Doc. no. J16/99-0030
WG21 N1206
Date: 25 August 1999
Project: Programming Language C++

C++ Standard Library Active Issues List (Revision 9)

Reference ISO/IEC IS 14882:1998(E)

Also see:

- Table of Contents including both active and closed issues.
- Index by Section including both active and closed issues.
- Index by Status including both active and closed issues.
- 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 the "C++ Standard Library Closed Issues List" for closed issues.

The issues in these issues lists are not necessarily formal ISO Defect Reports (DR's). While some issues will eventually be elevated to 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

- R9: pre-Kona II mailing. Added issues 140 to 189. List document split into separate "active" and "closed" documents. (25 Aug 99)
- R8: post-Dublin mailing. Updated to reflect LWG and full committee actions in Dublin. (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. (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:

---

5. String::compare specification questionable

Section: 21.3.6.8 lib.string::compare **Status**: Ready **Submitter:** Jack Reeves **Date:** 11 Dec 97

At the very end of the basic_string class definition is the signature: int compare(size_type pos1, size_type n1, const charT* s, size_type n2 = npos) const; In the following text this is defined as: returns basic_string<charT,traits,Allocator>(*this,pos1,n1).compare(basic_string<charT,traits,Allocator>(s,n2);

Since the constructor basic_string(const charT* s, size_type n, const Allocator& a = Allocator()) clearly requires that s != NULL and n < npos and further states that it throws length_error if n == npos, it appears the compare() signature above should always throw length error if invoked like so: str.compare(1, str.size()-1, s); where 's' is some null terminated character array.

This appears to be a typo since the obvious intent is to allow either the call above or something like: str.compare(1, str.size()-1, s, strlen(s)-1);

This would imply that what was really intended was two signatures int compare(size_type pos1, size_type n1, const charT* s) const int compare(size_type pos1, size_type n1, const charT* s, size_type n2) const; each defined in terms of the corresponding constructor.

Proposed Resolution:

Replace the compare signature in 21.3 lib.basic.string (at the very end of the basic_string synopsis) which reads:

```
int compare(size_type pos1, size_type n1,
            const charT* s, size_type n2 = npos) const;
```

with:

```
int compare(size_type pos1, size_type n1,
            const charT* s) const;
int compare(size_type pos1, size_type n1,
            const charT* s, size_type n2) const;
```

Replace the portion of 21.3.6.8 lib.string::compare paragraphs 5 and 6 which read:

```
int compare(size_type pos, size_type n1,
            charT * s, size_type n2 = npos) const;
Returns:
basic_string<charT,traits,Allocator>(*this, pos, n1).compare(
             basic_string<charT,traits,Allocator>( s, n2))
```

with:

```
int compare(size_type pos, size_type n1,
```

```
            const charT * s) const;
Returns:
basic_string<charT,traits,Allocator>(*this, pos, n1).compare(
             basic_string<charT,traits,Allocator>( s ))

int compare(size_type pos, size_type n1,
            const charT * s, size_type n2) const;
Returns:
basic_string<charT,traits,Allocator>(*this, pos, n1).compare(
             basic_string<charT,traits,Allocator>( s, n2))
```

Editors please note that in addition to splitting the signature, the third argument becomes const, matching the existing synopsis.

Rationale:

While the LWG dislikes adding signatures, this is a clear defect in the Standard which must be fixed. The same problem was also identified in issues 7.5 and 87.

---

7. String clause minor problems

Section: 21 lib.strings **Status**: Ready **Submitter:** Matt Austern **Date:** 15 Dec 97

(1) In 21.3.5.4 lib.string::insert, the description of template <class InputIterator> insert(iterator, InputIterator, InputIterator) makes no sense. It refers to a member function that doesn't exist. It also talks about the return value of a void function.

(2) Several versions of basic_string::replace don't appear in the class synopsis.

(3) basic_string::push_back appears in the synopsis, but is never described elsewhere. In the synopsis its agument is const charT, which doesn't makes much sense; it should probably be charT, or possible const charT&.

(4) basic_string::pop_back is missing.

(5) int compare(size_type pos, size_type n1, charT* s, size_type n2 = npos) make no sense. First, it's const charT* in the synopsis and charT* in the description. Second, given what it says in RETURNS, leaving out the final argument will always result in an exception getting thrown. This is paragraphs 5 and 6 of 21.3.6.8 lib.string::compare.

(6) In table 37, in section 21.1.1 lib.char.traits.require, there's a note for X::move(s, p, n). It says "Copies correctly even where p is in [s, s+n)". This is correct as far as it goes, but it doesn't go far enough; it should also guarantee that the copy is correct even where s in in [p, p+n). These are two orthogonal guarantees, and neither one follows from the other. Both guarantees are necessary if X::move is supposed to have the same sort of semantics as memmove (which was clearly the intent), and both guarantees are necessary if X::move is actually supposed to be useful.

Proposed Resolution:

ITEM 1: In 21.3.5.4 [lib.string::insert], change paragraph 16 to

EFFECTS: Equivalent to insert(p - begin(), basic_string(first, last)).

ITEM 2: Not a defect; the Standard is clear.. There are ten versions of replace() in the synopsis, and ten versions in 21.3.5.6 [lib.string::replace].

ITEM 3: Change the declaration of push_back in the string synopsis (21.3, [lib.basic.string]) from:

void push_back(const charT)

to

void push_back(charT)

Add the following text immediately after 21.3.5.2 [lib.string::append], paragraph 10.

void basic_string::push_back(charT c);
EFFECTS: Equivalent to append(static_cast<size_type>(1), c);

ITEM 4: Not a defect. The omission appears to have been deliberate.

ITEM 5: Duplicate; see issue 5 (and 87).

ITEM 6: In table 37, Replace:

"Copies correctly even where p is in [s, s+n)."

with:

"Copies correctly even where the ranges [p, p+n) and [s, s+n) overlap."

---

8. Locale::global lacks guarantee

Section: 22.1.1.5 lib.locale.statics **Status**: Open **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 note in 22.1.1.5 lib.locale.statics: "the library shall behave as if no other library function calls locale::global()."

---

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:

---

11. Bitset minor problems

Section: 23.3.5 lib.template.bitset **Status**: Ready **Submitter:** Matt Austern **Date:** 22 Jan 98

(1) bitset<>::operator[] is mentioned in the class synopsis (23.3.5), but it is not documented in 23.3.5.2.

(2) The class synopsis only gives a single signature for bitset<>::operator[]m reference operator[](size_t pos). This doesn't make much sense. It ought to be overloaded on const. reference operator[](size_t pos) bool operator[](size_t pos) const.

(3) Bitset's stream input function (23.3.5.3) ought to skip all whitespace before trying to extract 0s and 1s. The standard doesn't explicitly say that, though. This should go in the Effects clause.

Rationale:

The LWG believes Item 3 is not a defect. "Formatted input" implies the desired semantics. See 27.6.1.2 lib.istream.formatted.

Proposed Resolution:

ITEMS 1 AND 2:

In the bitset synopsis (23.3.5, [lib.template.bitset]), replace the member function

```
reference operator[](size_t pos);
```

with the two member functions

```
bool operator[](size_t pos) const;
reference operator[](size_t pos);
```

Add the following text at the end of 23.3.5.2 [lib.bitset.members], immediately after paragraph 45:

```
bool operator[](size_t pos) const;
```
Requires: pos is valid
Throws: nothing
Returns: `test(pos)`

```
bitset<N>::reference operator[](size_t pos);
```
Requires: pos is valid
Throws: nothing
Returns: An object of type `bitset<N>::reference` such that `(*this)[pos] == this->test(pos)`, and such that `(*this)[pos] = val` is equivalent to `this->set(pos, val);`

---

17. Bad bool parsing

Section: 22.2.2.1.2 lib.facet.num.get.virtuals **Status**: Review **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**: Open **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: input sequence is identical to converted sequence.

and change the Note in paragraph 2 to normative text as follows:

> If returns _noconv_, the converted sequence is identical to the input sequence _[from,from_next) _._to_next_ is set equal to _to_, and the value of _state_ is unchanged.

---

21. Codecvt_byname<> instantiations

Section: 22.1.1.1.1 lib.locale.category **Status**: Review **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**: Ready **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**: Review **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**: Review **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 subobject of a `basic_ios<>` object or subobject, the effect is equivalent to calling `basic_ios<>::setstate(failbit)` 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, 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()`.

Rationale:

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

The LWG considered two other possible resolutions:

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);
```

C. 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()`.

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:

---

48. Use of non-existent exception constructor

Section: 27.4.2.1.1 lib.ios::failure **Status**: Ready **Submitter:** Matt Austern **Date:** 21 Jun 98

27.4.2.1.1, paragraph 2, says that class failure initializes the base class, exception, with exception(msg). Class exception (see 18.6.1) has no such constructor.

Proposed Resolution:

Replace 27.4.2.1.1 [lib.ios::failure], paragraph 2, with

> EFFECTS: Constructs an object of class `failure`.

---

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**: Open **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:

---

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()`.

Add a footnote to 27.4.4.2 lib.basic.ios.members paragraph 6, rdbuf effects, which says:

`rdbuf(0)` does not set `badbit`.

---

54. Basic_streambuf's destructor

Section: 27.5.2.1 lib.streambuf.cons **Status**: Ready **Submitter:** Matt Austern **Date:**25 Jun 98

The class synopsis for basic_streambuf shows a (virtual) destructor, but the standard doesn't say what that destructor does. My assumption is that it does nothing, but the standard should say so explicitly.

Proposed Resolution:

Add after 27.5.2.1 lib.streambuf.cons paragraph 2:

```
virtual  ~basic_streambuf();
```

Effects: None.

---

55. Invalid stream position is undefined

Section: 27 lib.input.output **Status**: Ready **Submitter:** Matt Austern **Date:**26 Jun 98

Several member functions in clause 27 are defined in certain circumstances to return an "invalid stream position", a term that is defined nowhere in the standard. Two places (27.5.2.4.2, paragraph 4, and 27.8.1.4, paragraph 15) contain a cross-reference to a definition in _lib.iostreams.definitions_, a nonexistent section.

I suspect that the invalid stream position is just supposed to be pos_type(-1). Probably best to say explicitly in (for example) 27.5.2.4.2 that the return value is pos_type(-1), rather than to use the term "invalid stream position", define that term somewhere, and then put in a cross-reference.

The phrase "invalid stream position" appears ten times in the C++ Standard. In seven places it refers to a return value, and it should be changed. In three places it refers to an argument, and it should not be changed. Here are the three places where "invalid stream position" should not be changed:

27.7.1.3 [lib.stringbuf.virtuals], paragraph 14
27.8.1.4 [lib.filebuf.virtuals], paragraph 14

D.7.1.3 [depr.strstreambuf.virtuals], paragraph 17

Proposed Resolution:

In 27.5.2.4.2 [lib.streambuf.virt.buffer], paragraph 4, change "Returns an object of class pos_type that stores an invalid stream position (_lib.iostreams.definitions_)" to "Returns `pos_type(off_type(-1))`".

In 27.5.2.4.2 [lib.streambuf.virt.buffer], paragraph 6, change "Returns an object of class pos_type that stores an invalid stream position" to "Returns `pos_type(off_type(-1))`".

In 27.7.1.3 [lib.stringbuf.virtuals], paragraph 13, change "the object stores an invalid stream position" to "the return value is `pos_type(off_type(-1))`".

In 27.8.1.4 [lib.filebuf.virtuals], paragraph 13, change "returns an invalid stream position (27.4.3)" to "returns `pos_type(off_type(-1))`"

In 27.8.1.4 [lib.filebuf.virtuals], paragraph 15, change "Otherwise returns an invalid stream position (_lib.iostreams.definitions_)" to "Otherwise returns `pos_type(off_type(-1))`"

In D.7.1.3 [depr.strstreambuf.virtuals], paragraph 15, change "the object stores an invalid stream position" to "the return value is `pos_type(off_type(-1))`"

In D.7.1.3 [depr.strstreambuf.virtuals], paragraph 18, change "the object stores an invalid stream position" to "the return value is `pos_type(off_type(-1))`"

---

58. Extracting a char from a wide-oriented stream

Section: 27.6.1.2.3 lib.istream::extractors **Status**: Open **Submitter:** Matt Austern **Date:**1 Jul 98

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

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

Proposed Resolution:

`operator>>` should use `narrow` to convert from `charT` to `char`.

---

60. What is a formatted input function?

Section: 27.6.1.2.1 lib.istream.formatted.reqmts **Status**: Open **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.

Proposed Resolution:

The three member functions described in paragraphs 1-5 and the one described in paragraph 12-14 of section 27.6.1.2.3 should each have something added (perhaps a Notes clause?) that says: "The common requirements listed in section 27.6.1.2.1 do not apply to this function."

The four member functions described in paragraphs 1-9 of section 27.6.2.5.3 should each have something added (perhaps a Notes clause?) and the one described in section that says: "The common requirements listed in section 27.6.2.5.1 do not apply to this function."

---

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:

Clarify that the phrase "thrown during input" refers only to exceptions thrown by streambuf's overridden virtuals, not exceptions thrown as part of istream's error-reporting mechanism.

---

63. Exception-handling policy for unformatted output

Section: 27.6.2.6 lib.ostream.unformatted **Status**: Open **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:

Add an exception-handling policy similar to the one in 27.6.2.5.1 lib.ostream.formatted.reqmts, paragraph 1. The omission seems to have been unintentional.

---

65. Underspecification of strstreambuf::seekoff

Section: D.7.1.3 depr.strstreambuf.virtuals **Status**: Open **Submitter:** Matt Austern **Date:**18 Aug 98

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

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

Proposed Resolution:

---

69. Must elements of a vector be contiguous?

Section: 23.2.4 lib.vector **Status**: Ready **Submitter:** Andrew Koenig **Date:** 29 Jul 1998

The issue is this:

> Must the elements of a vector be in contiguous memory?

(Please note that this is entirely separate from the question of whether a vector iterator is required to be a pointer; the answer to that question is clearly "no," as it would rule out debugging implementations)

Proposed Resolution:

Add the following text to the end of 23.2.4 [lib.vector], paragraph 1.

> The elements of a vector are stored contiguously, meaning that if V is a `vector<T, Allocator>` where T is some type other than `bool`, then it obeys the identity `&V[n] == &V[0] + n` for all `0 <= n < V.size()`.

Rationale:

The LWG feels that as a practical matter the answer is clearly "yes". There was considerable discussion as to the best way to express the concept of "contiguous", which is not directly defined in the standard. Discussion included:

- An operational definition similar to the above proposed resolution is already used for valarray (26.3.2.3).
- There is no need to explicitly consider a user-defined operator& because elements must be copyconstructible (23.1 para 3) and copyconstructible (20.1.3) specifies requirements for operator&.
- There is no issue of one-past-the-end because of language rules.

---

74. Garbled text for `codecvt::do_max_length`

Section: 22.2.1.5.2 lib.locale.codecvt.virtuals **Status**: Ready **Submitter:** Matt Austern **Date:** 18 Sep 98

The text of `codecvt::do_max_length`'s "Returns" clause (22.2.1.5.2, paragraph 11) is garbled. It has unbalanced parentheses and a spurious **n**.

Proposed Resolution:

Replace 22.2.1.5.2 lib.locale.codecvt.virtuals paragraph 11 with the following:

> Returns: The maximum value that `do_length(state, from, from_end, 1)` can return for any valid range `[from, from_end)` and `stateT` value state. The specialization `codecvt<char, char, mbstate_t>::do_max_length()` returns 1.

---

75. Contradiction in `codecvt::length`'s argument types

Section: 22.2.1.5 lib.locale.codecvt **Status**: Ready **Submitter:** Matt Austern **Date:** 18 Sep 98

The class synopses for classes `codecvt<>` (22.2.1.5) and `codecvt_byname<>` (22.2.1.6) say that the first parameter of the member functions `length` and `do_length` is of type `const stateT&`. The member function descriptions, however (22.2.1.5.1, paragraph 6; 22.2.1.5.2, paragraph 9) say that the type is `stateT&`. Either the synopsis or the summary must be changed.

If (as I believe) the member function descriptions are correct, then we must also add text saying how `do_length` changes its `stateT` argument.

Proposed Resolution:

In 22.2.1.5 [lib.locale.codecvt], and also in 22.2.1.6 [lib.locale.codecvt_byname], change the `stateT` argument type on both member `length()` and member `do_length()` from

```
const stateT&
```

to

```
stateT&
```

In 22.2.1.5.2 [lib.locale.codecvt.virtuals], add to the definition for member `do_length` a paragraph:

> Effects: The effect on the `state` argument is ``as if'' it called `do_in(state, from, from_end, from, to, to+max, to)` for `to` pointing to a buffer of at least `max` elements.

---

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 lib.strings **Status**: Open **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:

---

86. String constructors don't describe exceptions

Section: 21.3.1 lib.string.cons **Status**: Open **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:

---

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

Section: 21.3.7.9 lib.string.io **Status**: Open **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 reachin EOF. So shouldn't it be changed into that it reads until !good() ?

Proposed resolution:

---

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 impolement 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)".

---

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].

Rationale:

The LWG believes the answer should be "yes, adding template parameters with default values should be permitted." A careful reading of 17.4.4 and its subclauses found no mention of additional template parameters.

---

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.

Proposed Resolution:

---

110. istreambuf_iterator::equal not const

Section: 24.5.3 [lib.istreambuf.iterator], 24.5.3.5 [lib.istreambuf.iterator::equal] **Status**: Ready **Submitter:** Nathan Myers **Date:** 15 Oct 98

Member istreambuf_iterator<>::equal is not declared "const", yet 24.5.3.6 [lib.istreambuf.iterator::op==] says that operator==, which is const, calls it. This is contradictory.

Proposed Resolution:

In 24.5.3 [lib.istreambuf.iterator] and also in 24.5.3.5 [lib.istreambuf.iterator::equal], replace:

```
bool equal(istreambuf_iterator& b);
```

with:

```
bool equal(const istreambuf_iterator& b) 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**: Review **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
```

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 the two sentences above, since 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.

---

124. ctype_byname<charT>::do_scan_is & do_scan_not return type should be const charT*

Section: 22.2.1.2 lib.locale.ctype.byame **Status:** Ready **Submitter:** Judy Ward **Date:** 15 Dec 1998

In Section 22.2.1.2 lib.locale.ctype.byame ctype_byname<charT>::do_scan_is() and do_scan_not() are declared to return a const char* not a const charT*.

Proposed Resolution:

Change Section 22.2.1.2 lib.locale.ctype.byame do_scan_is() and do_scan_not() to return a const charT*.

---

125. valarray<T>::operator!() return type is inconsistent

Section: 26.3.2 lib.template.valarray **Status:** Ready **Submitter:** Judy Ward **Date:** 15 Dec 1998

In Section 26.3.2 lib.template.valarray valarray<T>::operator!() is declared to return a valarray<T>, but in Section 26.3.2.5 lib.valarray.unary it is declared to return a valarray<bool>. The latter appears to be correct.

Proposed Resolution:

Change in Section 26.3.2 lib.template.valarray the declaration of operator!() so that the return type is valarray<bool>.

---

126. typos in Effects clause of ctype::do_narrow()

Section: 22.2.1.1.2 lib.locale.ctype.virtuals **Status:** Ready **Submitter:** Judy Ward **Date:** 15 Dec 1998

In Section 22.2.1.1.2 lib.locale.ctype.virtuals the following typos need to be fixed:

```
    do_widen(do_narrow(c),0) == c
```

should be:

```
    do_widen(do_narrow(c,0)) == c

    (is(M,c) || !ctc.is(M, do_narrow(c),dfault) )
```

should be:

```
    (is(M,c) || !ctc.is(M, do_narrow(c,dfault)) )
```

Proposed Resolution:

Fix as suggested above

---

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.

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`).

---

132. list::resize description uses random access iterators

Section: 23.2.2.2 lib.list.capacity **Status:** Ready **Submitter:** Howard Hinnant **Date:** 6 Mar 99

The description reads:

-1- Effects:

```
if (sz > size())
  insert(end(), sz-size(), c);
else if (sz < size())
  erase(begin()+sz, end());
else
  ;                           //  do nothing
```

Obviously list::resize should not be specified in terms of random access iterators.

Proposed Resolution:

Change 23.2.2.2 paragraph 1 to:

Effects:

```
if (sz > size())
  insert(end(), sz-size(), c);
else if (sz < size())
{
  iterator i = begin();
  advance(i, sz);
  erase(i, end());
}
```

---

133. map missing get_allocator()

Section: 23.3.1 lib.map **Status:** Ready **Submitter:** Howard Hinnant **Date:** 6 Mar 99

The title says it all.

Proposed Resolution:

Insert:

```
allocator_type get_allocator() const;
```

after operator= in 23.3.1, paragraph 2, in the map declaration.

---

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

---

139. Optional sequence operation table description unclear

Section: 23.1.1 lib.sequence.reqmts **Status**: Ready **Submitter:** Andrew Koenig **Date:** 30 Mar 99

The sentence introducting the Optional sequence operation table (23.1.1 paragraph 12) has two problems:

A. It says ``The operations in table 68 are provided only for the containers for which they take constant time."

That could be interpreted in two ways, one of them being ``Even though table 68 shows particular operations as being provided, implementations are free to omit them if they cannot implement them in constant time."

B. That paragraph says nothing about amortized constant time, and it should.

Proposed Resolution:

Replace the wording in 23.1.1 paragraph 12 with:

> Table 68 lists sequence operations that are provided for some types of sequential containers but not others. An implementation shall provide these operations for all container types shown in the ``container" column, and shall implement them so as to take amortized constant time.

---

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

Section: 23.3.1 lib.map **Status**: New **Submitter:** Mark Mitchell **Date:** 14 Apr 99

> [lib.container.requirements]
>
> | expression | return type | pre/post-condition |
> |---|---|---|
> | X::value_type | T | T is assignable |
>
> [lib.map]
>
> A map satisfies all the requirements of a container.
>
> For a map<Key, T> ... the value_type is pair<const Key, T>.

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

[See 103 for the slightly related issue of modifcation of set keys]

Proposed Resolution:

---

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**: New **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**: New **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."

---

143. C header wording unclear

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

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**: New **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.

Alternatively, if you want to talk about allocations too (but note that the other deque complexity requirements do not talk about allocations):

> Complexity: Makes N calls to the copy constructor of T, and performs a number of allocations that is linear in N, where N is last - first.

---

145. adjustfield lacks default value

Section: 27.4.4.1 lib.basic.ios.cons **Status**: New **Submitter:** Angelika Langer **Date:** 12 May 99

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

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

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

Proposed Resolution:

Change Table 89, "ios_base() effects" in section 27.4.4.1, flags() entry from:

```
skipws | dec
```

to:

```
skipws | dec | right
```

---

146. complex<T> Inserter and Extractor need sentries

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

---

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

Section: 17.4.4.3 lib.global.functions **Status**: New **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 "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**: New **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!"; }
```

---

149. Insert should return iterator to first element inserted

Section: 23.1.1 lib.sequence.reqmts **Status**: New **Submitter:** Andrew Koenig **Date:** 28 Jun 99

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

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

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

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

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

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

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

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

Alternatively, I might write something like

```
bool first = i == c.begin();
list<T>::iterator j = i;
if (!first) --j;
c.insert(i, c1.begin(), c1.end());
if (first)
   j = c.begin();
else
   ++j;
```

which, although wretched, requires less overhead.

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

Proposed Resolution:

---

150. Find_first_of says integer instead of iterator

Section: 25.1.4 lib.alg.find.first.of **Status**: New **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**: New **Submitter:** Ed Bray **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:** New **Submitter:** Dietmar Kühl **Date:** 20 Jul 99

The semantics of `scan_is()` (paragraph 4) 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:

Change the return clause to say "... such that `is(m, *p)` would..", that is, fix the typo.

---

153. Typo in `narrow()` semantics

Section:: 22.2.1.3.2 lib.facet.ctype.char.members **Status:** New **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:** New **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 `double`s is actually a problem I found quite often in production code, too).

Proposed resolution:

Add a row in the table to say that for `double` a length modifier `l` is to be used.

---

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

Section:: 27.3 lib.iostream.objects **Status:** New **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`".

---

156. Typo in `imbue()` description

Section:: 27.4.2.3 lib.ios.base.locales **Status:** New **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:

27.4.2.3 (lib.ios.base.locales) change "`locale const" to "locale const&`".

---

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

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

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

Proposed resolution:

Move the state handling and exception handling functionality from `basic_ios` into `ios_base`: It is character type independent anyway. Although this might be too big a fix for a technical corrigendum it is the only reasonable fix.

---

158. Underspecified semantics for `setbuf()`

Section:: 27.5.2.4.2 lib.streambuf.virt.buffer **Status:** New **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:

The text of 27.5.2.4.2 lib.streambuf.virt.buffer should be modified to become: "Default behavior: Does nothing. Returns this."

---

159. Strange use of `underflow()`

Section:: 27.5.2.4.3 lib.streambuf.virt.get **Status:** New **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 the description of `showmanyc()` return value to use a sentence like this, replacing the corresponding current sentence:

> If it returns a positive value, then successive calls to `sbumpc()` will not return `traits::eof()` until at least that number of characters have been supplied.

Beman Dawes comments: The full "from" and "to" wording needs to be supplied.

---

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

Section:: 27.6.1.1 lib.istream **Status:** New **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:** New **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...

---

162. Really "formatted input functions"?

Section:: 27.6.1.2.3 lib.istream::extractors **Status:** New **Submitter:** Dietmar Kühl **Date:** 20 Jul 99

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

See also below for the same problem is formatted output

Proposed resolution:

Clarify that these operators are not to be considered formatted input functions, eg. by explicitly stating that they don't construct a sentry object. ... or, clarify the other way around if there is a consensus that this is what is intended (as far as I can tell, no implementations consider them to be formatted input functions).

---

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

Section:: 27.6.1.3 lib.istream.unformatted **Status:** New **Submitter:** Dietmar Kühl **Date:** 20 Jul 99

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

Proposed resolution:

Clear things up, adding text which says for the functions in question whether they have to create a sentry object and what happens to the `gcount()`.

---

164. do_put() has apparently unused fill argument

Section:: 22.2.5.3.2 lib.locale.time.put.virtuals **Status:** New **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:

---

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

Section:: 27.6.2.1 lib.ostream **Status:** New **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.

---

166. Really "formatted output functions"?

Section:: 27.6.2.5.3 lib.ostream.inserters **Status:** New **Submitter:** Dietmar Kühl **Date:** 20 Jul 99

From 27.6.2.5.1 (lib.ostream.formatted.reqmts) it appears that all the functions defined in 27.6.2.5.3

(lib.ostream.inserters) have to construct a `sentry` object. Is this really intended?

This is basically the same problem as the corresponding defect report for formatted input but for output instead of input.

Proposed resolution:

Do one of

1. clarify, that these functions are indeed formatted output function.
2. clarify, that these functions are not formatted output functions.

(as far as I can tell, the latter would match existing implementations, the former the current wording; well, no, otherwise correcting it would be a change... :-)

---

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

Section:: 27.6.2.5.4 lib.ostream.inserters.character **Status:** New **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:** New **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:

Change the first sentence of paragraph 1 to read:

> Each **unformatted** output function begins execution by constructing an object of class sentry.

---

169. Bad efficiency of `overflow()` mandated

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

Paragraph 9 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.

Proposed resolution:

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.

Of course, this resolution 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()`.

---

170. Inconsistent definition of `traits_type`

Section:: 27.7.4 lib.stringstream **Status:** New **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:

Define `traits_type` for all types.

---

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

Section:: 27.8.1.4 lib.filebuf.virtuals **Status:** New **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...

---

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

Section:: 27.6.1.3 lib.istream.unformatted **Status:** New **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 19 (actually, the numbering of the paragraphs seems to be messed up there, too) 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:

Change all 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:** New **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 10 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:

Change all uses of `int` in the description of `setbuf()` to `streamsize`.

---

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

Section:: D.4.6 depr.ios.members **Status:** New **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:

Fix paragraph 1 to use `POS_T` for type `streampos`

---

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

Section:: D.4.6 depr.ios.members **Status:** New **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:

Remove the default arguments for these three functions. The only problem is for `basic_ofstream::open()` because the default arguments specified in 27.8.1.8 (lib.ofstream and Appendix 4.6 (depr.ios.members is inconsistent: `ios_base::out` vs. `ios_base::out | ios_base::trunc`.

---

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

Section:: D.4.6 depr.ios.members **Status:** New **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:

Move this function into the proper class (`basic_ios`).

---

177. Complex operators cannot be explicitly instantiated

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

---

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

Section: 27.3.1 lib.narrow.stream.objects **Status:** New **Submitter:** Judy Ward **Date:** 2 Jul 99

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

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

Proposed resolution:

Add requirements to section 27.3.1 that cerr and clog be initially tied to cout, i.e. cerr.tie() == &cout and clog.tie() == &cout.

---

179. Comparison of const_iterators to iterators doesn't work

Section: 24.1.1 lib.iterator.requirements **Status:** New **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::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:** New **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:** New **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 (&)[3]`, which type is uncopyable.

I doubt anyone intended that behavior...

Proposed resolution:

---

182. Ambiguous references to size_t

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

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

Proposed resolution:

---

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

Section: 27.6.3 lib.std.manip **Status:** New **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:** New **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:** New **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.

Furthermore: 20.3/p2: "Using function objects together with function templates increases the expressive power of the library as well as making the resulting code much more efficient."

Whilst this is probably generally true, should it be included as normative text? Perhaps turn this into a note. (Especially since it says nothing about "more efficient than what".)

Proposed resolution:

Remove from Paragraph 4 of 20.3 [lib.function.objects] the sentence

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

---

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

Section: 23.3.5.2 lib.bitset.members **Status:** New **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:** New **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).

---

188. valarray helpers missing augmented assignment operators

Section: 26.3.2.6 lib.valarray.cassign **Status:** New **Submitter:** Gabriel Dos Reis **Date:** 15 Aug 99

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

```
#include <valarray>
```

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

v[99] *= 2.0; // Ok

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

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

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

Proposed resolution:

I suggest we add those missing operators.

---

189. setprecision() not specified correctly

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

27.4.2.2 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:

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