

Doc. no. J16 00-0019R1
WG21 N1242
Date: 26 April 2000
Project: Programming Language C++
Reply to: Matt Austern <austern@isolde.engr.sgi.com>

C++ Standard Library Active Issues List (Revision 14)

Reference ISO/IEC IS 14882:1998(E)

Also see:

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

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

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

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

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

Prior to Revision 14, library issues lists existed in two slightly different versions; a Committee Version and a Public Version. Beginning with Revision 14 the two versions were combined into a single version.

This document includes *[bracketed italicized notes]* as a reminder to the LWG of current progress on issues. Such notes are strictly unofficial and should be read with caution as they may be incomplete or incorrect. Be aware that LWG support for a particular resolution can quickly change if new viewpoints or killer examples are presented in subsequent discussions.

For the most current 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++.com](http://news.comp.std.c++.com).

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

- R14: post-Tokyo II mailing; reflects committee actions taken in Tokyo. Added issues [228](#) to [232](#). (00-0019R1/N1242)
- R13: pre-Tokyo II updated: Added issues [212](#) to [227](#).
- R12: pre-Tokyo II mailing: Added issues [199](#) to [211](#). (00-0003/N1226)
- R11: post-Kona mailing: Updated to reflect LWG and full committee actions in Kona (99-0048/N1224). Note changed resolution of issues [4](#) and [38](#). Added issues [196](#) to [198](#). Closed issues list split into "defects" and "closed" documents.
- R10: pre-Kona updated. Added proposed resolutions [83](#), [86](#), [91](#), [92](#), [109](#). Added issues [190](#) to [195](#). (99-0033/D1209, 14 Oct 99)
- R9: pre-Kona mailing. Added issues [140](#) to [189](#). Issues list split into separate "active" and "closed" documents. (99-0030/N1206, 25 Aug 99)
- R8: post-Dublin mailing. Updated to reflect LWG and full committee actions in Dublin. (99-0016/N1193, 21 Apr 99)
- R7: pre-Dublin updated: Added issues [130](#), [131](#), [132](#), [133](#), [134](#), [135](#), [136](#), [137](#), [138](#), [139](#) (31 Mar 99)
- R6: pre-Dublin mailing. Added issues [127](#), [128](#), and [129](#). (99-0007/N1194, 22 Feb 99)
- R5: update issues [103](#), [112](#); added issues [114](#) to [126](#). Format revisions to prepare for making list public. (30 Dec 98)
- R4: post-Santa Cruz II updated: Issues [110](#), [111](#), [112](#), [113](#) added, several issues corrected. (22 Oct 98)
- R3: post-Santa Cruz II: Issues [94](#) to [109](#) added, many issues updated to reflect LWG consensus (12 Oct 98)
- R2: pre-Santa Cruz II: Issues [73](#) to [93](#) added, issue [17](#) updated. (29 Sep 98)
- R1: Correction to issue [55](#) resolution, [60](#) code format, [64](#) title. (17 Sep 98)

Issue Status

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

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

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

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

Dup - The LWG has reached consensus that the issue is a duplicate of another issue, and will not be further dealt with. A **Rationale** identifies 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:** [Ready](#) **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:

Change section 18.3/8 from:

First, objects with static storage duration are destroyed and functions registered by calling *atexit* are called. Objects with static storage duration are destroyed in the reverse order of the completion of their constructor. (Automatic objects are not destroyed as a result of calling *exit*(.) Functions registered with *atexit* are called in the reverse order of their registration. A function registered with *atexit* before an object *obj1* of static storage duration is initialized will not be called until *obj1*'s destruction has completed. A function registered with *atexit* after an object *obj2* of static storage duration is initialized will be called before *obj2*'s destruction starts.

to:

First, objects with static storage duration are destroyed and functions registered by calling *atexit* are called. Non-local objects with static storage duration are destroyed in the reverse order of the completion of their constructor. (Automatic objects are not destroyed as a result of calling *exit*(.) Functions registered with *atexit* are called in the reverse order of their registration, except that a function is called after any previously registered functions that had already been called at the time it was registered. A function registered with *atexit* before a non-local object *obj1* of static storage duration is initialized will not be called until *obj1*'s destruction has completed. A function registered with *atexit* after a non-local object *obj2* of static storage duration is initialized will be called before *obj2*'s destruction starts. A local static object *obj3* is destroyed at the same time it would be if a function calling the *obj3* destructor were registered with *atexit* at the completion of the *obj3* constructor.

Paper:

See 99-0039/N1215, October 22, 1999, by Stephen D. Clamage for the analysis supporting to the proposed resolution.

[Tokyo: Reviewed by the LWG.]

8. `locale::global` lacks guarantee

Section: 22.1.1.5 [lib.locale.statics](#) **Status:** [Ready](#) **Submitter:** Matt Austern **Date:** 24 Dec 97

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

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

Proposed Resolution:

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

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

[Tokyo: Reviewed by the LWG.]

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

Section: 18.4.1 [lib.new.delete](#) **Status:** [Ready](#) **Submitter:** Steve Clamage **Date:** 4 Jan 98

Scott Meyers, in a 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 in the regard? [Issues list maintainer's note: the IS is the same.]

Proposed Resolution:

Change the last paragraph of 3.7.3 from:

Any allocation and/or deallocation functions defined in a C++ program shall conform to the semantics specified in 3.7.3.1 and 3.7.3.2.

to:

Any allocation and/or deallocation functions defined in a C++ program, including the default versions in the library, shall conform to the semantics specified in 3.7.3.1 and 3.7.3.2.

Change 3.7.3.1/2, next-to-last sentence, from :

If the size of the space requested is zero, the value returned shall not be a null pointer value (4.10).

to:

Even if the size of the space requested is zero, the request can fail. If the request succeeds, the value returned shall be a non-null pointer value (4.10) `p0` different from any previously returned value `p1`, unless that value `p1` was since passed to an operator `delete`.

5.3.4/7 currently reads:

When the value of the expression in a direct-new-declarator is zero, the allocation function is called to

allocate an array with no elements. The pointer returned by the new-expression is non-null. [Note: If the library allocation function is called, the pointer returned is distinct from the pointer to any other object.]

Retain the first sentence, and delete the remainder.

18.4.1 currently has no text. Add the following:

Except where otherwise specified, the provisions of 3.7.3 apply to the library versions of operator new and operator delete.

To 18.4.1.3, add the following text:

The provisions of 3.7.3 do not apply to these reserved placement forms of operator new and operator delete.

Paper:

See 99-0040/N1216, October 22, 1999, by Stephen D. Clamage for the analysis supporting to the proposed resolution.

[Tokyo: Reviewed by the LWG.]

19. "Noconv" definition too vague

Section: 22.2.1.5.2 [lib.locale.codecvt.virtuals](#) **Status:** [Ready](#) **Submitter:** Nathan Myers **Date:** 6 Aug 98

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

Proposed Resolution:

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

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

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

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

[Tokyo: Reviewed by the LWG.]

26. Bad sentry example

Section: 27.6.1.1.2 [lib.istream::sentry](#) **Status:** [Ready](#) **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:

Remove the example above from 27.6.1.1.2 [lib.istream::sentry](#) paragraph 6.

Rationale:

The originally proposed replacement code for the example was not correct. The LWG tried in Kona and again in Tokyo to correct it without success. In Tokyo, an implementor reported that actual working code ran over one page in length and was quite complicated. The LWG decided that it would be counter-productive to include such a lengthy example, which might well still contain errors.

[Tokyo: Reviewed by the LWG.]

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

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, as long as some locale object refers to that facet.

[Tokyo: Reviewed by the LWG.]

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:

[Kona: Nathan to supply proposed wording.]

[Tokyo: the LWG reaffirmed that this is a defect, and requires careful review of clause 27 as the changes are context sensitive.]

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 sentence 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`.

[The LWG agrees (2) that a definition of synchronized is required. Jerry Schwarz will work by email with Matt Austern to provide such a definition.]

[Tokyo: PJP knows approximate wording, and will help Matt formulate final wording.]

61. Ambiguity in iostreams exception policy

Section: 27.6.1.3 [lib.istream.unformatted](#) **Status:** [Ready](#) **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:

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

[Tokyo: The LWG looked to two alternative wordings submitted by Matt, and choose the proposed resolution as better standardese.]

63. Exception-handling policy for unformatted output

Section: 27.6.2.6 [lib.ostream.unformatted](#) **Status:** [Ready](#) **Submitter:** Matt Austern **Date:** 11 Aug 98

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

Proposed Resolution:

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

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

[Kona: Matt Austern provided the proposed resolution wording.]

[Tokyo: Reviewed by the LWG.]

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 $N > I$. 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:

[Kona: Matt Austern will attempt wording; it is very complex.]

86. String constructors don't describe exceptions

Section: 21.3.1 [lib.string.cons](#) **Status:** [Ready](#) **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 throws clause. 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:

In 21.3.1 [lib.string.cons](#), Strike throws paragraphs for constructors which say "Throws: `length_error` if `n == npos`."

Rationale:

Throws clauses for `length_error` if `n == npos` are no longer needed because they are subsumed by the general wording added by the resolution for issue [83](#).

[Tokyo: Reviewed by the LWG.]

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 reaching EOF. So shouldn't it be changed into that it reads until !good() ?

Proposed resolution:

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

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

with:

- an attempt to extract a character fails;

In 21.3.7.9 [\[lib.string.io\]](#), paragraph 5, last sentence, replace :

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

with:

- an attempt to extract a character fails

In 23.3.5.3 [\[lib.bitset.operators\]](#), paragraph 5, last sentence, replace:

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

with:

- an attempt to extract a character fails;

[Tokyo: Sentiment was expressed for a single blanket statement which applies to all extractors (string, formatted, or unformatted). The problem with the proposed resolution is that there is no concept of read failure in iostreams. If not exception is thrown, sooner or later eof will be reached.]

92. Incomplete Algorithm Requirements

Section: 25 [lib.algorithms](#) **Status:** [Open](#) **Submitter:** Nico Josuttis **Date:** 29 Sep 98

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

```
class Nth {      // function object that returns true for the nth element
private:
    int nth;      // element to return true for
    int count;    // element counter
public:
    Nth (int n) : nth(n), count(0) {
    }
    bool operator() (int) {
        return ++count == nth;
    }
};

....
// remove third element
list<int>::iterator pos;
pos = remove_if(coll.begin(), coll.end(), // range
                Nth(3)),                 // remove criterion
```

```
coll.erase(pos, coll.end());
```

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

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

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

Proposed resolution:

In [lib.function.objects] 20.3 Function objects add as new paragraph 6 (or insert after paragraph 1):

Option 1:

Predicates are functions or function objects that fulfill the following requirements:

- They return a Boolean value (bool or a value convertible to bool)
- It doesn't matter for the behavior of a predicate how often it is copied or assigned and how often it is called.

Option 2:

- if it's a function:
 - All calls with the same argument values yield the same result.
- if it's a function object:
 - In any sequence of calls to operator () without calling any non-constant member function, all calls with the same argument values yield the same result.
 - After an assignment or copy both objects return the same result for the same values.

[Santa Cruz: The LWG believes that there may be more to this than meets the eye. It applies to all function objects, particularly predicates. Two questions: (1) must a function object be copyable? (2) how many times is a function object called? These are in effect questions about state. Function objects appear to require special copy semantics to make state work, and may fail if calling alters state and calling occurs an unexpected number of times.

Dublin: Pete Becker felt that this may not be a defect, but rather something that programmers need to be educated about. There was discussion of adding wording to the effect that the number and order of calls to function objects, including predicates, not affect the behavior of the function object.

Pre-Kona: Nico comments: It seems the problem is that we don't have a clear statement of "predicate" in the standard. People including me seemed to think "a function returning a Boolean value and being able to be called by an STL algorithm or be used as sorting criterion or ... is a predicate". But a predicate has more requirements: It should never change its behavior due to a call or being copied. IMHO we have to state this in the standard. If you like, see section 8.1.4 of my library book for a detailed discussion.

Kona: Nico will provide wording to the effect that "unless otherwise specified, the number of copies of and calls to function objects by algorithms is unspecified". Consider placing in 25 [lib.algorithms](#) after paragraph 9

Pre-Tokyo: Angelika Langer comments: if the resolution is that algorithms are free to copy and pass around any function objects, then it is a valid question whether they are also allowed to change the type information from reference type to value type.

Tokyo: Nico will discuss this further with Matt as there are multiple problems beyond the underlying problem of no definition of "Predicate".

Post-Tokyo: Nico provided the above proposed resolutions.]

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

[Kona: The LWG agreed the standard needs clarification. After discussion with John Spicer, it seems added template parameters can be detected by a program using template-template parameters. A straw vote - "should implementors be allowed to add template parameters?" found no consensus ; 5 - yes, 7 - no.]

[Post-Kona comment from Steve Cleary via comp.std.c++:

I disagree [with the proposed resolution] for the following reason: consider user library code with template template parameters. For example, a user library object may be templated on the type of underlying sequence storage to use (deque/list/vector), since these classes all take the same number and type of template parameters; this would allow the user to determine the performance tradeoffs of the user library object. A similar example is a user library object templated on the type of underlying set storage (set/multiset) or map storage (map/multimap), which would allow users to change (within reason) the semantic meanings of operations on that object.

I think that additional template parameters should be forbidden in the Standard classes. Library writers don't lose any

expressive power, and can still offer extensions because additional template parameters may be provided by a non-Standard implementation class:

```
template <class T, class Allocator = allocator<T>, int N = 1>
class __vector
{ ... };
template <class T, class Allocator = allocator<T> >
class vector: public __vector<T, Allocator>
{ ... };
```

]

[Tokyo: Discussed but no action taken. Still no consensus as to which answer serves the greatest need.]

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:

[In Santa Cruz the LWG felt that this was Not A Defect.]

[In Dublin many present felt that failure to meet Container requirements was a defect. There was disagreement as to whether or not the optimization requirements constituted a defect.

The LWG looked at the following resolutions in some detail:

- * Not A Defect.*
- * Add a note explaining that vector<bool> does not meet Container requirements.*
- * Remove vector<bool>.*
- * Add a new category of container requirements which vector<bool> would meet.*
- * Rename vector<bool>.*

No alternative had strong, wide-spread, support and every alternative had at least one "over my dead body" response.

There was also mention of a transition scheme something like (1) add vector_bool and deprecate vector<bool> in the next standard. (2) Remove vector<bool> in the following standard.

Modifying container requirements to permit returning proxies (thus allowing container requirements conforming vector<bool>) was also discussed.

It was also noted that there is a partial but ugly workaround in that vector<bool> maybe further specialized with a customer allocator.

Kona: Herb Sutter presented his paper J16/99-0035==WG21/N1211, vector<bool>: More Problems, Better Solutions. Much discussion of a two step approach: a) deprecate, b) provide replacement under a new name. LWG straw vote on that: 1-favor, 11-could live with, 2-over my dead body. This resolution was mentioned in the LWG report to the full committee, where several additional committee members indicated over-my-dead-body positions.

Tokyo: Not discussed by the full LWG; no one claimed new insights and so time was more productively spent on other issues. In private discussions it was asserted that requirements for any solution include 1) Increasing the full committee's understanding of the problem, and 2) providing compiler vendors, authors, teachers, and of course users with specific suggestions as to how to apply the eventual solution.]

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 over constrains implementors.

Proposed Resolution:

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

[Dublin: Pete Becker will attempt improved wording.]

[Tokyo: The essence of the issue seems to have escaped. Pete will email Valentin to try to recapture it.]

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:

$N + \log(\text{size}())$ if `[i,j)` is sorted according to `value_comp()`

[This may need better specification. Matt Austern will ask Dave Musser.]

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

Section: 23.1.2 [lib.associative.reqmts](#) **Status:** [Review](#) **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.

Other Options Evaluated:

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.]

Proposed Resolution:

Add the following to 23.1.2 [lib.associative.reqmts](#) at the indicated location:

At the end of paragraph 3: "For any two keys `k1` and `k2` in the same container, calling `comp(k1, k2)` shall always return the same value."

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."

Rationale:

Several arguments were advanced for and against allowing set elements to be mutable as long as the ordering was not effected. The argument which swayed the LWG was one of safety; if elements were mutable, there would be no compile-

time way to detect of a simple user oversight which caused ordering to be modified. There was a report that this had actually happened in practice, and had been painful to diagnose.

Simply requiring that keys be immutable is not sufficient, because the comparison object may indirectly (via pointers) operate on values outside of the keys.

[Tokyo: The LWG crafted the proposed resolution and rationale.]

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

Section: 18.6.1 [lib.exception](#) para 8, 9 **Status:** [Ready](#) **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.

[Tokyo: Reviewed by the LWG.]

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.

[Dublin: Nico volunteered to organize a discussion of this and related issues. Here it is:]

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

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

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

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

```
template <class Operation>
```

```

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

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

```

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

```

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

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

```

Proposed Resolution:

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

```

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

```

insert:

```

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

```

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

```

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

```

insert:

```

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

```

[Kona: The LWG discussed this at some length. It was agreed that this is a mistake in the design, but there was no consensus on whether it was a defect in the Standard. Straw vote:

5 NAD
3 As Proposed
6 Leave open

Tokyo: The issue was not discussed.]

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.

[Dublin: People present saw no compelling reason to make change. There is also concern over not-equal. The issue is being held open for input from Nathan.

Kona: Did not discuss due to lack of time.

Tokyo: Still no discussion.]

112. Minor typo in `ostreambuf_iterator` constructor

Section: 24.5.4.1 [[lib.ostreambuf.iter.cons](#)] **Status:** [Ready](#) **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.

[Tokyo: Reviewed by the LWG.]

114. Placement forms example in error twice

Section: 18.4.1.3 [[lib.new.delete.placement](#)] **Status:** [Ready](#) **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 likely 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));
```

[Kona: See issue [196](#) (forwarded from Core), which is the same issue but with a different resolution. Need to resolve the difference.]

Tokyo: Reviewed by the LWG, which resolved the difference.]

115. Typo in stringstream constructors

Section: D.7.4.1 [[depr stringstream.cons](#)] **Status:** [Ready](#) **Submitter:** Steve Clamage **Date:** 2 Nov 1998

D.7.4.1 stringstream constructors paragraph 2 says:

Effects: Constructs an object of class stringstream, initializing the base class with `iostream(& sb)` and initializing `sb` with 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.ostream.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`.

[*Project Editor in lib-6682 indicated that these changes have already been made as editorial.*]

[*Tokyo: Reviewed by the LWG.*]

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

Section: 27.6.2.5.2 [lib.ostream.inserters.arithmetic](#) **Status:** [Open](#) **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();
```

[Dublin: The LWG feels this is probably correct, but would like to review it one more time with additional technical experts. Issue 118 is related.]

Tokyo: Matt should speak to PJP; the first example is too simplistic for signed int and signed short.]

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

Section: 27.6.1.2.2 [lib.istream.formatted.arithmetic](#) **Status:** [Review](#) **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`, `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:

In 27.6.1.2.2 Arithmetic Extractors [\[lib.istream.formatted.arithmetic\]](#), remove the two lines (1st and 3rd) which read:

```
operator>>(short& val);
...
operator>>(int& val);
```

And add the following at the end of that section (27.6.1.2.2) :

```
operator>>(short& val);
```

The conversion occurs as if performed by the following code fragment (using the same notation as for the preceding code fragment):

```
typedef num_get< charT, istreambuf_iterator<charT, traits> > numget;
iostate err = 0;
```

```

long lval;
use_facet< numget >(loc).get(*this, 0, *this, err, lval);
    if (err == 0
        && (lval < SHRT_MIN || SHRT_MAX < lval))
        err = ios_base::failbit;
setstate(err);

operator>>(int& val);

```

The conversion occurs as if performed by the following code fragment (using the same notation as for the preceding code fragment):

```

typedef num_get< charT,istreambuf_iterator<charT,traits> > numget;
iostate err = 0;
long lval;
use_facet< numget >(loc).get(*this, 0, *this, err, lval);
    if (err == 0
        && (lval < INT_MIN || INT_MAX < lval))
        err = ios_base::failbit;
setstate(err);

```

[Dublin: What about do_get? Aren't two functions need there too? Also, the LWG would like to see full wording for the Proposed Resolution.

Post-Tokyo: PJP provided the above wording.]

120. Can an implementor add specializations?

Section: 17.4.3.1 [lib.reserved.names](#) **Status:** [Review](#) **Submitter:** Judy Ward **Date:** 15 Dec 1998

Section 17.4.3.1 says:

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

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

```

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

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

```

Lib-7047 *[Matt Austern]* comments:

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

The specialization issue is actually more serious than the instantiation one. One could argue that there is a consistent status quo as far as instantiations go, but one can't argue that in the case of specializations. The standard must either (1) give library implementors license to provide explicit specializations of any library template; or (2) give a complete list of exactly which specializations must be provided, and forbid library implementors from providing any specializations not

on that list. At present the standard does neither.

Proposed Resolution:

Append to 17.4.3.1 [lib.reserved.names](#) paragraph 1:

A program may manually instantiate any templates in the standard library only if the declaration depends on a user-defined name of external linkage and the instantiation meets the standard library requirements for the original template.

[Kona: Wording should be added to the effect that users will not be allowed to manual instantiate any templates in the standard library. Judy will work on the proposed wording. Also see [issue 177](#).

Post-Tokyo: Judy Ward provided the above wording.]

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

Section: 27.5.2 [lib.streambuf](#) **Status:** [Ready](#) **Submitter:** Judy Ward **Date:** 15 Dec 1998

Section 27.5.2 describes the streambuf classes this way:

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

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

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

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

Proposed Resolution:

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

Rationale:

The streambuf synopsis already has a declaration for the typedefs and that is sufficient.

[Tokyo: Reviewed by the LWG.]

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.

[Dublin: Pete Becker spoke to Daveed Vandevoorde about this and will work on a proposed resolution.

Tokyo: Discussed together with the AFNOR paper 00-0023/N1246. The current helper slices now violate language rules due to a core language change (but most compilers don't check, so the violation has previously gone undetected). Major surgery is being asked for in this and other valarray proposals (see issue [77](#) Rationale), and a complete design review is needed before making piecemeal changes. Robert Klarer will work on formulating the issues.]

127. `auto_ptr<>` conversion issues

Section: 20.4.5 [lib.auto_ptr](#) **Status:** [Ready](#) **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`. *Also submitted by Nathan Myers, with the same proposed resolution.*

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 *suggested by Dave Abrahams, with Greg Colvin's concurrence*; 1) changed "assignment operator" to "public assignment operator", 2) changed effects to specify use of `release()`, 3) made the conversion to `auto_ptr_ref` `const`.

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

Tokyo: The LWG removed the following from the proposed resolution:

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();
```

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`.

[Tokyo: Reviewed by the LWG.]

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

Section: 27.6.1.3 [lib.istream.unformatted](#) and 27.6.2.4 [lib.istream.seek](#) **Status:** [Ready](#) **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 character 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.seek](#):

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

[Tokyo: Reviewed by the LWG.]

134. vector constructors over specified

Section: 23.2.4.1 [lib.vector.cons](#) **Status:** [Ready](#) **Submitter:** Howard Hinnant **Date:** 6 Mar 99

The complexity description says: "It does at most $2N$ calls to the copy constructor of `T` and $\log N$ 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 23.2.4.1 [lib.vector.cons](#), 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.

[Dublin: The issues hinges on whether at "most 2N calls" is correct or not. There was a feeling that 2N is correct, but the issue will be left open to allow Howard to further analyze the complexity.]

Tokyo: Needs to be integrated with issue [144](#). The LWG now agrees Howard is correct for vector, but is concerned about deque.

Post-Tokyo: Howard Hinnant analyzed proposed resolutions for deque constructors in both [134](#) and [144](#), and says that the wording in [144](#) is better for deque. Thus [134](#) has been modified to deal only with vector, and [144](#) will resolve the issue for deque.]

136. seekp, seekg setting wrong streams?

Section: 27.6.1.3 [lib.istream.unformatted](#) **Status:** [Review](#) **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)`.

[Dublin: Dietmar Kühl thinks this is probably correct, but would like the opinion of more iostream experts before taking action.]

Tokyo: Reviewed by the LWG. PJP noted that although his docs are incorrect, his implementation already implements the Proposed Resolution.

Post-Tokyo: Matt Austern comments:

Is it a problem with `basic_istream` and `basic_ostream`, or is it a problem with `basic_stringbuf`?

We could resolve my issue either by changing `basic_istream` and `basic_ostream`, or by changing `basic_stringbuf`. I actually prefer the latter change (or maybe both changes): I don't see any reason for the standard to require that `std::stringbuf s(std::string("foo"), std::ios_base::in); s.pubseekoff(0, std::ios_base::beg);` must fail.

This requirement is actually a bit weird. There's no similar requirement for `basic_streambuf<>::seekpos`, or for `basic_filebuf<>::seekoff` or `basic_filebuf<>::seekpos`.]

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

Section: 22.1.1 [lib.locale](#) **Status:** [Ready](#) **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:

Remove the phrase:

(or, failing that, in the global locale)

from section 22.1.1.

[Dublin: The opinion of other iostream experts is required.]

Tokyo: Reviewed by the LWG.]

142. lexicographical_compare complexity wrong

Section: 25.3.8 [lib.alg.lex.comparison](#) **Status:** [Ready](#) **Submitter:** Howard Hinnant **Date:** 20 Jun 99

The lexicographical_compare complexity is specified as:

"At most $\min((\text{last1} - \text{first1}), (\text{last2} - \text{first2}))$ applications of the corresponding comparison."

The best I can do is twice that expensive.

Nicolai Josuttis comments in lib-6862: You mean, to check for equality you have to check both < and > ? Yes, IMO you are right! (and Matt states this complexity in his book)

Proposed Resolution:

Change 25.3.8 [\[lib.alg.lex.comparison\]](#) complexity to:

At most $2 * \min((\text{last1} - \text{first1}), (\text{last2} - \text{first2}))$ applications of the corresponding comparison.

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

[Example:

```
for ( ; first1 != last1 && first2 != last2 ; ++first1, ++first2) {
    if (*first1 < *first2) return true;
    if (*first2 < *first1) return false;
}
return first1 == last1 && first2 != last2;
```

--end example]

[Kona: Matt Austern provided the proposed resolution wording at the request of the LWG.]

Tokyo: Reviewed by the LWG.]

144. Deque constructor complexity wrong

Section: 23.2.1.1 [lib.deque.cons](#) **Status:** [Ready](#) **Submitter:** Herb Sutter **Date:** 9 May 99

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

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

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

It makes at most $2N$ calls to the copy constructor of T and $\log N$ reallocations if they are input iterators.¹⁾

1) The complexity is greater in the case of input iterators because each element must be added individually: it is impossible to determine the distance between first and 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 .

[Kona: Reviewed by the LWG.]

Tokyo: Needs to be integrated with issue [134](#).

Post-Tokyo: Howard Hinnant analyzed proposed resolutions for deque constructors in both [134](#) and [144](#), and says that the wording in [144](#) is better for deque. Thus [134](#) has been modified to deal only with vector, and [144](#) will resolve the issue for deque.]

146. `complex<T>` Inserter and Extractor need sentries

Section: 26.2.6 [lib.complex.ops](#) **Status:** [Ready](#) **Submitter:** Angelika Langer **Date:** 12 May 99

The [extractor](#) for complex numbers is specified as:

```
template<class T, class charT, class traits>
basic_istream<charT, traits>&
operator>>(basic_istream<charT, traits>& is, complex<T>& x);
```

Effects: Extracts a complex number x of the form: u , (u) , or (u,v) , where u is the real part and v is the imaginary part (`lib.istream.formatted`).

Requires: The input values be convertible to T . If bad input is encountered, calls `is.setstate(ios::failbit)` (which may throw `ios::failure` (`lib.iostate.flags`)).

Returns: `is`.

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

The [inserter](#) for complex numbers is specified as:

```
template<class T, class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& o, const complex<T>& x);
```

Effects: inserts the complex number x onto the stream o as if it were implemented as follows:

```
template<class T, class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& o, const complex<T>& x)
```

```

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

```

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

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

Proposed Resolution:

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

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

Rationale:

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

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

[Tokyo: Reviewed by the LWG.]

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

Section: 17.4.4.3 [lib.global.functions](#) **Status:** [Ready](#) **Submitter:** Lois Goldthwaite **Date:** 4 Jun 99

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

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

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

17.4.4.3 - Global functions [lib.global.functions]

-1- It is unspecified whether any global functions in the C++ Standard Library are defined as inline (dcl.fct.spec).

-2- A call to a global function signature described in Clauses lib.language.support through lib.input.output behaves the same as if the implementation declares no additional global function signatures.*

[Footnote: A valid C++ program always calls the expected library global function. An implementation may also define additional global functions that would otherwise not be called by a valid C++ program. --
- end footnote]

-3- A global function cannot be declared by the implementation as taking additional default arguments.

17.4.4.4 - Member functions [lib.member.functions]

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

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

Proposed Resolution:

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

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

[Kona: Because operator new and delete are global, the proposed resolution was changed from "non-member" to "global or non-member.]

[Tokyo: Reviewed by the LWG.]

153. Typo in `narrow()` semantics

Section:: 22.2.1.3.2 [lib.facet.ctype.char.members](#) **Status:** [Review](#) **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 three 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](#) paragraph 10 and 11 from:

```
char          narrow(char c, char /*default*/) const;
const char* narrow(const char* low, const char* high,
                  char /*default*/, char* to) const;
```

Returns: `do_narrow(low, high, to)`.

to:

```
char          narrow(char c, char default) const;
const char* narrow(const char* low, const char* high,
                  char default, char* to) const;
```

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

[Kona: 1) the problem occurs in additional places, 2) a user defined version could be different.

Post-Tokyo: Dietmar provided the above wording at the request of the LWG. He could find no other places the problem

occurred. He asks for clarification of the Kona "a user defined version..." comment above. Perhaps it was a circuitous way of saying "dfault" needed to be uncommented?]

159. Strange use of `underflow()`

Section:: 27.5.2.4.3 [lib.streambuf.virt.get](#) **Status:** [Ready](#) **Submitter:** Dietmar Kühl **Date:** 20 Jul 99

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

Proposed resolution:

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

[Tokyo: Reviewed by the LWG.]

164. `do_put()` has apparently unused fill argument

Section:: 22.2.5.3.2 [lib.locale.time.put.virtuals](#) **Status:** [Ready](#) **Submitter:** Angelika Langer **Date:** 23 Jul 99

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

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

Proposed resolution:

Add the following note after 22.2.5.3.2 [lib.locale.time.put.virtuals](#) paragraph 2:

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

Rationale:

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

[Tokyo: Reviewed by the LWG.]

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

Section:: 27.6.2.1 [lib.ostream](#) **Status:** [Review](#) **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_streambuf...`" although the problem with the virtual members exists in both cases.

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.

Proposed resolution:

Change the last sentence of 27.6.2.1 (`lib.ostream`) paragraph 2 from:

They may use other public members of `basic_ostream` except that they do not invoke any virtual members of `rdbuf()` except `overflow()`.

to:

They may use other public members of `basic_ostream` except that they shall not invoke any virtual members of `rdbuf()` except `overflow()`, `xsputn()`, and `sync()`.

[Kona: the LWG believes this is a problem. Wish to ask Jerry or PJP why the standard is written this way.]

Post-Tokyo: Dietmar supplied wording at the request of the LWG. He comments: The rules can be made a little bit more specific if necessary by explicitly spelling out what virtuals are allowed to be called from what functions and eg to state specifically that `flush()` is allowed to call `sync()` while other functions are not.]

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

Section:: 27.6.2.5.4 [lib.ostream.inserters.character](#) **Status:** [Review](#) **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:

Change 27.6.2.5.4 ([lib.ostream.inserters.character](#)) paragraph 4 from:

Effects: Behaves like an formatted inserter (as described in `lib.ostream.formatted.reqmts`) of `out`. After a sentry object is constructed it inserts characters. The number of characters starting at `s` to be inserted is `traits::length(s)`. Padding is determined as described in `lib.facet.num.put.virtuals`. The `traits::length(s)` characters starting at `s` are widened using `out.widen` (`lib.basic.ios.members`). The widened characters and any required padding are inserted into `out`. Calls `width(0)`.

to:

Effects: Behaves like an formatted inserter (as described in `lib ostream.formatted.reqmts`) of `out`. After a sentry object is constructed it inserts characters. The number `len` of characters starting at `s` to be inserted is

- `traits::length(s)` if the second argument is of type `const charT*`
- `char_traits<char>::length(s)` if the second argument is of type `char` and `charT` is not `char`
- `char_traits<signed char>::length(s)` if the second argument is of type `signed char` and `charT` is not `signed char`
- `char_traits<unsigned char>::length(s)` if the second argument is of type `unsigned char` and `charT` is not `unsigned char`

Padding is determined as described in `lib.facet.num.put.virtuals`. The `len` characters starting at `s` are widened using `out.widen` (`lib.basic.ios.members`). The widened characters and any required padding are inserted into `out`. Calls `width(0)`.

[Kona: It is clear to the LWG there is a defect here. Dietmar will supply specific wording.]

[Post-Tokyo: Dietmar supplied the above wording.]

170. Inconsistent definition of `traits_type`

Section:: 27.7.4 [lib.stringstream](#) **Status:** [Ready](#) **Submitter:** Dietmar Kühl **Date:** 20 Jul 99

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

Proposed resolution:

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

```
typedef traits traits_type;
```

[Tokyo: Reviewed by the LWG.]

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

Section:: 27.8.1.4 [lib.filebuf.virtuals](#) **Status:** [Review](#) **Submitter:** Dietmar Kühl **Date:** 20 Jul 99

Overridden virtual functions, `seekpos()`

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

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

Plus the appropriate error handling, that is...

Proposed resolution:

Change the unnumbered paragraph in 27.8.1.4 (`lib.filebuf.virtuals`) before paragraph 14 from:

```
pos_type seekpos(pos_type sp, ios_base::openmode = ios_base::in | ios_base::out);
```

Alters the file position, if possible, to correspond to the position stored in `sp` (as described below).

- if `(which&ios_base::in)!=0`, set the file position to `sp`, then update the input sequence

- if `(which&ios_base::out)!=0`, then update the output sequence, write any unshift sequence, and set the file position to `sp`.

to:

```
pos_type seekpos(pos_type sp, ios_base::openmode = ios_base::in | ios_base::out);
```

Alters the file position, if possible, to correspond to the position stored in `sp` (as described below).
Altering the file position performs as follows:

1. if `(om & ios_base::out)!=0`, then update the output sequence and write any unshift sequence;
2. set the file position to `sp`;
3. if `(om & ios_base::in)!=0`, then update the input sequence;

where `om` is the open mode passed to the last call to `open()`. The operation fails if `is_open()` return false.

[Kona: Dietmar is working on a proposed resolution.]

[Post-Tokyo: Dietmar supplied the above wording.]

179. Comparison of `const_iterator`s to `iterators` doesn't work

Section: 23.1 [lib.container.requirements](#) **Status:** [Review](#) **Submitter:** Judy Ward **Date:** 2 Jul 1998

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

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

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

```
}
```

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

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

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

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

Proposed Resolution:

In section 23.1 ([lib.container.requirements](#)) after paragraph 7 add:

It is possible to mix iterators and `const_iterators` in iterator comparison operations.

[Kona: The LWG does wish the example to work. Judy will provide wording.]

[Post-Tokyo: Judy supplied the above wording at the request of the LWG.]

181. `make_pair()` unintended behavior

Section: 20.2.2 [lib.pairs](#) **Status:** [Review](#) **Submitter:** Andrew Koenig **Date:** 3 Aug 99

The claim has surfaced in Usenet that expressions such as

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

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

I doubt anyone intended that behavior...

Proposed resolution:

In 20.2 [[lib.utility](#)], paragraph 1 change the following declaration of `make_pair()`:

```
template <class T1, class T2> pair<T1,T2> make_pair(const T1&, const T2&);
```

to:

```
template <class T1, class T2> pair<T1,T2> make_pair(T1, T2);
```

In 20.2.2 [[lib.pairs](#)] paragraph 7 and the line before change:

```
template <class T1, class T2>
pair<T1, T2> make_pair(const T1& x, const T2& y);
```

to:

```
template <class T1, class T2>
pair<T1, T2> make_pair(T1 x, T2 y);
```

and add the following footnote to the effects clause:

According to 12.8 [class.copy], an implementation is permitted to not perform a copy of an argument, thus avoiding unnecessary copies.

[Kona: The LWG agreed that this is a probable defect, but would like to see fixes spelled out to verify the fix isn't worse than the problem.]

Two potential fixes were suggested by Matt Austern and Dietmar Kühl, respectively, 1) overloading with array arguments, and 2) use of a reference_traits class with a specialization for arrays.

Tokyo: Andy Koenig suggested changing to pass by value. In discussion, it appeared that this was a much smaller change to the standard than the other two suggestions, and any efficiency concerns were more than offset by the advantages of the solution. Two implementors reported that the proposed resolution passed their test suites.

Post-Tokyo: Nico Josuttis provided the above proposed resolution at the request of the LWG.]

182. Ambiguous references to size_t

Section: 17 [lib.library](#) **Status:** [Review](#) **Submitter:** Al Stevens **Date:** 15 Aug 99

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

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

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

Proposed resolution:

In 17.4.3.4 [[lib.replacement.functions](#)] paragraph 2: replace:

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

by:

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

In [[lib.allocator.requirements](#)] 20.1.5, paragraph 4: replace:

The typedef members `pointer`, `const_pointer`, `size_type`, and `difference_type` are required to be `T*`, `T const*`, `size_t`, and `ptrdiff_t`, respectively.

by:

The typedef members `pointer`, `const_pointer`, `size_type`, and `difference_type` are required to be `T*`, `T const*`, `std::size_t`, and `std::ptrdiff_t`, respectively.

In [lib.allocator.members] 20.4.1.1, paragraphs 3 and 6: replace:

3 Notes: Uses `::operator new(size_t)` (18.4.1).

6 Note: the storage is obtained by calling `::operator new(size_t)`, but it is unspecified when or how often this function is called. The use of `hint` is unspecified, but intended as an aid to locality if an implementation so desires.

by:

3 Notes: Uses `::operator new(std::size_t)` (18.4.1).

6 Note: the storage is obtained by calling `::operator new(std::size_t)`, but it is unspecified when or how often this function is called. The use of `hint` is unspecified, but intended as an aid to locality if an implementation so desires.

In [lib.char.traits.require] 21.1.1, paragraph 1: replace:

In Table 37, `X` denotes a Traits class defining types and functions for the character container type `CharT`; `c` and `d` denote values of type `CharT`; `p` and `q` denote values of type `const CharT*`; `s` denotes a value of type `CharT*`; `n`, `i` and `j` denote values of type `size_t`; `e` and `f` denote values of type `X::int_type`; `pos` denotes a value of type `X::pos_type`; and `state` denotes a value of type `X::state_type`.

by:

In Table 37, `X` denotes a Traits class defining types and functions for the character container type `CharT`; `c` and `d` denote values of type `CharT`; `p` and `q` denote values of type `const CharT*`; `s` denotes a value of type `CharT*`; `n`, `i` and `j` denote values of type `std::size_t`; `e` and `f` denote values of type `X::int_type`; `pos` denotes a value of type `X::pos_type`; and `state` denotes a value of type `X::state_type`.

In [lib.char.traits.require] 21.1.1, table 37: replace the return type of `X::length(p)`: `"size_t"` by `"std::size_t"`.

In [lib.std.iterator.tags] 24.3.3, paragraph 2: replace:

`typedef ptrdiff_t difference_type;`

by:

`typedef std::ptrdiff_t difference_type;`

In [lib.locale ctype] 22.2.1.1 put namespace `std` { ... } around the declaration of template `<class charT> class ctype`.

In [lib.iterator.traits] 24.3.1, paragraph 2 put namespace `std` { ... } around the declaration of:

```
template<class Iterator> struct iterator_traits
template<class T> struct iterator_traits<T*>
template<class T> struct iterator_traits<const T*>
```

Rationale:

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

change.

[Post-Tokyo: Nico Josuttis provided the above wording at the request of the LWG.]

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

Section: 27.6.3 [lib.std.manip](#) **Status:** [Review](#) **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:

Replace section 27.6.3 [lib.std.manip](#) except paragraph 1 with the following:

2- The type designated *smanip* in each of the following function descriptions is implementation-specified and may be different for each function.

```
smanip resetiosflags(ios_base::fmtflags mask);
```

-3- Returns: An object *s* of unspecified type such that if *out* is an instance of `basic_ostream<charT,traits>` then the expression `out<<s` behaves as if `f(s, mask)` were called, or if *in* is an instance of `basic_istream<charT,traits>` then the expression `in>>s` behaves as if `f(s, mask)` were called. The function *f* can be defined as:*

[Footnote: The expression `cin >> resetiosflags(ios_base::skipws)` clears `ios_base::skipws` in the format flags stored in the `basic_istream<charT,traits>` object *cin* (the same as `cin >> noskipws`), and the expression `cout << resetiosflags(ios_base::showbase)` clears `ios_base::showbase` in the format flags stored in the `basic_ostream<charT,traits>` object *cout* (the same as `cout << noshowbase`). --- end footnote]

```
ios_base& f(ios_base& str, ios_base::fmtflags mask)
{
```



```
// reset specified flags
str.setf(ios_base::fmtflags(0), mask);
return str;
}
```

The expression `out<<s` has type `basic_ostream<charT,traits>&` and value `out`. The expression `in>>s` has type `basic_istream<charT,traits>&` and value `in`.

```
smanip setiosflags(ios_base::fmtflags mask);
```

-4- Returns: An object `s` of unspecified type such that if `out` is an instance of `basic_ostream<charT,traits>` then the expression `out<<s` behaves as if `f(s, mask)` were called, or if `in` is an instance of `basic_istream<charT,traits>` then the expression `in>>s` behaves as if `f(s, mask)` were called. The function `f` can be defined as:

```
ios_base& f(ios_base& str, ios_base::fmtflags mask)
{
    // set specified flags
    str.setf(mask);
    return str;
}
```

The expression `out<<s` has type `basic_ostream<charT,traits>&` and value `out`. The expression `in>>s` has type `basic_istream<charT,traits>&` and value `in`.

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

-5- Returns: An object `s` of unspecified type such that if `out` is an instance of `basic_ostream<charT,traits>` then the expression `out<<s` behaves as if `f(s, base)` were called, `in` is an instance of `basic_istream<charT,traits>` then the expression `in>>s` behaves as if `f(s, base)` were called. The function `f` can be defined as:

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

The expression `out<<s` has type `basic_ostream<charT,traits>&` and value `out`. The expression `in>>s` has type `basic_istream<charT,traits>&` and value `in`.

```
smanip setfill(char_type c);
```

-6- Returns: An object `s` of unspecified type such that if `out` is (or is derived from) `basic_ostream<charT,traits>` and `c` has type `charT` then the expression `out<<s` behaves as if `f(s, c)` were called. The function `f` can be defined as:

```
template<class charT, class traits>
basic_ios<charT,traits>& f(basic_ios<charT,traits>& str, charT c)
{
    // set fill character
    str.fill(c);
    return str;
}
```

The expression `out<<s` has type `basic_ostream<charT,traits>&` and value `out`.

```
smanip setprecision(int n);
```

-7- Returns: An object *s* of unspecified type such that if *out* is an instance of `basic_ostream<charT,traits>` then the expression `out<<s` behaves as if `f(s, n)` were called, or if *in* is an instance of `basic_istream<charT,traits>` then the expression `in>>s` behaves as if `f(s, n)` were called. The function *f* can be defined as:

```
ios_base& f(ios_base& str, int n)
{
    // set precision
    str.precision(n);
    return str;
}
```

The expression `out<<s` has type `basic_ostream<charT,traits>&` and value *out*. The expression `in>>s` has type `basic_istream<charT,traits>&` and value *in*.

```
.smanip setw(int n);
```

-8- Returns: An object *s* of unspecified type such that if *out* is an instance of `basic_ostream<charT,traits>` then the expression `out<<s` behaves as if `f(s, n)` were called, or if *in* is an instance of `basic_istream<charT,traits>` then the expression `in>>s` behaves as if `f(s, n)` were called. The function *f* can be defined as:

```
ios_base& f(ios_base& str, int n)
{
    // set width
    str.width(n);
    return str;
}
```

The expression `out<<s` has type `basic_ostream<charT,traits>&` and value *out*. The expression `in>>s` has type `basic_istream<charT,traits>&` and value *in*.

[Kona: Andy Sawyer and Beman Dawes will work to improve the wording of the proposed resolution.]

Tokyo - The LWG noted that issue [216](#) involves the same paragraphs.

Post-Tokyo: The issues list maintainer combined the proposed resolution of this issue with the proposed resolution for issue [216](#) as they both involved the same paragraphs, and were so intertwined that dealing with them separately appear fraught with error.

The full text was supplied by Bill Plauger; it was cross checked against changes supplied by Andy Sawyer. It should be further checked by the LWG.]

184. `numeric_limits<bool>` wording problems

Section: 18.2.1.5 [lib.numeric.special](#) **Status:** [Review](#) **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:

Append to the end of 18.2.1.5 [[lib.numeric.special](#)]:

The specialization for bool shall be provided as follows:

```
namespace std {
    template<> class numeric_limits<bool> {
    public:
        static const bool is_specialized = true;
        static T min() throw() { return false; }
        static T max() throw() { return true; }

        static const int  digits = 1;
        static const int  digits10 = 0;
        static const bool is_signed = false;
        static const bool is_integer = true;
        static const bool is_exact = true;
        static const int  radix = 2;
        static T epsilon() throw() { return bool(0); }
        static T round_error() throw() { return bool(0); }

        static const int  min_exponent = 0;
        static const int  min_exponent10 = 0;
        static const int  max_exponent = 0;
        static const int  max_exponent10 = 0;

        static const bool has_infinity = false;
        static const bool has_quiet_NaN = false;
        static const bool has_signaling_NaN = false;
        static const float_denorm_style has_denorm = denorm_absent;
        static const bool has_denorm_loss = false;
        static T infinity() throw() { return bool(0); }
        static T quiet_NaN() throw() { return bool(0); }
        static T signaling_NaN() throw() { return bool(0); }
        static T denorm_min() throw() { return bool(0); }

        static const bool is_iec559 = false;
        static const bool is_bounded = false;
        static const bool is_modulo = false;

        static const bool traps = false;
        static const bool tinyness_before = false;
        static const float_round_style round_style = round_toward_zero;
    };
};
```

}

[Tokyo: The LWG desires wording that specifies exact values rather than more general wording in the original proposed resolution..

Post-Tokyo: At the request of the LWG in Tokyo, Nico Josuttis provided the above wording.]

185. Questionable use of term "inline"

Section: 20.3 [lib.function.objects](#) **Status:** [Review](#) **Submitter:** UK Panel **Date:** 26 Jul 99

Paragraph 4 of 20.3 [\[lib.function.objects\]](#) says:

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

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

Indeed both:

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

and

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

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

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

Proposed resolution:

In 20.3 [\[lib.function.objects\]](#) paragraph 1, remove the sentence:

They are important for the effective use of the library.

Remove 20.3 [\[lib.function.objects\]](#) paragraph 2, which reads:

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

In 20.3 [\[lib.function.objects\]](#) paragraph 4, remove the sentence:

The corresponding functions will inline the addition and the negation.

[Kona: The LWG agreed there was a defect.

Tokyo: The LWG crafted the proposed resolution.]

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

Section: 23.3.5.2 [lib.bitset.members](#) **Status:** [Review](#) **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 [[lib.template.bitset](#)] Para 1 Replace:

```
bitset<N>& set(size_t pos, int val = true );
```

With:

```
bitset<N>& set(size_t pos, bool val = true );
```

In 23.3.5.2 [[lib.bitset.members](#)] Para 12(.5) Replace:

```
bitset<N>& set(size_t pos, int val = 1 );
```

With:

```
bitset<N>& set(size_t pos, bool val = true );
```

[Kona: The LWG agrees with the description. Andy Sawyers will work on better P/R wording.

Post-Tokyo: Andy provided the above wording.]

197. `max_size()` underspecified

Section: 20.1.5 [lib allocator.requirements](#), 23.1 [lib.container.requirements](#) **Status:** [Review](#) **Submitter:** Andy Sawyer
Date: 21 Oct 99

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

Must the value returned from `max_size()` be meaningful?

Possible meanings identified in lib-6827:

- 1) The largest container the implementation can support given "best case" conditions - i.e. assume the run-time platform is "configured to the max", and no overhead from the program itself. This may possibly be determined at the point the library is written, but certainly no later than compile time.
- 2) The largest container the program could create, given "best case" conditions - i.e. same platform assumptions as (1), but take into account any overhead for executing the program itself. (or, roughly "storage=storage-sizeof(program)"). This does NOT include any resource allocated by the program. This may (or may not) be determinable at compile time.
- 3) The largest container the current execution of the program could create, given knowledge of the actual run-time

platform, but again, not taking into account any currently allocated resource. This is probably best determined at program start-up.

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

Proposed Resolution:

Change 20.1.5 [lib.allocator.requirements](#) table 32 `max_size()` wording from:

the largest value that can meaningfully be passed to `X::allocate`
to:
the value of the largest constant expression (5.19 [expr.const](#)) that could ever meaningfully be passed to `X::allocate`

Change 23.1 [lib.container.requirements](#) table 65 `max_size()` wording from:

`size()` of the largest possible container.
to:
the value of the largest constant expression (5.19 [expr.const](#)) that could ever meaningfully be returned by `X::size()`.

[Kona: The LWG informally discussed this and asked Andy Sawyer to submit an issue.

Tokyo: The LWG believes (1) above is the intended meaning.

Post-Tokyo: Beman Dawes supplied the above resolution at the request of the LWG. 21.3.3 [lib.string.capacity](#) was not changed because it references `max_size()` in 23.1. The term "compile-time" was avoided because it is not defined anywhere in the standard (even though it is used several places in the library clauses).]

198. Validity of pointers and references unspecified after iterator destruction

Section: 24.1 [lib.iterator.requirements](#) **Status:** [Open](#) **Submitter:** Beman Dawes **Date:** 3 Nov 99

Is a pointer or reference obtained from an iterator still valid after destruction of the iterator?

```
// assume iter is a dereferenceable iterator with value_type T

T& r = *iter;
T* p = &*iter;

// are r and p still valid at this point even though the iterators
// they were obtained from have been destroyed?
```

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

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

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

Proposed Resolution:

Add a new paragraph to 24.1 [lib.iterator.requirements](#):

Destruction of an iterator may invalidate pointers and references previously obtained from that iterator.

[Tokyo: The LWG reformulated the question purely in terms of iterators. The answer to the question is "no, pointers and references don't remain valid after iterator destruction." PJP explained that implementors use considerable care to avoid such ephemeral pointers and references. Several LWG members said that they thought that the standard did not actually specify the lifetime of pointers and references obtained from iterators, except possibly input iterators.

Post-Tokyo: The issue has been reformulated purely in terms of iterators.]

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

Section: 20.1.5 [lib.allocator.requirements](#) **Status:** [Ready](#) **Submitter:** Matt Austern **Date:** 19 Nov 99

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

Original proposed resolutions:

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

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

Proposed Resolution:

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

[Tokyo: The LWG says a key to understanding this issue is that the ultimate use of `allocate()` is to construct an iterator, and that iterator for zero length sequences must be the container's past-the-end representation. Since this already implies special case code, it would be over-specification to mandate the return value. Thus the LWG formulated the above proposed resolution.]

200. Forward iterator requirements don't allow constant iterators

Section: 24.1.3 [lib.forward.iterators](#) **Status:** [Open](#) **Submitter:** Matt Austern **Date:** 19 Nov 99

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

Proposed Resolution:

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

[Tokyo: The LWG believes this is the tip of a larger iceberg; there are multiple `const` problems with the STL portion of the library and that these should be addressed as a single package. Note that issue [180](#) has already been declared

NAD Future for that very reason.]

201. Numeric limits terminology wrong

Section: 18.2.1 [lib.limits](#) **Status:** [Ready](#) **Submitter:** Stephen Cleary **Date:** 21 Dec 1999

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

Proposed Resolution:

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

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

to:

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

Change 18.2.1 [lib.limits] para 1 from:

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

to:

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

Change 18.2.1 [lib.limits] para 2 from:

Specializations shall be provided for each fundamental type. . .

to:

Specializations shall be provided for each arithmetic type. . .

Change 18.2.1 [lib.limits] para 4 from:

Non-fundamental standard types. . .

to:

Non-arithmetic standard types. . .

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

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

to:

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

[Tokyo: Reviewed by the LWG.]

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

Section: 25.2.8 [lib.alg.unique](#) **Status:** [Open](#) **Submitter:** Andrew Koenig **Date:** 13 Jan 00

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

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

The example that raised this question is from Usenet:

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

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

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

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

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

Proposed Resolution:

Options:

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

Effects: Eliminates all but the first element `e` from every consecutive group of elements

referred to by the iterator `i` in the range `[first, last)` for which the following corresponding conditions hold: `e == *i` or `pred(e, *i) != false`.

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

A LWG member [*Nico Josuttis*] comments:

First, I agree that the current wording is simply wrong. However, to follow all [known] current implementations I propose [option 3 above].

[*Tokyo: The issue was discussed at length without reaching consensus.*

Straw vote:

*Option 1 - preferred by 2 people.
Option 2 - preferred by 0 people.
Option 3 - preferred by 3 people.
Many abstentions.]*

207. `ctype<char>` members return clause incomplete

Section: 22.2.1.3.2 [lib.facet.ctype.char.members](#) **Status:** [Open](#) **Submitter:** Robert Klarer **Date:** 2 Nov 99

Proposed Resolution:

Change the returns clause in 22.2.1.3.2 [lib.facet.ctype.char.members](#) paragraph 10 from:

Returns: `do_widen(low, high, to)`.

to:

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

Change the returns clause in 22.2.1.3.2 [lib.facet.ctype.char.members](#) paragraph 11 from:

Returns: `do_narrow(low, high, to)`.

to:

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

[*Post-Tokyo: This appears to be a duplicate of issue [153](#).*]

208. Unnecessary restriction on past-the-end iterators

Section: 24.1 [lib.iterators](#) **Status:** [Ready](#) **Submitter:** Stephen Cleary **Date:** 02 Feb 00

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

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

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

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

Proposed Resolution:

Change 24.1 [[lib.iterators](#)] paragraph 5, the last sentence, from:

Dereferenceable and past-the-end values are always non-singular.

to:

Dereferenceable values are always non-singular.

[Tokyo: After discussion of the meaning of "non-singular", and working out several examples, the LWG changed the proposed resolution to simply strike the words "and past-the-end".]

209. basic_string declarations inconsistent

Section: 21.3 [lib.basic.string](#) **Status:** [Ready](#) **Submitter:** Igor Stauder **Date:** 11 Feb 00

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

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

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

Proposed Resolution:

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

```
void push_back(charT c);

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

[Tokyo: Although the standard is in general not consistent in declaration style, the basic_string declarations are consistent other than the above. The LWG felt that this was sufficient reason to merit the change.]

210. distance first and last confused

Section: 25 [lib.algorithms](#) **Status:** [Ready](#) **Submitter:** Lisa Lippincott **Date:** 15 Feb 00

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

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

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

Proposed Resolution:

On the last line of paragraph 9 of section 25 [[lib.algorithms](#)] change "a-b" to "b-a".

[Tokyo: There are two ways to fix the defect; change the description to b-a or change the return to distance(b,a). The LWG preferred the former for consistency.]

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

Section: 21.3.7.9 [lib.string.io](#) **Status:** [Ready](#) **Submitter:** Scott Snyder **Date:** 4 Feb 00

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

This implies that the typical construction

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

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

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

Proposed Resolution:

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

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

[Tokyo: Reviewed by the LWG.]

212. Empty range behavior unclear for several algorithms

Section: 25.3.7 [lib.alg.min.max](#) **Status:** [Ready](#) **Submitter:** Nico Josuttis **Date:** 26 Feb 00

The standard doesn't specify what `min_element()` and `max_element()` shall return if the range is empty (first equals last). The usual implementations return last. This problem seems also apply to `partition()`, `stable_partition()`, `next_permutation()`, and `prev_permutation()`.

Proposed Resolution:

In 25.3.7 - Minimum and maximum [[lib.alg.min.max](#)], paragraphs 7 and 9 append: Returns last if `first==last`.

[Tokyo: The LWG looked in some detail at all of the above mentioned algorithms, but believes that except for `min_element()` and `max_element()` it is already clear that last is returned if `first == last`.]

214. `set::find()` missing `const` overload

Section: 23.3.3 23.3.4 [lib.set](#) **Status:** [Review](#) **Submitter:** Judy Ward **Date:** 28 Feb 00

The specification for the associative container requirements in Table 69 state that the `find` member function should "return iterator; `const_iterator` for constant a". The map and multimap container descriptions have two overloaded versions of `find`, but set and multiset do not, all they have is:

```
iterator find(const key_type & x) const;
```

Proposed Resolution:

Change the prototypes for `find()`, `lower_bound()`, `upper_bound()`, and `equal_ranger()` in section 23.3.3 [lib.set](#) and section 23.3.4 [lib.multiset](#) to each have two overloads:

```
iterator find(const key_type & x);
const_iterator find(const key_type & x) const;

iterator lower_bound(const key_type & x);
const_iterator lower_bound(const key_type & x) const;

iterator upper_bound(const key_type & x);
const_iterator upper_bound(const key_type & x) const;

pair<iterator, iterator> equal_range(const key_type & x);
pair<const_iterator, const_iterator> equal_range(const key_type & x) const;
```

[Tokyo: At the request of the LWG, Judy Ward provided wording extending the proposed resolution to `lower_bound`, `upper_bound`, and `equal_range`.]

216. `setbase` manipulator description flawed

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

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

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

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

`in>>s` behaves as if `f(s)` were called. Where `f` can be defined as:

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

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

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

Proposed Resolution:

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

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

Post-Tokyo: The resolution was combined with issue [183](#) as they affect the same text.]

217. Facets example (Classifying Japanese characters) contains errors

Section: 22.2.8 [lib.facets.examples](#) **Status:** [Ready](#) **Submitter:** Martin Sebor **Date:** 29 Feb 00

The example in 22.2.8, paragraph 11 contains the following errors:

- 1) The member function `My::Jctype::is_kanji()` is non-const; the function must be const in order for it to be callable on a const object (a reference to which which is what `std::use_facet<>()` returns).
- 2) In file `filt.C`, the definition of `Jctype::id` must be qualified with the name of the namespace `My`.
- 3) In the definition of `loc` and subsequently in the call to `use_facet<>()` in `main()`, the name of the facet is misspelled: it should read `My::Jctype` rather than `My::JCType`.

Proposed Resolution:

Replace the "Classifying Japanese characters" example in 22.2.8, paragraph 11 with the following:

```
#include <locale>

namespace My {
    using namespace std;
    class Jctype : public locale::facet {
    public:
        static locale::id id;        // required for use as a new locale
    }
    facet
        bool is_kanji (wchar_t c) const;
        Jctype() {}
```

```

    protected:
        ~JCTYPE() {}
    };
}

// file: filt.C
#include <iostream>
#include <locale>
#include "jctype"           // above
std::locale::id My::JCTYPE::id; // the static JCTYPE member
                              // declared above.

int main()
{
    using namespace std;
    typedef ctype<wchar_t> wctype;
    locale loc(locale(""),           // the user's preferred locale...
               new My::JCTYPE);      // and a new feature ...
    wchar_t c = use_facet<wctype>(loc).widen('!');
    if (use_facet<My::JCTYPE>(loc).is_kanji(c))
        cout << "no it isn't!" << endl;
    return 0;
}

```

[Tokyo: Reviewed by the LWG.]

220. ~ios_base() usage valid?

Section: 27.4.2.7 [lib.ios.base.cons](#) **Status:** [Ready](#) **Submitter:** Jonathan Schilling, Howard Hinnant **Date:** 13 Mar 00

The pre-conditions for the ios_base destructor are described in 27.4.2.7 paragraph 2:

Effects: Destroys an object of class ios_base. Calls each registered callback pair (fn,index) (27.4.2.6) as (*fn)(erase_event,*this,index) at such time that any ios_base member function called from within fn has well defined results.

But what is not clear is: If no callback functions were ever registered, does it matter whether the ios_base members were ever initialized?

For instance, does this program have defined behavior:

```

#include <ios>

class D : public std::ios_base { };

int main() { D d; }

```

It seems that registration of a callback function would surely affect the state of an ios_base. That is, when you register a callback function with an ios_base, the ios_base must record that fact somehow.

But if after construction the ios_base is in an indeterminate state, and that state is not made determinate before the destructor is called, then how would the destructor know if any callbacks had indeed been registered? And if the number of callbacks that had been registered is indeterminate, then is not the behavior of the destructor undefined?

By comparison, the basic_ios class description in 27.4.4.1 paragraph 2 makes it explicit that destruction before initialization results in undefined behavior.

Proposed Resolution:

Modify 27.4.2.7 paragraph 1 from

Effects: Each `ios_base` member has an indeterminate value after construction.

to

Effects: Each `ios_base` member has an indeterminate value after construction. These members must be initialized by calling `basic_ios::init`. If an `ios_base` object is destroyed before these initializations have taken place, the behavior is undefined.

[Tokyo: Reviewed by the LWG and accepted after changing "calling `ios_base` member functions." to "calling `basic_ios::init`".]

221. `num_get<>::do_get` stage 2 processing broken

Section: 22.2.2.1.2 [lib.facet.num.get.virtuals](#) **Status:** [Review](#) **Submitter:** Matt Austern **Date:** 14 Mar 00

Stage 2 processing of numeric conversion is broken.

Table 55 in 22.2.2.1.2 says that when `basefield` is 0 the integral conversion specifier is `%i`. A `%i` specifier determines a number's base by its prefix (0 for octal, 0x for hex), so the intention is clearly that a 0x prefix is allowed. Paragraph 8 in the same section, however, describes very precisely how characters are processed. (It must be done "as if" by a specified code fragment.) That description does not allow a 0x prefix to be recognized.

Very roughly, stage 2 processing reads a `char_type` `ct`. It converts `ct` to a `char`, not by using `narrow` but by looking it up in a translation table that was created by widening the string literal `"0123456789abcdefABCDEF+-"`. The character `"x"` is not found in that table, so it can't be recognized by stage 2 processing.

Proposed Resolution:

In 22.2.2.1.2 paragraph 8, replace the line:

```
static const char src[] = "0123456789abcdefABCDEF+-";
```

with the line:

```
static const char src[] = "0123456789abcdefxABCDEFX+-";
```

222. Are throw clauses necessary if a throw is already implied by the effects clause?

Section: 17.3.1.3 [lib.structure.specifications](#) **Status:** [Ready](#) **Submitter:** Judy Ward **Date:** 17 Mar 00

Section 21.3.6.8 describes the `basic_string::compare` function this way:

```
21.3.6.8 - basic_string::compare [lib.string::compare]

int compare(size_type pos1, size_type n1,
            const basic_string<charT, traits, Allocator>& str,
            size_type pos2, size_type n2) const;
```


-4- Returns:

```
basic_string<charT,traits,Allocator>(*this,pos1,n1).compare(
    basic_string<charT,traits,Allocator>(str,pos2,n2)) .
```

and the constructor that's implicitly called by the above is defined to throw an out-of-range exception if `pos > str.size()`. See section [21.3.1](#) paragraph 4.

On the other hand, the compare function descriptions themselves don't have "Throws: " clauses and according to 17.3.1.3, paragraph 3, elements that do not apply to a function are omitted.

So it seems there is an inconsistency in the standard -- are the "Effects" clauses correct, or are the "Throws" clauses missing?

Proposed Resolution:

In 17.3.1.3 [[lib.structure.specifications](#)] paragraph 3, the footnote 148 attached to the sentence "Descriptions of function semantics contain the following elements (as appropriate):", insert the word "further" so that the foot note reads:

To save space, items that do not apply to a function are omitted. For example, if a function does not specify any further preconditions, there will be no "Requires" paragraph.

[Tokyo: First it was observed that the standard is somewhat inconsistent, but that a failure to note a throw condition in a throws clause does not grant permission not to throw. Then it was noted that the inconsistent wording is in a footnote, and thus non-normative. The proposed resolution from the LWG clarifies the footnote.]

223. reverse algorithm should use `iter_swap` rather than `swap`

Section: 25.2.9 [lib.alg.reverse](#) **Status:** [Ready](#) **Submitter:** Dave Abrahams **Date:** 21 Mar 00

Shouldn't the effects say "applies `iter_swap` to all pairs..."?

Proposed Resolution:

In 25.2.9 [lib.alg.reverse](#), replace:

Effects: For each non-negative integer $i \leq (\text{last} - \text{first})/2$, applies swap to all pairs of iterators $\text{first} + i$, $(\text{last} - i) - 1$.

with:

Effects: For each non-negative integer $i \leq (\text{last} - \text{first})/2$, applies `iter_swap` to all pairs of iterators $\text{first} + i$, $(\text{last} - i) - 1$.

[Tokyo: Reviewed by the LWG.]

224. `clear()` complexity for associative containers refers to undefined N

Section: 23.1.2 [lib.associative.reqmts](#) **Status:** [Ready](#) **Submitter:** Ed Brey **Date:** 23 Mar 00

In the associative container requirements table in 23.1.2 paragraph 7, `a.clear()` has complexity " $\log(\text{size}()) + N$ ".

However, the meaning of N is not defined.

Proposed Resolution:

In the associative container requirements table in 23.1.2 paragraph 7, the complexity of `a.clear()`, change "N" to "size()".

[Tokyo: Reviewed by the LWG. Proposed resolution changed after discussion of how complexity is described in the standard. It was noted that the standard does not always use "big-O notation" in the strict sense.]

225. `std::` algorithms use of other unqualified algorithms

Section: 17.4.4.3 [lib.global.functions](#), 25 [lib.algorithms](#) **Status:** [Open](#) **Submitter:** Dave Abrahams **Date:** 01 Apr 00

Are algorithms in `std::` allowed to use other algorithms without qualification, so functions in user namespaces might be found through Koenig lookup?

For example, a popular standard library implementation includes this implementation of `std::unique`:

```
namespace std {
    template <class _ForwardIter>
    _ForwardIter unique(_ForwardIter __first, _ForwardIter __last) {
        __first = adjacent_find(__first, __last);
        return unique_copy(__first, __last, __first);
    }
}
```

Imagine two users on opposite sides of town, each using `unique` on his own sequences bounded by `my_iterator`. User1 looks at his standard library implementation and says, "I know how to implement a more efficient `unique_copy` for `my_iterator`", and writes:

```
namespace user1 {
    class my_iterator;
    // faster version for my_iterator
    my_iterator unique_copy(my_iterator, my_iterator, my_iterator);
}
```

`user1::unique_copy()` is selected by Koenig lookup, as he intended.

User2 has other needs, and writes:

```
namespace user2 {
    class my_iterator;
    // Returns true iff *c is a unique copy of *a and *b.
    bool unique_copy(my_iterator a, my_iterator b, my_iterator c);
}
```

User2 is shocked to find later that his fully-qualified use of `std::unique(user2::my_iterator, user2::my_iterator, user2::my_iterator)` fails to compile (if he's lucky). Looking in the standard, he sees the following Effects clause for `unique()`:

Effects: Eliminates all but the first element from every consecutive group of equal elements referred to by the iterator `i` in the range `[first, last)` for which the following corresponding conditions hold: `*i == *(i - 1)` or `pred(*i, *(i - 1)) != false`

The standard gives user2 absolutely no reason to think he can interfere with `std::unique` by defining names in namespace `user2`. His standard library has been built with the template export feature, so he is unable to inspect the

implementation. User1 eventually compiles his code with another compiler, and his version of `unique_copy` silently stops being called. Eventually, he realizes that he was depending on an implementation detail of his library and had no right to expect his `unique_copy()` to be called portably.

On the face of it, and given above scenario, it may seem obvious that the implementation of `unique()` shown is non-conforming because it uses `unique_copy()` rather than `::std::unique_copy()`. Most standard library implementations, however, seem to disagree with this notion.

[Tokyo: Steve Adamczyk from the core working group indicates that "std::" is sufficient; leading "::" qualification is not required because any namespace qualification is sufficient to suppress Koenig lookup.]

Proposed Resolution:

Add a paragraph and a note at the end of 17.4.4.3 [lib.global.functions](#):

Unless otherwise specified, no global or non-member function in the standard library shall use a function from another namespace which is found through *argument-dependent name lookup* ([basic.lookup.koenig](#)).

[Note: the phrase "unless otherwise specified" is intended to allow Koenig lookup in cases like that of `ostream_iterators`:

Effects:

```
*out_stream << value;
if(delim != 0) *out_stream << delim;
return (*this);
```

--end note]

[Tokyo: The LWG agrees that this is a defect in the standard, but is as yet unsure if the proposed resolution is the best solution. Furthermore, the LWG believes that the same problem of unqualified library names applies to wording in the standard itself, and has opened issue [229](#) accordingly. Any resolution of issue [225](#) should be coordinated with the resolution of issue [229](#).]

226. User supplied specializations or overloads of namespace std function templates

Section: 17.4.3.1 [lib.reserved.names](#) **Status:** [Open](#) **Submitter:** Dave Abrahams **Date:** 01 Apr 00

The issues are:

1. How can a 3rd party library implementor (lib1) write a version of a standard algorithm which is specialized to work with his own class template?
2. How can another library implementor (lib2) write a generic algorithm which will take advantage of the specialized algorithm in lib1?

This appears to be the only viable answer under current language rules:

```
namespace lib1
{
    // arbitrary-precision numbers using T as a basic unit
    template <class T>
    class big_num { //...
```

```

};

// defining this in namespace std is illegal (it would be an
// overload), so we hope users will rely on Koenig lookup
template <class T>
void swap(big_int<T>&, big_int<T>&);
}

#include <algorithm>
namespace lib2
{
    template <class T>
    void generic_sort(T* start, T* end)
    {
        ...
        // using-declaration required so we can work on built-in types
        using std::swap;
        // use Koenig lookup to find specialized algorithm if available
        swap(*x, *y);
    }
}

```

This answer has some drawbacks. First of all, it makes writing lib2 difficult and somewhat slippery. The implementor needs to remember to write the using-declaration, or generic_sort will fail to compile when T is a built-in type. The second drawback is that the use of this style in lib2 effectively "reserves" names in any namespace which defines types which may eventually be used with lib2. This may seem innocuous at first when applied to names like swap, but consider more ambiguous names like unique_copy() instead. It is easy to imagine the user wanting to define these names differently in his own namespace. A definition with semantics incompatible with the standard library could cause serious problems (see issue [225](#)).

Why, you may ask, can't we just partially specialize std::swap()? It's because the language doesn't allow for partial specialization of function templates. If you write:

```

namespace std
{
    template <class T>
    void swap(lib1::big_int<T>&, lib1::big_int<T>&);
}

```

You have just overloaded std::swap, which is illegal under the current language rules. On the other hand, the following full specialization is legal:

```

namespace std
{
    template <>
    void swap(lib1::other_type&, lib1::other_type&);
}

```

[This issue reflects concerns raised by the "Namespace issue with specialized swap" thread on comp.lang.c++.moderated. A similar set of concerns was earlier raised on the boost.org mailing list and the ACCU-general mailing list. Also see library reflector message c++std-lib-7354.]

Proposed Resolution:

[Tokyo: Summary, "There is no conforming way to extend std::swap for user defined templates." The LWG agrees that there is a problem. Would like more information before proceeding. This may be a core issue.

It was also noted that submissions regarding this issue have been received from several sources, but too late to be integrated into the issues list.

Post-Tokyo: A paper with several proposed resolutions, J16/00-0029==WG21/N1252, "Shades of namespace std functions " by Alan Griffiths, is in the Post-Tokyo mailing. It should be considered a part of this issue.]

227. std::swap() should require CopyConstructible or DefaultConstructible arguments

Section: 25.2.2 [lib.alg.swap](#) **Status:** [Ready](#) **Submitter:** Dave Abrahams **Date:** 09 Apr 00

25.2.2 reads:

```
template<class T> void swap(T& a, T& b);
```

Requires: Type T is Assignable (`_lib.container.requirements_`).

Effects: Exchanges values stored in two locations.

The only reasonable** generic implementation of swap requires construction of a new temporary copy of one of its arguments:

```
template<class T> void swap(T& a, T& b);
{
    T tmp(a);
    a = b;
    b = tmp;
}
```

But a type which is only Assignable cannot be swapped by this implementation.

**Yes, there's also an unreasonable implementation which would require T to be DefaultConstructible instead of CopyConstructible. I don't think this is worthy of consideration:

```
template<class T> void swap(T& a, T& b);
{
    T tmp;
    tmp = a;
    a = b;
    b = tmp;
}
```

Proposed Resolution:

Change 25.2.2 paragraph 1 from:

Requires: Type T is Assignable (23.1).

to:

Requires: Type T is CopyConstructible (20.1.3) and Assignable (23.1)

[Tokyo: Reviewed by the LWG. Also see issue [230](#), identifying other places in the standard where Assignable is specified without also specifying CopyConstructible.]

228. Incorrect specification of "..._byname" facets

Section: 22.2 [lib.locale.categories](#) **Status:** [New](#) **Submitter:** Dietmar Kühl **Date:** 20 Apr 00

The sections 22.2.1.2 ([lib.locale ctype.byname](#)), 22.2.1.4 ([lib.locale ctype.byname.special](#)), 22.2.1.6 ([lib.locale codecvt.byname](#)), 22.2.3.2 ([lib.locale numpunct.byname](#)), 22.2.4.2 ([lib.locale collate.byname](#)), 22.2.5.4 ([lib.locale time.put.byname](#)), 22.2.6.4 ([lib.locale moneypunct.byname](#)), and 22.2.7.2 ([lib.locale messages.byname](#)) overspecify the definitions of the "..._byname" classes by listing a bunch of virtual functions. At the same time, no semantics of these functions are defined. Real implementations do not define these functions because the functional part of the facets is actually implemented in the corresponding base classes and the constructor of the "..._byname" version just provides suitable data used by these implementations. For example, the 'numpunct' methods just return values from a struct. The base class uses a statically initialized struct while the derived version reads the contents of this struct from a table. However, no virtual function is defined in 'numpunct_byname'.

For most classes this does not impose a problem but specifically for 'ctype' it does: The specialization for 'ctype_byname<char>' is required because otherwise the semantics would change due to the virtual functions defined in the general version for 'ctype_byname': In 'ctype<char>' the method 'do_is()' is not virtual but it is made virtual in both 'ctype<cT>' and 'ctype_byname<cT>'. Thus, a class derived from 'ctype_byname<char>' can tell whether this class is specialized or not under the current specification: Without the specialization, 'do_is()' is virtual while with specialization it is not virtual.

Proposed Resolution:

Change section 22.2.1.2 ([lib.locale ctype.byname](#)) to become:

```
namespace std {
    template <class charT>
    class ctype_byname : public ctype<charT> {
    public:
        typedef ctype<charT>::mask mask;
        explicit ctype_byname(const char*, size_t refs = 0);
    protected:
        ~ctype_byname();                // virtual
    };
}
```

Change section 22.2.1.4 ([lib.locale ctype.byname.special](#)) to become:

```
namespace std {
    template <> class ctype_byname<char> : public ctype<char> {
    public:
        explicit ctype_byname(const char*, size_t refs = 0);
    protected:
        ~ctype_byname();                // virtual
    };
}
```

Change section 22.2.1.6 ([lib.locale codecvt.byname](#)) to become:

```
namespace std {
    template <class internT, class externT, class stateT>
    class codecvt_byname : public codecvt<internT, externT, stateT> {
    public:
        explicit codecvt_byname(const char*, size_t refs = 0);
    protected:
        ~codecvt_byname();                // virtual
    };
}
```

Change section 22.2.3.2 ([lib.locale numpunct.byname](#)) to become:

```
namespace std {
```

```

template <class charT>
class numpunct_byname : public numpunct<charT> {
// this class is specialized for char and wchar_t.
public:
    typedef charT          char_type;
    typedef basic_string<charT> string_type;
    explicit numpunct_byname(const char*, size_t refs = 0);
protected:
    ~numpunct_byname();          // virtual
};
}

```

Change section 22.2.4.2 (lib.locale.collate.byname) to become:

```

namespace std {
    template <class charT>
    class collate_byname : public collate<charT> {
    public:
        typedef basic_string<charT> string_type;
        explicit collate_byname(const char*, size_t refs = 0);
    protected:
        ~collate_byname();          // virtual
    };
}

```

Change section 22.2.5.2 (lib.locale.time.get.byname) to become:

```

namespace std {
    template <class charT, class InputIterator = istreambuf_iterator<charT> >
    class time_get_byname : public time_get<charT, InputIterator> {
    public:
        typedef time_base::dateorder dateorder;
        typedef InputIterator          iter_type

        explicit time_get_byname(const char*, size_t refs = 0);
    protected:
        ~time_get_byname();          // virtual
    };
}

```

Change section 22.2.5.4 (lib.locale.time.put.byname) to become:

```

namespace std {
    template <class charT, class OutputIterator = ostreambuf_iterator<charT> >
    class time_put_byname : public time_put<charT, OutputIterator>
    {
    public:
        typedef charT          char_type;
        typedef OutputIterator iter_type;

        explicit time_put_byname(const char*, size_t refs = 0);
    protected:
        ~time_put_byname();          // virtual
    };
}

```

Change section 22.2.6.4 (lib.locale.money.punct.byname) to become:

```

namespace std {
    template <class charT, bool Intl = false>
    class money_punct_byname : public money_punct<charT, Intl> {
    public:
        typedef money_base::pattern pattern;
        typedef basic_string<charT> string_type;

```

```

        explicit moneypunct_byname(const char*, size_t refs = 0);
protected:
    ~moneypunct_byname();           // virtual
    };
}

```

Change section 22.2.7.2 ([lib.locale.messages.byname](#)) to become:

```

namespace std {
    template <class charT>
    class messages_byname : public messages<charT> {
    public:
        typedef messages_base::catalog catalog;
        typedef basic_string<charT>    string_type;

        explicit messages_byname(const char*, size_t refs = 0);
protected:
        ~messages_byname();           // virtual
        virtual catalog do_open(const basic_string<char>&, const locale&) const;
        virtual string_type do_get(catalog, int set, int msgid,
                                   const string_type& default) const;
        virtual void do_close(catalog) const;
    };
}

```

Remove section 22.2.1.4 ([lib.locale ctype.byname.special](#)) completely (because in this case only those members are defined to be virtual which are defined to be virtual in 'ctype<CT>'.)

[Post-Tokyo: Dietmar Kühl submitted this issue at the request of the LWG to solve the underlying problems raised by issue [138](#).]

229. Unqualified references of other library entities

Section: 17.4.1.1 [lib.contents](#) **Status:** [New](#) **Submitter:** Steve Clamage **Date:** 19 Apr 00

Throughout the library chapters, the descriptions of library entities refer to other library entities without necessarily qualifying the names.

For example, section 25.2.2 "Swap" describes the effect of `swap_ranges` in terms of the unqualified name "swap". This section could reasonably be interpreted to mean that the library must be implemented so as to do a lookup of the unqualified name "swap", allowing users to override any `::std::swap` function when Koenig lookup applies.

Although it would have been best to use explicit qualification with `::std::` throughout, too many lines in the standard would have to be adjusted to make that change in a Technical Corrigendum.

Proposed Resolution:

To section 17.4.1.1 "Library contents" Add the following paragraph:

Whenever a name `x` defined in the standard library is mentioned, the name `x` is assumed to be fully qualified as `::std::x`, unless explicitly described otherwise. For example, if the Effects section for library function `F` is described as calling library function `G`, the function `::std::G` is meant.

[Post-Tokyo: Steve Clamage submitted this issue at the request of the LWG to solve a problem in the standard itself similar to the problem within implementations of library identified by issue [225](#). Any resolution of issue [225](#) should be coordinated with the resolution of issue [229](#).]

230. Assignable specified without also specifying CopyConstructible

Section: 17 [lib.library](#) **Status:** [New](#) **Submitter:** Beman Dawes **Date:** 26 Apr 00

Issue [227](#) identified an instance (std::swap) where Assignable was specified without also specifying CopyConstructible. The LWG asked that the standard be searched to determine if the same defect existed elsewhere.

There are a number of places (see proposed resolution below) where Assignable is specified without also specifying CopyConstructible. There are also several cases where both are specified. For example, 26.4.1 [lib.accumulate].

Proposed Resolution:

In [lib.container.requirements] 23.1 table 65 for value_type: change "T is Assignable" to "T is CopyConstructible and Assignable"

In [lib.associative.reqmts] 23.1.2 table 69 X::key_type; change "Key is Assignable" to "Key is CopyConstructible and Assignable"

In [lib.input.iterators] 24.1.1 paragraph 3, which reads:

[Note: For input iterators, a == b does not imply ++a == ++b. (Equality does not guarantee the substitution property or referential transparency.) Algorithms on input iterators should never attempt to pass through the same iterator twice. They should be single pass algorithms. Value type T is not required to be an Assignable type (23.1). These algorithms can be used with istreams as the source of the input data through the istream_iterator class.]

Change "... not required to be an Assignable type (23.1)" to "... not required to be a CopyConstructible (20.1.3) or Assignable type (23.1)".

In [lib.output.iterators] 24.1.2 paragraph 1, change:

A class or a built-in type X satisfies the requirements of an output iterator if X is an Assignable type (23.1) and also the following expressions are valid, as shown in Table 73:

to:

A class or a built-in type X satisfies the requirements of an output iterator if X is a CopyConstructible (20.1.3) and Assignable type (23.1) and also the following expressions are valid, as shown in Table 73:

In [lib.alg.replace] 25.2.4 paragraph 1 and 4 respectively, change:

1 Requires: Type T is Assignable (23.1) (and, for replace(), EqualityComparable (20.1.1)).

4 Requires: Type T is Assignable (23.1) (and, for replace_copy(), EqualityComparable

to:

1 Requires: Type T is CopyConstructible (20.1.3) and Assignable (23.1) (and, for replace(), EqualityComparable (20.1.1)).

4 Requires: Type T is CopyConstructible (20.1.3) and Assignable (23.1) (and, for replace_copy(), EqualityComparable

In [lib.alg.fill] 25.2.5 paragraph 1, change:

1 Requires: Type T is Assignable (23.1), Size is convertible to an integral type (4.7, 12.3).

to:

1 Requires: Type T is CopyConstructible (20.1.3) and Assignable (23.1), Size is convertible to an integral type (4.7, 12.3).

[Post-Tokyo: Beman Dawes submitted this issue at the request of the LWG .

He asks that the [lib.alg.replace] 25.2.4 and [lib.alg.fill] 25.2.5 changes be studied carefully, as it is not clear that CopyConstructible is really a requirement and may be overspecification.]

231. Precision in iostream?

Section: 22.2.2.2.2 [lib.facet.num.put.virtuals](#) **Status:** [New](#) **Submitter:** James Kanze, Stephen Clamage **Date:** 25 Apr 00

What is the following program supposed to output?

```
#include <iostream>

int
main()
{
    std::cout.setf( std::ios::scientific , std::ios::floatfield ) ;
    std::cout.precision( 0 ) ;
    std::cout << 1.23 << '\n' ;
    return 0 ;
}
```

From my C experience, I would expect "1e+00"; this is what printf("%.0e" , 1.23) ; does. G++ outputs "1.000000e+00".

The only indication I can find in the standard is 22.2.2.2.2/11, where it says "For conversion from a floating-point type, if (flags & fixed) != 0 or if str.precision() > 0, then str.precision() is specified in the conversion specification." This is an obvious error, however, fixed is not a mask for a field, but a value that a multi-bit field may take -- the results of and'ing fmtflags with ios::fixed are not defined, at least not if ios::scientific has been set. G++'s behavior corresponds to what might happen if you do use (flags & fixed) != 0 with a typical implementation (floatfield == 3 << something, fixed == 1 << something, and scientific == 2 << something).

Presumably, the intent is either (flags & floatfield) != 0, or (flags & floatfield) == fixed; the first gives something more or less like the effect of precision in a printf floating point conversion. Only more or less, of course. In order to implement printf formatting correctly, you must know whether the precision was explicitly set or not. Say by initializing it to -1, instead of 6, and stating that for floating point conversions, if precision < -1, 6 will be used, for fixed point, if precision < -1, 1 will be used, etc. Plus, of course, if precision == 0 and flags & floatfield == 0, 1 should be = used. But it probably isn't necessary to emulate all of the anomalies of printf:-).

Proposed Resolution:

232. "depends" poorly defined in 17.4.3.1

Section: 17.4.3.1 [lib.reserved.names](#) **Status:** [New](#) **Submitter:** Peter Dimov **Date:** 18 Apr 00

17.4.3.1/1 uses the term "depends" to limit the set of allowed specializations of standard templates to those that "depend on a user-defined name of external linkage."

This term, however, is not adequately defined, making it possible to construct a specialization that is, I believe, technically legal according to 17.4.3.1/1, but that specializes a standard template for a built-in type such as 'int'.

The following code demonstrates the problem:

```
#include <algorithm>

template<class T> struct X
{
    typedef T type;
};

namespace std
{
    template<> void swap(::X<int>::type& i, ::X<int>::type& j);
}
```

Proposed Resolution

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