 lib.filebuf.virtuals **Status:** Open **Submitter:** Dietmar Kühl **Date:** 20 Jul 99

Overridden virtual functions, seekpos()

In 27.8.1.1 (lib.filebuf) paragraph 3, it is stated that a joint input and output position is maintained by `basic_filebuf`. Still, the description of `seekpos()` seems to talk about different file positions. In particular, it is unclear (at least to me) what is supposed to happen to the output buffer (if there is one) if only the input position is changed. The standard seems to mandate that the output buffer is kept and processed as if there was no positioning of the output position (by changing the input position). Of course, this can be exactly what you want if the flag `ios_base::ate` is set. However, I think, the standard should say something like this:

- If `(which & mode) == 0` neither read nor write position is changed and the call fails. Otherwise, the joint read and write position is altered to correspond to `sp`.
- If there is an output buffer, the output sequences is updated and any unshift sequence is written before the position is altered.
- If there is an input buffer, the input sequence is updated after the position is altered.

Plus the appropriate error handling, that is...

Proposed resolution:

---

172. Inconsistent types for `basic_istream::ignore()`

Section:: 27.6.1.3 lib.istream.unformatted **Status:** Ready **Submitter:** Greg Comeau, Dietmar Kühl **Date:** 23 Jul 99

In 27.6.1.1 (lib.istream) the function `ignore()` gets an object of type `streamsize` as first argument. However, in 27.6.1.3 (lib.istream.unformatted) paragraph 23 the first argument is of type `int`.

As far as I can see this is not really a contradiction because everything is consistent if `streamsize` is typedef to be `int`. However, this is almost certainly not what was intended. The same thing happened to `basic_filebuf::setbuf()`.

Darin Adler also submitted this issue, commenting: Either 27.6.1.1 should be modified to show a first parameter of type int, or 27.6.1.3 should be modified to show a first parameter of type streamsize and use numeric_limits<streamsize>::max.

Proposed resolution:

In 27.6.1.3 (lib.istream.unformatted) paragraph 23 and 24, change both uses of `int` in the description of `ignore()` to `streamsize`.

---

173. Inconsistent types for `basic_filebuf::setbuf()`

Section:: 27.8.1.4 lib.filebuf.virtuals **Status:** Ready **Submitter:** Greg Comeau, Dietmar Kühl **Date:** 23 Jul 99

In 27.8.1.1 (lib.istream) the function `setbuf()` gets an object of type `streamsize` as second argument. However, in 27.8.1.4 (lib.istream.unformatted) paragraph 9 the second argument is of type `int`

. As far as I can see this is not really a contradiction because everything is consistent if `streamsize` is typedef to be `int`. However, this is almost certainly not what was intended. The same thing happened to `basic_istream::ignore()`.

Proposed resolution:

In 27.8.1.4 (lib.istream.unformatted) paragraph 9, change all uses of `int` in the description of `setbuf()` to `streamsize`.

---

174. Typo: `OFF_T` vs. `POS_T`

Section:: D.6 depr.ios.members **Status:** Ready **Submitter:** Dietmar Kühl **Date:** 23 Jul 99

According to paragraph 1 of this section, `streampos` is the type `OFF_T`, the same type as `streamoff`. However, in paragraph 6 the `streampos` gets the type `POS_T`

Proposed resolution:

Change D.6 depr.ios.members paragraph 1 from "`typedef OFF_T streampos;`" to "`typedef POS_T streampos;`"

---

175. Ambiguity for `basic_streambuf::pubseekpos()` and a few other functions.

Section:: D.6 depr.ios.members **Status:** Ready **Submitter:** Dietmar Kühl **Date:** 23 Jul 99

According to paragraph 8 of this section, the methods `basic_streambuf::pubseekpos()`, `basic_ifstream::open()`, and `basic_ofstream::open` "may" be overloaded by a version of this function taking the type `ios_base::open_mode` as last argument argument instead of `ios_base::openmode` (`ios_base::open_mode` is defined in this section to be an alias for one of the integral types). The clause specifies, that the last argument has a default argument in three cases. However, this generates an ambiguity with the overloaded version because now the arguments are absolutely identical if the last argument is not specified.

Proposed resolution:

In D.6 depr.ios.members paragraph 8, remove the default arguments for `basic_streambuf::pubseekpos()`, `basic_ifstream::open()`, and `basic_ofstream::open()`.

---

176. `exceptions()` in `ios_base`...?

Section:: D.6 depr.ios.members **Status:** Ready **Submitter:** Dietmar Kühl **Date:** 23 Jul 99

The "overload" for the function `exceptions()` in paragraph 8 gives the impression that there is another function of this function defined in class `ios_base`. However, this is not the case. Thus, it is hard to tell how the semantics (paragraph 9) can be implemented: "Call the corresponding member function specified in clause lib.input.output."

Proposed resolution:

In D.6 depr.ios.members paragraph 8, move the declaration of the function `exceptions()`into class `basic_ios`.

---

177. Complex operators cannot be explicitly instantiated

Section: 26.2.6 lib.complex.ops **Status:** Open **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.

Proposed Resolution:

I'm not really sure. I think the choices are:

1. Do nothing. I think users will be surprised that there are certain functions in the standard library that cannot be explicitly instantiated.

2. Add specializations of iterator_traits for the built-in types or specialize it in general for iterator_traits<T>.

3. Put the non-member operator functions that are currently all in namespace std in different namespaces, i.e. the complex operators would have their own subnamespace, the reverse_iterator operators would have their own namespace, etc.

---

179. Comparison of const_iterators to iterators doesn't work

Section: 24.1.1 lib.iterator.requirements **Status:** Open **Submitter:** Judy Ward **Date:** 2 Jul 1998

Currently the following will not compile on two well-known standard library implementations:

```
#include <set>
using namespace std;

void f(const set<int> &s)
{
  set<int>::iterator i;
  if (i==s.end()); // s.end() returns a const_iterator
}
```

The reason this doesn't compile is because operator== was implemented as a member function of the nested classes set:iterator and set::const_iterator, and there is no conversion from const_iterator to iterator. Surprisingly, (s.end() == i) does work, though, because of the conversion from iterator to const_iterator.

I don't see a requirement anywhere in the standard that this must work. Should there be one? If so, I think the requirement would need to be added to the tables in section 24.1.1. I'm not sure about the wording. If this requirement existed in the standard, I would think that implementors would have to make the comparison operators non-member functions.

This issues was also raised on comp.std.c++ by Darin Adler. The example given was:

```
bool check_equal(std::deque<int>::iterator i,
std::deque<int>::const_iterator ci)
{
```

```
return i == ci;
}
```

Proposed Resolution:

---

180. Container member iterator arguments constness has unintended consequences

Section: 23 lib.containers **Status:** Open **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.

Proposed resolution:

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);
```

---

181. make_pair() unintended behavior

Section: 20.2.2 lib.pairs **Status:** Open **Submitter:** Andrew Koenig **Date:** 3 Aug 99

The claim has surfaced in Usenet that expressions such as

```
make_pair("abc", 3)
```

are illegal, notwithstanding their use in examples, because template instantiation tries to bind the first template parameter to `const char (&)[4]`, which type is uncopyable.

I doubt anyone intended that behavior...

Proposed resolution:

---

182. Ambiguous references to size_t

Section: 17 lib.library **Status:** Review **Submitter:** Al Stevens **Date:** 15 Aug 99

Many references to `size_t` throughout the document omit the `std::` namespace qualification.

For example, 17.4.3.4 [lib.replacement.functions] paragraph 2:

```
— operator new(size_t)
— operator new(size_t, const std::nothrow_t&)
— operator new[](size_t)
— operator new[](size_t, const std::nothrow_t&)
```

Proposed resolution:

Throughout the library clauses of the Standard, qualify with `std::` names from namespace `std`, such as `size_t` and `ptrdiff_t`, unless their use is within the scope of namespace `std`.

Rationale:

The LWG believes correcting names like `size_t` and `ptrdiff_t` to `std::size_t` and `std::ptrdiff_t` to be essentially editorial. The issue is treated as a Defect Report to make explicit the Project Editor's authority to make this change.

---

183. I/O stream manipulators don't work for wide character streams

Section: 27.6.3 lib.std.manip **Status:** Open **Submitter:** Andy Sawyer **Date:** 7 Jul 99

27.6.3 [lib.std.manip] paragraph 3 says (clause numbering added for exposition):

Returns: An object s of unspecified type such that if [1] out is an (instance of) basic_ostream then the expression out<<s behaves as if f(s) were called, and if [2] 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, ios_base::fmtflags mask) { // reset specified flags str.setf(ios_base::fmtflags(0), mask); return str; } [3] The expression out<<s has type ostream& and value out. [4] The expression in>>s has type istream& and value in.

Given the definitions [1] and [2] for out and in, surely [3] should read: "The expression out << s has type basic_ostream& ..." and [4] should read: "The expression in >> s has type basic_istream& ..."

If the wording in the standard is correct, I can see no way of implementing any of the manipulators so that they will work with wide character streams.

e.g. wcout << setbase(16);

Must have value 'wcout' (which makes sense) and type 'ostream&' (which doesn't).

The same "cut'n'paste" type also seems to occur in Paras 4,5,7 and 8. In addition, Para 6 [setfill] has a similar error, but relates only to ostreams.

I'd be happier if there was a better way of saying this, to make it clear that the value of the expression is "the same specialization of basic_ostream as out"&

Proposed resolution:

Maybe replace [1] with "out is an instance of basic_ostream<charT,traitsT> for some charT and some traitsT" ... and [3] with: "The expression out << s has type basic_ostream&<charT,traitsT>" ... and do something similar for [2]&[4]. But

this strikes me as being somewhat cumbersome.

---

184. numeric_limits<bool> wording problems

Section: 18.2.1 lib.limits **Status:** Open **Submitter:** Gabriel Dos Reis **Date:** 21 Jul 99

bools are defined by the standard to be of integer types, as per 3.9.1/7 [basic.fundamental]. However "integer types" seems to have a special meaning for the author of 18.2. The net effect is an unclear and confusing specification for numeric_limits<bool> as evidenced below.

18.2.1.2/7 says numeric_limits<>::digits is, for built-in integer types, the number of non-sign bits in the representation.

4.5/4 states that a bool promotes to int ; whereas 4.12/1 says any non zero arithmetical value converts to true.

I don't think it makes sense at all to require numeric_limits<bool>::digits and numeric_limits<bool>::digits10 to be meaningful.

The standard defines what constitutes a signed (resp. unsigned) integer types. It doesn't categorize bool as being signed or unsigned. And the set of values of bool type has only two elements.

I don't think it makes sense to require numeric_limits<bool>::is_signed to be meaningful.

18.2.1.2/18 for numeric_limits<integer_type>::radix says:

> For integer types, specifies the base of the representation.186)

This disposition is at best misleading and confusing for the standard requires a "pure binary numeration system" for integer types as per 3.9.1/7

The footnote 186) says: "Distinguishes types with base other than 2 (e.g BCD)." This also erroneous as the standard never defines any integer types with base representation other than 2.

Furthermore, numeric_limits<bool>::is_modulo and numeric_limits<bool>::is_signed have similar problems.

Proposed resolution:

Change 18.2.1 [lib.limits] paragraph 2, from:

> Specializations shall be provided for each fundamental type, both floating point and integer, including bool.

to:

> Specializations shall be provided for each fundamental type, both floating point and integer, except bool.

Remove `template<> class numeric_limits<bool>;` from the synopsis, 18.2.1 paragraph 4.

Change18.2.1.2 lib.numeric.limits.members paragraph18 from:

> For integer types, specifies the base of the representation.

to:

For all integer types other than bool, shall be 2 (3.9.1). Not meaningful for bool.

Remove footnote 186 which reads:

Distinguishes types with base other than 2 (e.g BCD).

---

185. Questionable use of term "inline"

Section: 20.3 lib.function.objects **Status:** Open **Submitter:** UK Panel **Date:** 26 Jul 99

Paragraph 4 of 20.3 [lib.function.objects] says:

> [Example: To negate every element of a: transform(a.begin(), a.end(), a.begin(), negate<double>()); The corresponding functions will inline the addition and the negation. end example]

(Note: The "addition" referred to in the above is in para 3) we can find no other wording, except this (non-normative) example which suggests that any "inlining" will take place in this case.

Indeed both:

> 17.4.4.3 Global Functions [lib.global.functions] 1 It is unspecified whether any global functions in the C++ Standard Library are defined as inline (7.1.2).

and

> 17.4.4.4 Member Functions [lib.member.functions] 1 It is unspecified whether any member functions in the C++ Standard Library are defined as inline (7.1.2).

take care to state that this may indeed NOT be the case.

Thus the example "mandates" behavior that is explicitly not required elsewhere.

Proposed resolution:

Change 20.3 [lib.function.objects] paragraph 2 from:

> Using function objects together with function templates increases the expressive power of the library as well as making the resulting code much more efficient.

to:

> Using function objects together with function templates increases the expressive power of the library.

Option 1: Remove from 20.3 [lib.function.objects] paragraph 4 the sentence:

> "The corresponding functions will inline the addition and the negation."

Option 2: Change "will" to "may" in 20.3 [lib.function.objects] paragraph 4 so the sentence becomes:

> "The corresponding functions may inline the addition and the negation."

---

186. bitset::set() second parameter should be bool

Section: 23.3.5.2 lib.bitset.members **Status:** Open **Submitter:** Darin Adler **Date:** 13 Aug 99

In section 23.3.5.2 [lib.bitset.members], paragraph 13 defines the bitset::set operation to take a second parameter of type int. The function tests whether this value is non-zero to determine whether to set the bit to true or false. The type of this second parameter should be bool. For one thing, the intent is to specify a Boolean value. For another, the result type from test() is bool. In addition, it's possible to slice an integer that's larger than an int. This can't happen with bool, since conversion to bool has the semantic of translating 0 to false and any non-zero value to true.

Proposed resolution:

In 23.3.5.2 [lib.bitset.members], paragraph 13 and in 23.3.5 [lib.template.bitset] change the type of the second parameter to bitset::set to `bool`

---

187. iter_swap underspecified

Section: 25.2.2 lib.alg.swap **Status:** Ready **Submitter:** Andrew Koenig **Date:** 14 Aug 99

The description of iter_swap in 25.2.2 paragraph 7,says that it ``exchanges the values'' of the objects to which two iterators refer.

What it doesn't say is whether it does so using swap or using the assignment operator and copy constructor.

This question is an important one to answer, because swap is specialized to work efficiently for standard containers. For example:

```
vector<int> v1, v2;
iter_swap(&v1, &v2);
```

Is this call to iter_swap equivalent to calling swap(v1, v2)? Or is it equivalent to

```
{
vector<int> temp = v1;
v1 = v2;
v2 = temp;
}
```

The first alternative is O(1); the second is O(n).

A LWG member, comments:

> Not an objection necessarily, but I want to point out the cost of that requirement:
>
> ```
> iter_swap(list<T>::iterator, list<T>::iterator)
> ```
>
> can currently be specialized to be more efficient than iter_swap(T*,T*) for many T (by using splicing). Your proposal would make that optimization illegal.

Proposed resolution:

Change the effect clause of iter_swap in 25.2.2 paragraph 7 from:

> Exchanges the values pointed to by the two iterators a and b.

to

```
swap(*a, *b).
```

---

189. setprecision() not specified correctly

Section: 27.4.2.2 lib.fmtflags.state **Status:** Ready **Submitter:** Andrew Koenig **Date:** 25 Aug 99

27.4.2.2 paragraph 9 claims that setprecision() sets the precision, and includes a parenthetical note saying that it is the number of digits after the decimal point.

This claim is not strictly correct. For example, in the default floating-point output format, setprecision sets the number of significant digits printed, not the number of digits after the decimal point.

I would like the committee to look at the definition carefully and correct the statement in 27.4.2.2

Proposed resolution:

Remove from 27.4.2.2 lib.fmtflags.state, paragraph 9, the text "(number of digits after the decimal point)".

---

193. Heap operations description incorrect

Section: 25.3.6 lib.alg.heap.operations **Status**: Ready **Submitter:** Markus Mauhart **Date:** 24 Sep 99

25.3.6 [lib.alg.heap.operations] states two key properties of a heap [a,b), the first of them is

`"(1) *a is the largest element"

I think this is incorrect and should be changed to the wording in the proposed resolution.

Actually there are two independent changes:

> A-"part of largest equivalence class" instead of "largest", cause 25.3 [lib.alg.sorting] asserts "strict weak ordering" for all its sub clauses.
>
> B-Take 'an oldest' from that equivalence class, otherwise the heap functions could not be used for a priority queue as explained in 23.2.3.2.2 [lib.priqueue.members] (where I assume that a "priority queue" respects priority AND time).

Proposed Resolution:

Change 25.3.6 [lib.alg.heap.operations] property (1) from:

> (1) *a is the largest element

to:

(1) There is no element greater than `*a`

---

195. Should `basic_istream::sentry`'s constructor ever set eofbit?

Section: 27.6.1.1.2 lib.istream::sentry **Status**: Ready **Submitter:** Matt Austern **Date:**13 Oct 99

Suppose that `is.flags() & ios_base::skipws` is nonzero. What should `basic_istream<>::sentry`'s constructor do if it reaches eof while skipping whitespace? 27.6.1.1.2/5 suggests it should set failbit. Should it set eofbit as well? The standard doesn't seem to answer that question.

On the one hand, nothing in 27.6.1.1.2 [lib.istream::sentry] says that `basic_istream<>::sentry` should ever set eofbit. On the other hand, 27.6.1.1/4 [lib.istream] says that if extraction from a `streambuf` "returns `traits::eof()`, then the input function, except as explicitly noted otherwise, completes its actions and does `setstate(eofbit)`". So the question comes down to whether `basic_istream<>::sentry`'s constructor is an input function.

Comments from Jerry Schwarz:

> It was always my intention that eofbit should be set any time that a virtual returned something to indicate eof, no matter what reason iostream code had for calling the virtual.
>
> The motivation for this is that I did not want to require streambufs to behave consistently if their virtuals are called after they have signalled eof.
>
> The classic case is a streambuf reading from a UNIX file. EOF isn't really a state for UNIX file descriptors. The convention is that a read on UNIX returns 0 bytes to indicate "EOF", but the file descriptor isn't shut down in any way and future reads do not neccessarily also return 0 bytes. In particular, you can read from tty's on UNIX even after they have signalled "EOF". (It isn't always understood that a ^D on UNIX is not an EOF indicator, but an EOL indicator. By typing a "line" consisting solely of ^D you cause a read to return 0 bytes, and by convention this is interpreted as end of file.)

Proposed Resolution:

Add a sentence to the end of 27.6.1.1.2 paragraph 2:

> If `is.rdbuf()->sbumpc()` or `is.rdbuf()->sgetc()` returns `traits::eof()`, the function calls `setstate(failbit | eofbit)` (which may throw `ios_base::failure`).

---

196. Placement new example has alignment problems

Section: 18.4.1.3 lib.new.delete.placement **Status**: New **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.

Proposed Resolution:

Change the code in the example in 18.4.1.3 [lib.new.delete.placement] paragraph 4 to:

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

---

197. max_size() underspecified

Section: 20.1.5 lib.allocator.requirements, 21.3.3 lib.string.capacity, 23.1 lib.container.requirements **Status**: New
Submitter: Andy Sawyer **Date:** 21 Oct 99

Must the value returned by max_size() be unchanged from call to call?

Must the value returned from max_size() be meaningful?

Possible meanings identified in lib-6827:

1) The largest container the implementation can support given "best case" conditions - i.e. assume the run-time platform is "configured to the max", and no overhead from the program itself. This may possibly be determined at the point the library is written, but certainly no later than compile time.

2) The largest container the program could create, given "best case" conditions - i.e. same platform assumptions as (1), but take into account any overhead for executing the program itself. (or, roughly "storage=storage-sizeof(program)"). This does NOT include any resource allocated by the program. This may (or may not) be determinable at compile time.

3) The largest container the current execution of the program could create, given knowledge of the actual run-time platform, but again, not taking into account any currently allocated resource. This is probably best determined at program start-up.

4) The largest container the current execution program could create at the point max_size() is called (or more correctly at the point max_size() returns :-), given it's current environment (i.e. taking into account the actual currently available resources). This, obviously, has to be determined dynamically each time max_size() is called.

Proposed Resolution:

---

198. Validity of references is unspecified after iterator destruction

Section: 23.1 lib.container.requirements **Status**: New **Submitter:** Beman Dawes **Date:** 3 Nov 99

Is a reference or pointer to a container element still valid after destruction of the iterator that the reference was obtained from?

```
// assume c is some non-empty standard library container
T* p = &*c.begin();

... // is p still valid at this point?

c.clear();  // clearly invalidates p
```

If references must remain valid after iterator destruction, it is not possible to implement standard conforming containers which return iterators to cached elements. This is a particular problem for large disk-based containers like B-trees as they cannot be portably implemented without caching elements.

Three well-known implementations of <algorithm> seem to be written as if references do not remain valid after iterator destruction. Thus these implementations appear to already conform to the proposed resolution. Whether this is by design or happenstance isn't known.

The standard doesn't appear to address this question. It needs to be made clear to both users and implementors.

Proposed Resolution:

Add a new paragraph at the end of 23.1 lib.container.requirements:

> Destruction of an iterator invalidates container element references and pointers previously obtained from that iterator.

---

199. What does `allocate(0)` return?

Section: 20.1.5 lib.allocator.requirements **Status**: New **Submitter:** Matt Austern **Date:** 19 Nov 99

Suppose that `A` is a class that conforms to the Allocator requirements of Table 32, and `a` is an object of class `A` What should be the return value of `a.allocate(0)`? Three reasonable possibilities: forbid the argument `0`, return a null pointer, or require that the return value be a unique non-null pointer.

Proposed Resolution:

Alternative A: Add a note to the `allocate` row of Table 32: "[*Note:* If `n == 0`, the return value is a null pointer. --*end note*]"

Alternative B: Add a note to the `allocate` row of Table 32: "[*Note:* The return value is not a null pointer even when `n == 0`. --*end note*]"

---

200. Forward iterator requirements don't allow constant iterators

Section: 24.1.3 lib.forward.iterators **Status**: New **Submitter:** Matt Austern **Date:** 19 Nov 99

In table 74, the return type of the expression `*a` is given as `T&`, where `T` is the iterator's value type. For constant iterators, however, this is wrong. ("Value type" is never defined very precisely, but it is clear that the value type of, say, `std::list<int>::const_iterator` is supposed to be `int`, not `const int`.)

Proposed Resolution:

In table 74, change the **return type** column for `*a` from "`T&`" to "`T&` if `X` is mutable, otherwise `const T&`".

---

201. Numeric limits terminology unclear

Section: 18.2.1 lib.limits **Status**: New **Submitter:** Stephen Cleary **Date:** 21 Dec 1999

In some places in this section, the terms "fundamental types" and "scalar types" are used when the term "arithmetic types" is intended. The current usage is incorrect because void is a fundamental type and pointers are scalar types, neither of which should have specializations of numeric_limits.

Proposed Resolution:

Change 18.2 [lib.support.limits] para 1 from:

> The headers <limits>, <climits>, and <cfloat> supply characteristics of implementation-dependent fundamental types (3.9.1).

to:

> The headers <limits>, <climits>, and <cfloat> supply characteristics of implementation-dependent arithmetic types (3.9.1).

Change 18.2.1 [lib.limits] para 1 from:

> The numeric_limits component provides a C++ program with information about various properties of the implementation's representation of the fundamental types.

to:

> The numeric_limits component provides a C++ program with information about various properties of the implementation's representation of the arithmetic types.

Change 18.2.1 [lib.limits] para 2 from:

> Specializations shall be provided for each fundamental type. . .

to:

> Specializations shall be provided for each arithmetic type. . .

Change 18.2.1 [lib.limits] para 4 from:

> Non-fundamental standard types. . .

to:

> Non-arithmetic standard types. . .

Change 18.2.1.1 [lib.numeric.limits] para 1 from:

> The member is_specialized makes it possible to distinguish between fundamental types, which have specializations, and non-scalar types, which do not.

to:

> The member is_specialized makes it possible to distinguish between arithmetic types, which have specializations, and non-arithmetic types, which do not.

---

202. unique() effects unclear when predicate not an equivalence relation

Section: 25.2.8 lib.alg.unique **Status**: New **Submitter:** Andrew Koenig **Date:** 13 Jan 00

What should unique() do if you give it a predicate that is not an equivalence relation? There are at least two plausible answers:

> 1. You can't, because 25.2.8 says that it it "eliminates all but the first element from every consecutive group of equal elements..." and it wouldn't make sense to interpret "equal" as meaning anything but an equivalence relation. [It also doesn't make sense to interpret "equal" as meaning ==, because then there would never be any sense in giving a predicate as an argument at all.]
>
> 2. The word "equal" should be interpreted to mean whatever the predicate says, even if it is not an equivalence relation (and in particular, even if it is not transitive).

The example that raised this question is from Usenet:

```
int f[] = { 1, 3, 7, 1, 2 };
int* z = unique(f, f+5, greater<int>());
```

If one blindly applies the definition using the predicate greater<int>, and ignore the word "equal", you get:

> Eliminates all but the first element from every consecutive group of elements referred to by the iterator i in the range [first, last) for which *i > *(i - 1).

The first surprise is the order of the comparison. If we wanted to allow for the predicate not being an equivalence relation, then we should surely compare elements the other way: pred(*(i - 1), *i). If we do that, then the description would seem to say: "Break the sequence into subsequences whose elements are in strictly increasing order, and keep only the first element of each subsequence". So the result would be 1, 1, 2. If we take the description at its word, it would seem to call for strictly DEcreasing order, in which case the result should be 1, 3, 7, 2.

In fact, the SGI implementation of unique() does neither: It yields 1, 3, 7.

Proposed Resolution:

What should we do? I think there are two alternatives:

> 1. Impose an additional requirement that the predicate be an equivalence relation.
>
> 2. Drop the word "equal" from the description to make it clear that the intent is to compare pairs of adjacent elements.

If we adopt (2), we also need to decide whether pred(*i, *(i - 1)) is really what we meant, or whether pred(*(i - 1), i) is more appropriate.

---

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

Section: 27.6.1.1.2 lib.istream::sentry **Status**: New **Submitter:** Matt McClure and Dietmar Kuehl **Date:** 1 Jan 00

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.

Proposed Resolution:

Change the first sentence of 27.6.1.1.2 paragraph 4:

> To decide if the character c is a whitespace character, the constructor behaves, during translation and execution "as if" it were implemented with the following code fragment: ...

---

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

Section: 24.3.4 lib.iterator.operations **Status**: New **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?

Proposed Resolution:

---

205. numeric_limits unclear on how to determine floating point types

Section: 18.2.1.2 lib.numeric.limits.members **Status**: New **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.

Proposed Resolution:

---

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:** New **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.

Proposed resolution:

Change the default behavior of operator new (size_t, nothrow_t) in 18.4.1.1 lib.new.delete.single (paragraph 8) from:

-8- Default behavior:

- Executes a loop: Within the loop, the function first attempts to allocate the requested storage. Whether the attempt involves a call to the Standard C library function `malloc` is unspecified.
- Returns a pointer to the allocated storage if the attempt is successful. Otherwise, if the last argument to `set_new_handler()` was a null pointer, return a null pointer.
- Otherwise, the function calls the current *new_handler* (lib.new.handler). If the called function returns, the loop repeats.
- The loop terminates when an attempt to allocate the requested storage is successful or when a called *new_handler* function does not return. If the called *new_handler* function terminates by throwing a `bad_alloc` exception, the function returns a null pointer.

to

-8- Default behavior:

- Calls the ordinary operator new(size_t) returning the result, but catches any exceptions thrown.
- If an exception is caught, returns a null pointer.

---

207. ctype<char> members return clause incomplete

Section: 22.2.1.3.2 lib.facet.ctype.char.members **Status**: New **Submitter:** Robert Klarer **Date:** 2 Nov 99

Proposed Resolution:

Change the returns clause in 22.2.1.3.2 lib.facet.ctype.char.members paragraph 10 from:

Returns: do_widen(low, high, to).

to:

Returns: do_widen(c) or do_widen(low, high, to), respectively.

Change the returns clause in 22.2.1.3.2 lib.facet.ctype.char.members paragraph 11 from:

Returns: do_narrow(low, high, to).

to:

Returns: do_narrow(c) or do_narrow(low, high, to), respectively.

---

208. Unnecessary restriction on past-the-end iterators

Section: 24.1 lib.iterators **Status**: New **Submitter:** Stephen Cleary **Date:** 02 Feb 00

In 24.1 paragraph 5, it is stated ". . . Dereferenceable and past-the-end values are always non-singular."

This places an unnecessary restriction on past-the-end iterators for containers with forward iterators (for example, a singly-linked list). If the past-the-end value on such a container was a well-known singular value, it would still satisfy all forward iterator requirements.

Removing this restriction would allow, for example, a singly-linked list without a "footer" node.

This would have an impact on existing code that expects past-the-end iterators obtained from different (generic) containers being not equal.

Proposed Resolution:

Change the wording of 24.1 [Iterator requirements] paragraph 5 to ". . . Dereferenceable values are always non-singular. If the iterator category is bidirectional or random access, a past-the-end value of that iterator is always non-singular."

---

209. basic_string declarations inconsistent

Section: 21.3 lib.basic.string **Status**: New **Submitter:** Igor Stauder **Date:** 11 Feb 00

In Section 21.3 [lib.basic.string] the basic_string member function declarations use a consistent style except for the following functions:

```
void push_back(const charT);
basic_string& assign(const basic_string&);
void swap(basic_string<charT,traits,Allocator>&);
```

- push_back, assign, swap: missing argument name
- push_back: use of const with charT (i.e. POD type passed by value not by reference - should be charT or const charT&)
- swap: redundant use of template parameters in argument basic_string<charT,traits,Allocator>&

Proposed Resolution:

In Section 21.3 [lib.basic.string] change the basic_string member function declarations push_back, assign, and swap to:

```
void push_back(charT c);
        or
void push_back(const charT& c);


basic_string& assign(const basic_string& str);
void swap(basic_string& str);
```

---

210. distance first and last confused

Section: 25 lib.algorithms **Status**: New **Submitter:** Lisa Lippincott **Date:** 15 Feb 00

In paragraph 9 of section 25 [lib.algorithms], it is written:

> In the description of the algorithms operators + and - are used for some of the iterator categories for which they do not have to be defined. In these cases the semantics of [...] a-b is the same as of

```
return distance(a, b);
```

Proposed Resolution:

On the last line of paragraph 9 of section 25 [lib.algorithms] change `return distance(a, b);` to `return distance(b, a);`

---

211. operator>>(istream&, string&) doesn't set failbit

Section: 21.3.7.9 lib.string.io **Status**: New **Submitter:** Scott Snyder **Date:** 4 Feb 00

The description of the stream extraction operator for std::string (section 21.3.7.9 [lib.string.io]) does not contain a requirement that failbit be set in the case that the operator fails to extract any characters from the input stream.

This implies that the typical construction

```
std::istream is;
std::string str;
...
while (is >> str) ... ;
```

(which tests failbit) is not required to terminate at EOF.

Furthermore, this is inconsistent with other extraction operators, which do include this requirement. (See sections 27.6.1.2 [lib.istream.formatted] and 27.6.1.3 [lib.istream.unformatted], where this requirement is present, either explicitly or implicitly, for the extraction operators. It is also present explicitly in the description of getline (istream&, string&, charT) in section 21.3.7.9 [lib.string.io] paragraph 8.)

Proposed Resolution:

Insert new paragraph after paragraph 2 in section 21.3.7.9 [lib.string.io]:

> If the function extracts no characters, it calls is.setstate(ios::failbit) which may throw ios_base::failure (27.4.4.3).

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