

Doc. no. J16/00-0046
WG21 N1269
Date: 10 Nov 2000
Project: Programming Language C++
Reply to: Matt Austern <austern@research.att.com>

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

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://www.research.att.com/~austern/csc/faq.html>. Public discussion of C++ Standard related issues occurs on [news:comp.std.c++](news:comp.std.c++.).

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

Revision history

- R16: post-Toronto mailing; reflects actions taken in Toronto. Added issues 265-277.
- R15: pre-Toronto mailing. Added issues 233-264. Some small HTML formatting changes so that we pass Weblint tests.
- 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

23. Num_get overflow result

Section: 22.2.2.1.2 [lib.facet.num.get.virtuals](#) **Status:** [Open](#) **Submitter:** Nathan Myers **Date:** 6 Aug 98

The current description of numeric input does not account for the possibility of overflow. This is an implicit result of changing the description to rely on the definition of `scanf()` (which fails to report overflow), and conflicts with the documented behavior of traditional and current implementations.

Users expect, when reading a character sequence that results in a value unrepresentable in the specified type, to have an error reported. The standard as written does not permit this.

Further comments from Dietmar:

I don't feel comfortable with the proposed resolution to issue 23: It kind of simplifies the issue too much. Here is what is going on:

Currently, the behavior of numeric overflow is rather counter intuitive and hard to trace, so I will describe it briefly:

- According to 22.2.2.1.2 ([lib.facet.num.get.virtuals](#)) paragraph 11 `failbit` is set if `scanf()` would return an input error; otherwise a value is converted to the rules of `scanf`.
- `scanf()` is defined in terms of `fscanf()`.
- `fscanf()` returns an input failure if during conversion no character matching the conversion specification could be extracted before reaching EOF. This is the only reason for `fscanf()` to fail due to an input error and clearly does not apply to the case of overflow.

- Thus, the conversion is performed according to the rules of `fscanf()` which basically says that `strtod`, `strtol()`, etc. are to be used for the conversion.
- The `strtod()`, `strtol()`, etc. functions consume as many matching characters as there are and on overflow continue to consume matching characters but also return a value identical to the maximum (or minimum for signed types if there was a leading minus) value of the corresponding type and set `errno` to `ERANGE`.
- Thus, according to the current wording in the standard, overflows can be detected! All what is to be done is to check `errno` after reading an element and, of course, clearing `errno` before trying a conversion. With the current wording, it can be detected whether the overflow was due to a positive or negative number for signed types.

Now the proposed resolution results in not modifying the value passed as last argument if an overflow is encountered but `failbit` is set. Checking `errno` for `ERANGE` still allows for detection of an overflow but not what the sign was.

Actually, my problem is not that much with the sign but this is at least making things worse... My problem is more that it is still necessary to check `errno` for the error discription. Thus, I propose the following resolution:

Change paragraph 11 from

-11- Stage 3: The result of stage 2 processing can be one of

- A sequence of `chars` has been accumulated in stage 2 that is converted (according to the rules of `scanf()`) to a value of the type of `val`. This value is stored in `val` and `ios_base::goodbit` is stored in `err`.
- The sequence of `chars` accumulated in stage 2 would have caused `scanf` to report an input failure. `ios_base::failbit` is assigned to `err`.

to become

-11- Stage 3: The result of stage 2 processing can be one of

- A sequence of `chars` has been accumulated in stage 2 that is converted (according to the rules of `scanf()`) to a value of the type of `val`. This value is stored in `val`. If the conversion reported an overflow error for the type of `val` (ie. `errno` would be set to `ERANGE` by the used conversion function) then `ios_base::failbit` is stored in `err`, otherwise `ios_base::goodbit` is stored in `err`.
- The sequence of `chars` accumulated in stage 2 would have caused `scanf` to report an input failure. `ios_base::failbit` is assigned to `err`.

With this definition, overflow can be detected easily by storing a value different from the maximum value in `val` and checking whether this value was modified in case `failbit` is set: If it was, there was an overflow error, otherwise some other input error occurred (under the conditions for the second bullet `val` is not changed).

Proposed Resolution:

In 22.2.2.1.2 [[lib.facet.num.get.virtuals](#)], paragraph 11, second bullet item, change

The sequence of `chars` accumulated in stage 2 would have caused `scanf` to report an input failure.

to

The sequence of chars accumulated in stage 2 would have caused scanf to report an input failure, or the value of the sequence cannot be represented in the type of `_val_`.

[post-Toronto: "cannot be represented" is probably wrong: infinity can be represented on an IEC559 platform, but 0.1 cannot be represented exactly. However, the alternate proposal may be wrong as well. It's not clear whether overflow (and underflow?) should always be treated as errors. This issue requires much more thought.]

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 istreams 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 istream objects (27.3) are synchronized and otherwise returns `false`.

to:

`true` if the previous state of the standard istream 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.]

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

Section: 22.2.1.5 [lib.locale.codecvt](#) **Status:** [Open](#) **Submitter:** Matt Austern **Date:** 25 Sep 98

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

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

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

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

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

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

Proposed Resolution:

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

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 1998

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;

[Toronto: The issue was clarified. If characters are extracted from the streambuf (e.g. with `sgetc` or with `streambuf_iterator`), then there is no issue: a streambuf can signal failure only by throwing an exception or by returning `eof`. So the question is whether the string input operations are supposed to obtain each character using the streambuf, or using an istream member function such as `get()`. The answer has implications for `gcount`.

We already have an explicit answer for the input operations of clause 27: see 27.6.1.1 [lib.istream](#), paragraph 2. We also already have an answer for bitset input, since 23.3.5.3 [lib.bitset.operators](#), paragraph 4, says that bitset's `operator>>` is a formatted input function, and includes a reference to the appropriate definition in 27.6.1.2 [lib.istream.formatted](#).

It is only string input, in 21.3.7.9 [lib.string.io](#), that is potentially ambiguous. The LWG believes it was clearly intended that string's input functions obtain characters from the streambuf, rather than by using something like `basic_istream::get()`. Howard will provide wording.]

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:** [NAD](#) **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>
{ ... };
```

]

Rationale:

There is no ambiguity; the standard is clear as written. Library implementors are not permitted to add template parameters to standard library classes. This does not fall under the "as if" rule, so it would be permitted only if the standard gave explicit license for implementors to do this. This would require a change in the standard.

The LWG decided against making this change, because it would break user code involving template template parameters or specializations of standard library class templates.

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:** [Dup](#) **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()`

Rationale:

Subsumed by issue [264](#).

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

Section: 23.1.2 [lib.associative.reqmts](#) **Status:** [Ready](#) **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. If users need to modify elements, it is possible to use mutable members or `const_cast`.

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

The types `iterator` and `const_iterator` are permitted to be different types to allow for potential future work in which some member functions might be overloaded between the two types. No such member functions exist now, and the LWG believes that user functionality will not be impaired by permitting the two types to be the same. A function that operates on both iterator types can be defined for `const_iterator` alone, and can rely on the automatic conversion from `iterator` to `const_iterator`.

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

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));
// ERROR
}

```

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 `binder2nd` 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.binder.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.binder.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_iterator`s.

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.

[Toronto: most people saw no compelling reason to make this change. There was some argument that the standard already permits this behavior, on the grounds that it is illegal to have two different `istreambuf_iterator`s into the same stream. A possible counterexample:

```
istreambuf_iterator i(cin);
assert(i == i);
```

The standard currently requires that the assertion succeeds. (Assuming that we haven't reached eof on standard input.)]

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.]

[Toronto: This resolution may not be adequate for hex and octal output of signed int and signed short: PJP believes they should be converted to unsigned int and unsigned short first.]

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

Section: 27.6.1.2.2 [lib.istream.formatted.arithmetic](#) **Status:** [Ready](#) **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;
iosstate 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;
iosstate err = 0;
long lval;
use_facet< numget >(loc).get(*this, 0, *this, err, lval);
if (err == 0
    && (lval < numeric_limits<short>::min() ||
numeric_limits<short>::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;
iosstate err = 0;
long lval;
use_facet< numget >(loc).get(*this, 0, *this, err, lval);
if (err == 0
    && (lval < numeric_limits<int>::min() ||
numeric_limits<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:** [Open](#) **Submitter:** Judy Ward **Date:** 15 Dec 1998

Section 17.4.3.1 says:

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

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

```
#include <locale>
```

```
#include <wchar.h>

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

Lib-7047 [Matt Austern] comments:

The status quo is unclear, and probably contradictory. This issue applies both to 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.]

[Toronto: The LWG is concerned about the scope of this proposed resolution: manually instantiating standard library templates is a common method for reducing compilation times. One possible alternative is a core change: allow (and ignore) manual instantiation requests when there is an explicit specialization. Another possible alternative is requiring that library implementors provide a list of specializations and explicit instantiations as part of their documentation. Judy has volunteered to provide wording for the latter alternative.]

123. Should valarray helper arrays fill functions be const?

Section: 26.3.5.4 [lib.slice.arr.fill](#), 26.3.7.4 [lib.gslicing.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.]

136. `seekp`, `seekg` setting wrong streams?

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

153. Typo in `narrow()` semantics

Section:: 22.2.1.3.2 [lib.facet ctype.char.members](#) Status: [Ready](#) 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 /*dfault*/) const;
const char* narrow(const char* low, const char* high,
                  char /*dfault*/, char* to) const;
Returns: do_narrow(low, high, to).
```

to:

```
char          narrow(char c, char dfault) const;
const char* narrow(const char* low, const char* high,
                  char dfault, char* to) const;
Returns: do_narrow(c, dfault) or
        do_narrow(low, high, dfault, 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?]

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

Section:: 27.6.2.1 [lib.ostream](#) **Status:** [Ready](#) **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 `xspn()`) is never called by any of the standard output functions. Is this really intended? At minimum it would be convenient to call `xspn()` 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 `xspn()` will never be called by any `ostream` member and that this is intended.
2. relax the restriction and allow calling `overflow()` and `xspn()`. 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()`, `xspn()`, 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_type::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((const char*)s)` if the second argument is of type `const charT*`
- `char_traits<charT>::length(s)` if the second argument is of type `const char*`, `const signed char*`, or `const unsigned char*` and `charT` is not `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.]

[Toronto: The original proposed resolution involved `char_traits<signed char>` and `char_traits<unsigned char>`. There was strong opposition to requiring that library implementors provide those specializations of `char_traits`.

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

Section:: 27.8.1.4 [lib.filebuf.virtuals](#) **Status:** [Ready](#) **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 sequence 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() returns false.

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

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

179. Comparison of const_iterators to iterators doesn't work

Section: 23.1 [lib.container.requirements](#) **Status:** [Ready](#) **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;
}
```

[pre-Toronto: John Potter made the following comment:]

In case nobody has noticed, accepting it will break reverse_iterator.

The fix is to make the comparison operators templated on two types.

```
template <class Iterator1, class Iterator2>
bool operator== (reverse_iterator<Iterator1> const& x,
reverse_iterator<Iterator2> const& y);
```

Obviously: return x.base() == y.base();

Currently, no reverse_iterator to const_reverse_iterator compares are valid.

BTW, I think the issue is in support of bad code. Compares should be between two iterators of the same type. All std::algorithms require the begin and end iterators to be of the same type.

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.]

[Toronto: The LWG believes it is clear that the above wording applies only to the nested types X::iterator and X::const_iterator. There is no requirement that X::reverse_iterator and X::const_reverse_iterator can be mixed. If mixing them is considered important, that's a separate issue.]

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.]

[Toronto: This is tangentially related to issue [229](#), but only tangentially: the intent of this issue is to address use of the name `size_t` in contexts outside of namespace `std`, such as in the description of `::operator new`. The proposed changes should be reviewed to make sure they are correct.]

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

Section: 27.6.3 [lib.std.manip](#) **Status:** [Ready](#) **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, or if `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:** [Ready](#) **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 bool min() throw() { return false; }
        static bool 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 bool epsilon() throw() { return 0; }
        static bool round_error() throw() { return 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 bool infinity() throw() { return 0; }
        static bool quiet_NaN() throw() { return 0; }
        static bool signaling_NaN() throw() { return 0; }
        static bool denorm_min() throw() { return 0; }

        static const bool is_iec559 = false;
        static const bool is_bounded = true;
        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:** [Ready](#) **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:** [Ready](#) **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.]

187. iter_swap underspecified

Section: 25.2.2 [lib.alg.swap](#) **Status:** [Open](#) **Submitter:** Andrew Koenig **Date:** 14 Aug 99

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

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

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

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

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

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

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

A LWG member, Dave Abrahams, comments:

Not an objection necessarily, but I want to point out the cost of that requirement:

```
iter_swap(list<T>::iterator, list<T>::iterator)
```

can currently be specialized to be more efficient than `iter_swap(T*,T*)` for many T (by using splicing).
Your proposal would make that optimization illegal.

[Kona: The LWG notes the original need for `iter_swap` was proxy iterators which are no longer permitted.]

Proposed resolution:

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

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

to

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

[post-Toronto: The LWG is concerned about possible overspecification: there may be cases, such as Dave Abrahams's example above, and such as `vector<bool>`'s iterators, where it makes more sense for `iter_swap` to do something other than swap. If performance is a concern, it may be better to have explicit complexity requirements than to say how `iter_swap` should be implemented.]

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).]

[Toronto: The LWG agrees with the general intent of the proposed resolution, but had some quibbles about the wording. Andy Sawyer has volunteered to provide revised wording.]

198. Validity of pointers and references unspecified after iterator destruction

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

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

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

```
#include <iostream>
#include <vector>
#include <iterator>

int main()
{
    typedef std::vector<int> vec_t;
    vec_t v;
    v.push_back( 1 );

    // Is a pointer or reference obtained from an iterator still
    // valid after destruction of the iterator?
    int * p = &*v.begin();
    std::cout << *p << '\n'; // OK?

    // Is a pointer or reference obtained from an iterator still
    // valid after the value of the iterator changes?
    vec_t::iterator iter( v.begin() );
    p = &*iter++;
    std::cout << *p << '\n'; // OK?
```

```
    return 0;
}
```

The standard doesn't appear to directly address these questions. The standard needs to be clarified. At least two real-world cases have been reported where library implementors wasted considerable effort because of the lack of clarity in the standard. The question is important because requiring pointers and references to remain valid has the effect for practical purposes of prohibiting iterators from pointing to cached rather than actual elements of containers.

The standard itself assumes that pointers and references obtained from an iterator are still valid after iterator destruction or change. The definition of `reverse_iterator::operator*()`, 24.4.1.3.3 [lib.reverse.iter.op.star](#), which returns a reference, defines effects:

```
Iterator tmp = current;
return *--tmp;
```

The definition of `reverse_iterator::operator->()`, 24.4.1.3.4 [lib.reverse.iter.opref](#), which returns a pointer, defines effects:

```
return &(operator*());
```

Because the standard itself assumes pointers and references remain valid after iterator destruction or change, the standard should say so explicitly. This will also reduce the chance of user code breaking unexpectedly when porting to a different standard library implementation.

Proposed Resolution:

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

Changing the value of or destroying a forward iterator does not invalidate pointers and references previously obtained from that iterator.

footnote: This may have the effect for practical purposes of prohibiting forward iterators from pointing to cached rather than actual elements of containers.

[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.]

[Pre-Toronto: Steve Cleary pointed out the no-invalidation assumption by `reverse_iterator`. The issue and proposed resolution was reformulated yet again to reflect this reality.]

[post-Toronto: There is more support in the LWG for this proposed resolution than for other alternatives, but there is still some uncertainty. Perhaps one problem is that the standard mixes traversal style with access policy.]

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:** [Open](#) **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 `<float>` supply characteristics of implementation-dependent fundamental types (3.9.1).

to:

The headers `<limits>`, `<climits>`, and `<float>` 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.

[post-Toronto: The opinion of the LWG is that the wording in the standard, as well as the wording of the proposed resolution, is flawed. The term "arithmetic types" is well defined in C and C++, and it is not clear that the term is being used correctly. It is also not clear that the term "implementation dependent" has any useful meaning in this context. The biggest problem is that `numeric_limits` seems to be intended both for built-in types and for user-defined types, and the standard doesn't make it clear how `numeric_limits` applies to each of those cases. A wholesale review of `numeric_limits` is needed.]

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 \equiv , 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 $\text{pred}(e, *i) \neq \text{false}$.

If we adopt (2), we also need to decide whether $\text{pred}(*i, *(i - 1))$ is really what we meant, or whether $\text{pred}(*(i - 1), i)$ 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](#).]

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

Section: 23.3.3 23.3.4 [lib.set](#) **Status:** [Ready](#) **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_range()` 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`.]

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

Section: 22.2.2.1.2 [lib.facet.num.get.virtuals](#) **Status:** [Ready](#) **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 "0123456789abcdefghijklmnopqrstuvwxyz+-". 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+-";
```

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_iterators . User1 looks at his standard library implementation and says, "I know how to implement a more efficient unique_copy for my_iterators", 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](#).]

[post-Toronto: The LWG is not sure if this is a defect in the standard. Most LWG members believe that an implementation of `std::unique` like the one quoted in this issue is already illegal, since, under certain circumstances, its semantics are not those specified in the standard. The standard's description of `unique` does not say that overloading `adjacent_find` should have any effect.]

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. Core issue 229 has been opened to discuss the core aspects of this problem.

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.]

Dave Abrahams and Peter Dimov <pdimov@mmltd.net> have proposed an alternative resolution that involves core changes:

7.3.3/9:

- change the note to refer to partial specializations in general:
"Note: template partial specializations are found by looking up the primary template and then considering all partial specializations of that template. If a using-declaration names a template, partial specializations introduced after the using-declaration are effectively visible because the primary template is visible (14.5.4)."

14/2:

- remove the second sentence
- change the note to read:
"Note: if the declarator-id is a template-id, the declaration declares a template partial specialization (14.5.4)."

14/4:

- change "class template partial specialization" to "template partial specialization"

14.5.4:

- change section name to "Template partial specializations"

14.5.4/1:

- remove all occurrences of the word "class".

14.5.4/4:

- optionally provide an example for a function template partial specialization:
-
- `template<class T1, class T2, int I> T1 f(T2 (&t2) [I]);`
- `template<class T, int I> T f<T, T*, I>(T* (&t) [I]);`
- `template<class T1, class T2, int I> T1* f<T1*, T2, I>(T2 (&) [I]);`
- `template<class T> int f<int, T*, 5>(T* (&t) [5]);`
- `template<class T1, class T2, int I> T1 f<T1, T2*, I>(T2* (&a) [I]);`

14.5.4/5:

- remove the word "class" in the second sentence

14.5.4/6:

- not sure about that one

14.5.4/7:

- remove the word "class" in the third sentence

14.5.4/9:

- remove the word "class" in the first sentence

14.5.4/11 (new paragraph):

A function template partial specialization specializes a primary template if and only if, after substituting the template arguments provided in the specialization template argument list into the primary template declaration, the resulting function signature matches that of the specialization.

[Note: each function template partial specialization specializes at most one primary template.]

14.5.4/12 (new paragraph):

[Example:

```
template<class T> void f(T x);           // primary template #1
template<class U> void f(U* y);         // primary template #2

template<class V> void f<V*>(V* z);     // specialization of #1, T = V*
template<class W> void f<W*>(W** w);    // specialization of #2, U = W*
```

-- end example.]

14.5.4.1/1:

- remove the first occurrence of "class" in the first sentence
- change the second "class" to "template" in the first sentence

- remove the word "class" in the second sentence
- remove the word "class" in "the use of the class template is ambiguous"

14.5.4.2:

- change section name to "Partial ordering of template specializations"

14.5.4.2/1: (change to):

For two template partial specializations (that specialize the same primary template,) the first is at least as specialized as the second if, given the following rewrite to two function templates, the first function template is at least as specialized as the second according to the ordering rules for function templates (14.5.5.2):

- synthesize a unique class template with the same parameter list as the primary template;
- the first function template has the same template parameters as the first partial specialization and has a single function parameter whose type is a class template specialization of the synthesized class template with the template arguments of the first partial specialization;
- the second function template has the same template parameters as the second partial specialization and has a single function parameter whose type is a class template specialization of the synthesized class template with the template arguments of the second partial specialization.

14.5.4.2/2 (change example to):

```
template<int I, int J, class T> class X { };
template<int I, int J>          class X<I, J, int> { };          //
#1
template<int I>                class X<I, I, int> { };          //
#2

template<int I, int J, class T> class __unique;

template<int I, int J>          void __f(__unique<I, J, int>);    //
#A
template<int I>                void __f(__unique<I, I, int>);    //
#B
```

14.5.4.2/3 (new paragraph):

[Example:

```
template<class T, class U, class V> U f          (V);
template<class U, class V>8         U f<int, U, V> (V);          // #1
template<class T>                  T f<int, T, T> (T);          // #2

template<class T, class U, class V> class __unique;

template<class U, class V>          void __f(__unique<int, U, V>); // #A
template<class T>                  void __f(__unique<int, T, T>);  // #B
```

-- end example.]

[post-Toronto: core is reluctant to add partial specialization of function templates. It is viewed as a large change, and the proposal presented above leaves some issues unopened. The LWG believes that there is a serious problem, however: there is no good way for users to force the library to use user specializations of generic standard library functions. Koenig lookup isn't adequate, since names within the library must be qualified with `std` (see issue 225), specialization doesn't work (we don't have partial specialization of function templates), and users aren't permitted to add overloads within namespace `std`. Possible solutions discussed by the LWG include:

- *Allow users to add overloads within namespace `std`.*
- *Allow partial specialization of function templates.*
- *Define a special set of names, such as `swap`, which library components must refer to without qualification; all names not on this list must be qualified by `std`: `::`. (There are many possible variations on this issue. For example, we might say that a function `f` should be unqualified within `g` if `g`'s description in the standard says that it calls `f`.)*

/

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

Section: 22.2 [lib.locale.categories](#) **Status:** [Review](#) **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.moneyput.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 22.2.1.6 ([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.moneyput.byname`) to become:

```

namespace std {
    template <class charT, bool Intl = false>
    class moneyput_byname : public moneyput<charT, Intl> {
    public:
        typedef money_base::pattern pattern;
        typedef basic_string<charT> string_type;
        explicit moneyput_byname(const char*, size_t refs = 0);
    protected:
        ~moneyput_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<T>'.)

[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:** [Open](#) **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.

Issue [182](#), which addresses qualification of `size_t`, is a special case of this.

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](#).]

[post-Toronto: Howard is undecided about whether it is appropriate for all standard library function names referred to in other standard library functions to be explicitly qualified by `std::`: it is common advice that users should define global functions that operate on their class in the same namespace as the class, and this requires argument-dependent lookup if those functions are intended to be called by library code. Several LWG members are concerned that `valarray` appears to require argument-dependent lookup, but that the wording may not be clear enough to fall under "unless explicitly described otherwise".]

230. Assignable specified without also specifying CopyConstructible

Section: 17 [lib.library](#) **Status:** [Review](#) **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.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:

[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.]

[Toronto: The original proposed resolution also included changes to input iterator, fill, and replace. The LWG believes that those changes are not necessary. The LWG considered some blanket statement, where an Assignable type was also required to be Copy Constructible, but decided against this because fill and replace really don't require the Copy Constructible property.]

231. Precision in iostream?

Section: 22.2.2.2.2 [lib.facet.num.put.virtuals](#) **Status:** [Open](#) **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:

[Toronto: the committee believes this is a genuine issue. In addition to the issue of Fixed being a value rather than a mask, the standard is not clear what the effects setting precision to 0 ought to be.]

232. "depends" poorly defined in 17.4.3.1

Section: 17.4.3.1 [lib.reserved.names](#) **Status:** [Open](#) **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

[Toronto: this may be related to issue [120](#).]

233. Insertion hints in associative containers

Section: 23.1.2 [lib.associative.reqmts](#) **Status:** [Open](#) **Submitter:** Andrew Koenig **Date:** 30 Apr 2000

If `mm` is a multimap and `p` is an iterator into the multimap, then `mm.insert(p, x)` inserts `x` into `mm` with `p` as a hint as to where it should go. Table 69 claims that the execution time is amortized constant if the insert winds up taking place adjacent to `p`, but does not say when, if ever, this is guaranteed to happen. All it says is that `p` is a hint as to where to insert.

The question is whether there is any guarantee about the relationship between `p` and the insertion point, and, if so, what it is.

I believe the present state is that there is no guarantee: The user can supply `p`, and the implementation is allowed to disregard it entirely.

Proposed Resolution:

OPTION 1:

General Idea: Point out that in `insert(p,t)`, the iterator `p` will (if possible) be used to insert `t` just before `p` or just after `p`. If this is not possible, the hint is ignored.

assertion/note/pre/postcondition in table 69

Change:

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

To:

if t is inserted, p is used as follows: insert t right before p if possible; otherwise, insert t right after p if possible; otherwise, p is ignored.

complexity:

Change:

right after p

To:

right before or right after p.

Thus making:

assertion/note/pre/postcondition:

inserts t if and only if there is no element with key equivalent to the key of t in containers with unique keys; always inserts t in containers with equivalent keys. always returns the iterator pointing to the element with key equivalent to the key of t. if t is inserted, p is used as follows: insert t right before p if possible; otherwise, insert t right after p if possible; otherwise, p is ignored.

complexity:

logarithmic in general, but amortized constant if t is inserted right before or right after p.

OPTION 2

General Idea (Andrew Koenig): t is inserted at the point closest to (the point immediately ahead of) p. That would give the user a way of controlling the order in which elements appear that have equal keys. Doing so would be particularly easy in two cases that I suspect are common:

```
mm.insert(mm.begin(), t); // inserts as first element of set of equal keys
mm.insert(mm.end(), t);   // inserts as last element of set of equal keys
```

These examples would allow t to be inserted at the beginning and end, respectively, of the set of elements with the same key as t.

assertion/note/pre/postcondition in table 69

Change:

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

To:

if t is inserted, p is used as follows: insert t right before p if possible; otherwise, if p is equal to a.end(), or if the key value of t is greater than the key value of *p, t is inserted just before a.lowerbound(the key value of t); otherwise, t is inserted right before a.upperbound(the key value of t).

complexity:

Change:

right after p

To:

right before p

Thus making:

assertion/note/pre/postcondition:

inserts t if and only if there is no element with key equivalent to the key of t in containers with unique keys; always inserts t in containers with equivalent keys. always returns the iterator pointing to the element with key equivalent to the key of t. if t is inserted, p is used as follows: insert t right before p if possible; otherwise, if p is equal to a.end(), or if the key value of t is greater than the key value of *p, t is inserted just before a.lowerbound(the key value of t); otherwise, t is inserted right before a.upperbound(the key value of t).

NON-NORMATIVE FOOTNOTE: | This gives the user a way of controlling the order | in which elements appear that have equal keys. Doing this is | particularly easy in two common cases:

```
| mm.insert(mm.begin(), t); // inserts as first element of set of
equal keys
| mm.insert(mm.end(), t);   // inserts as last element of set of equal
keys
```

END-FOOTNOTE

complexity:

logarithmic in general, but amortized constant if t is inserted right before p.

[Toronto: there was general agreement that this is a real defect: when inserting an element x into a multiset that already contains several copies of x, there is no way to know whether the hint will be used. There was some support for an alternative resolution: we check on both sides of the hint (both before and after, in that order). If either is the correct location, the hint is used; otherwise it is not. This is different from the original proposed resolution, because in the proposed resolution the hint will be used even if it is very far from the insertion point. JC van Winkel supplied precise wording for both options.]

234. Typos in allocator definition

Section: 20.4.1.1 [lib.allocator.members](#) **Status:** [Ready](#) **Submitter:** Dietmar Kühl **Date:** 24 Apr 2000

In paragraphs 12 and 13 the effects of `construct()` and `destruct()` are described as returns but the functions actually return `void`.

Proposed Resolution:

Substitute "Returns" by "Effect".

235. No specification of default ctor for `reverse_iterator`

Section: 24.4.1.1 [lib.reverse.iterator](#) **Status:** [Open](#) **Submitter:** Dietmar Kühl **Date:** 24 Apr 2000

The declaration of `reverse_iterator` lists a default constructor. However, no specification is given what this constructor should do.

Proposed Resolution:

[Toronto: there should be a default constructor, and it should default-initialize the appropriate member variable. Dietmar will provide wording.]

237. Undefined expression in complexity specification

Section: 23.2.2.1 [lib.list.cons](#) **Status:** [Ready](#) **Submitter:** Dietmar Kühl **Date:** 24 Apr 2000

The complexity specification in paragraph 6 says that the complexity is linear in `first - last`. Even if `operator-()` is defined on iterators this term is in general undefined because it would have to be `last - first`.

Proposed Resolution:

Change paragraph 6 from

Linear in *first - last*.

to become

Linear in *distance(first, last)*.

238. Contradictory results of stringbuf initialization.

Section: 27.7.1.1 [lib.stringbuf.cons](#) **Status:** [Review](#) **Submitter:** Dietmar Kühl **Date:** 11 May 2000

In 27.7.1.1 paragraph 4 the results of calling the constructor of 'basic_stringbuf' are said to be `str() == str`. This is fine that far but consider this code:

```
std::basic_stringbuf<char> sbuf("hello, world", std::ios_base::openmode(0));
std::cout << " " << sbuf.str() << "\n";
```

Paragraph 3 of 27.7.1.1 basically says that in this case neither the output sequence nor the input sequence is initialized and paragraph 2 of 27.7.1.2 basically says that `str()` either returns the input or the output sequence. None of them is initialized, ie. both are empty, in which case the return from `str()` is defined to be `basic_string<CT>()`.

However, probably only test cases in some testsuites will detect this "problem"...

Proposed Resolution:

Remove 27.7.1.1 paragraph 4.

Rationale:

We could fix 27.7.1.1 paragraph 4, but there would be no point. If we fixed it, it would say just the same thing as text that's already in the standard.

239. Complexity of `unique()` and/or `unique_copy` incorrect

Section: 25.2.8 [lib.alg.unique](#) **Status:** [Open](#) **Submitter:** Angelika Langer **Date:** May 15 2000

The complexity of `unique` and `unique_copy` are inconsistent with each other and inconsistent with the implementations. The standard specifies:

for `unique()`:

-3- Complexity: If the range (last - first) is not empty, exactly (last - first) - 1 applications of the corresponding predicate, otherwise no applications of the predicate.

for `unique_copy()`:

-7- Complexity: Exactly last - first applications of the corresponding predicate.

The implementations do it the other way round: `unique()` applies the predicate last-first times and `unique_copy()` applies it last-first-1 times.

As both algorithms use the predicate for pair-wise comparison of sequence elements I don't see a justification for `unique_copy()` applying the predicate last-first times, especially since it is not specified to which pair in the sequence the predicate is applied twice.

Proposed Resolution:

Change both complexity sections in 25.2.8 [lib.alg.unique](#) to:

Complexity: Exactly last - first - 1 applications of the corresponding predicate.

[Toronto: This is related to issue [202](#). We can't specify `unique`'s complexity until we decide what `unique` is supposed to do.]

240. Complexity of `adjacent_find()` is meaningless

Section: 25.1.5 [lib.alg.adjacent.find](#) **Status:** [Review](#) **Submitter:** Angelika Langer **Date:** May 15 2000

The complexity section of `adjacent_find` is defective.

```
template<class ForwardIterator, class BinaryPredicate>
    ForwardIterator adjacent_find(ForwardIterator first, ForwardIterator last,
                                BinaryPredicate pred);
```

-1- Returns: The first iterator `i` such that both `i` and `i + 1` are in the range `[first, last)` for which the following corresponding conditions hold: `*i == *(i + 1)`, `pred(*i, *(i + 1)) != false`. Returns `last` if no such iterator is found.

-2- Complexity: Exactly `find(first, last, value)` - first applications of the corresponding predicate.

In the Complexity section, it is not defined what "value" is supposed to mean. My best guess is that "value" means an object for which one of the conditions `pred(*i,value)` or `pred(value,*i)` is true, where `i` is the iterator defined in the Returns section. However, the value type of the input sequence need not be equality-comparable and for this reason the term `find(first, last, value) - first` is meaningless.

A term such as `find_if(first, last, bind2nd(pred,*i)) - first` or `find_if(first, last, bind1st(pred,*i)) - first` might come closer to the intended specification. Binders can only be applied to function objects that have the function call operator declared `const`, which is not required of predicates because they can have non-const data members. For this reason, a specification using a binder could only be an "as-if" specification.

Proposed Resolution:

Change the complexity section in 25.1.5 [lib.alg.adjacent.find](#) to: "For a nonempty range, at most $(last - first) - 1$ comparisons."

241. Does `unique_copy()` require `CopyConstructible` and `Assignable`?

Section: 25.2.8 [lib.alg.unique](#) **Status:** [Review](#) **Submitter:** Angelika Langer **Date:** May 15 2000

Some popular implementations of `unique_copy()` create temporary copies of values in the input sequence, at least if the input iterator is a pointer. Such an implementation is built on the assumption that the value type is `CopyConstructible` and `Assignable`.

It is common practice in the standard that algorithms explicitly specify any additional requirements that they impose on any of the types used by the algorithm. An example of an algorithm that creates temporary copies and correctly specifies the additional requirements is `accumulate()` [[lib.accumulate](#)].

Since the specifications of `unique()` and `unique_copy()` do not require `CopyConstructible` and `Assignable` of the `InputIterator`'s value type the above mentioned implementations are not standard-compliant. I cannot judge whether this is a defect in the standard or a defect in the implementations.

Proposed Resolution:

In 25.2.8 change:

-4- Requires: The ranges `[first, last)` and `[result, result+(last-first))` shall not overlap.

to:

-4- Requires: The ranges `[first, last)` and `[result, result+(last-first))` shall not overlap. The expression `*result = *first` is valid

Rationale:

Creating temporary copies is unavoidable, since the arguments may be input iterators; this implies that the value type must be copy constructible. However, we don't need to say this explicitly; it's already implied by table 72 in 24.1.1. We don't precisely want to say that the input iterator's value type `T` must be assignable, because we never quite use that property. We assign through the output iterator. The output iterator might have a different value type, or no value type; it might not use `T`'s assignment operator. If it's an `ostream_iterator`, for example, then we'll use `T`'s `operator<<` but not its assignment operator.

242. Side effects of function objects

Section: 25.2.3 [lib.alg.transform](#) and 26.4 [lib.numeric.ops](#) **Status:** [Review](#) **Submitter:** Angelika Langer **Date:** May 15 2000

The algorithms `transform()`, `accumulate()`, `inner_product()`, `partial_sum()`, and `adjacent_difference()` require that the function object supplied to them shall not have any side effects.

The standard defines a side effect in [\[intro.execution\]](#) as:

-7- Accessing an object designated by a volatile lvalue (`basic.lval`), modifying an object, calling a library I/O function, or calling a function that does any of those operations are all side effects, which are changes in the state of the execution environment.

As a consequence, the function call operator of a function object supplied to any of the algorithms listed above cannot modify data members, cannot invoke any function that has a side effect, and cannot even create and modify temporary objects. It is difficult to imagine a function object that is still useful under these severe limitations. For instance, any non-trivial transformer supplied to `transform()` might involve creation and modification of temporaries, which is prohibited according to the current wording of the standard.

On the other hand, popular implementations of these algorithms exhibit uniform and predictable behavior when invoked with a side-effect-producing function objects. It looks like the strong requirement is not needed for efficient implementation of these algorithms.

The requirement of side-effect-free function objects could be replaced by a more relaxed basic requirement (which would hold for all function objects supplied to any algorithm in the standard library):

A function objects supplied to an algorithm shall not invalidate any iterator or sequence that is used by the algorithm. Invalidation of the sequence includes destruction of the sorting order if the algorithm relies on the sorting order (see section 25.3 - Sorting and related operations [[lib.alg.sorting](#)]).

I can't judge whether it is intended that the function objects supplied to `transform()`, `accumulate()`, `inner_product()`, `partial_sum()`, or `adjacent_difference()` shall not modify sequence elements through dereferenced iterators.

It is debatable whether this issue is a defect or a change request. Since the consequences for user-supplied function objects are drastic and limit the usefulness of the algorithms significantly I would consider it a defect.

Proposed Resolution:

[Toronto: Dave Abrahams supplied wording.]

Things to notice about these changes:

1. *The fully-closed ("`[]`") as opposed to half-closed ("`[]`") ranges are intentional. we want to prevent side-effects from invalidating the end iterators.*
2. *That has the unintentional side-effect of prohibiting modification of the end element as a side-effect. This could conceivably be significant in some cases.*
3. *The wording also prevents side-effects from modifying elements of the output sequence. I can't imagine why anyone would want to do this, but it is arguably a restriction that implementors don't need to place on users.*
4. *Lifting the restrictions imposed in #2 and #3 above is possible and simple, but would require more verbiage.*

Change 25.2.3/2 from:

-2- Requires: `op` and `binary_op` shall not have any side effects.

to:

-2- Requires: `op` and `binary_op` shall not invalidate iterators or subranges, or modify elements in the ranges `[first1, last1]`, `[first2, first2 + (last1 - first1)]`, and `[result, result + (last1 - first1)]`.

Change 26.4.1/2 from:

-2- Requires: `T` must meet the requirements of `CopyConstructible` (`lib.copyconstructible`) and `Assignable` (`lib.container.requirements`) types. `binary_op` shall not cause side effects.

to:

-2- Requires: `T` must meet the requirements of `CopyConstructible` (`lib.copyconstructible`) and `Assignable` (`lib.container.requirements`) types. In the range `[first, last]`, `binary_op` shall neither modify elements nor invalidate iterators or subranges.

Change 26.4.2/2 from:

-2- Requires: `T` must meet the requirements of `CopyConstructible` (`lib.copyconstructible`) and `Assignable` (`lib.container.requirements`) types. `binary_op1` and `binary_op2` shall not cause side effects.

to:

-2- Requires: `T` must meet the requirements of `CopyConstructible` (`lib.copyconstructible`) and `Assignable` (`lib.container.requirements`) types. In the ranges `[first, last]` and `[first2, first2 + (last - first)]`, `binary_op1` and `binary_op2` shall neither modify elements nor invalidate iterators or subranges.

Change 26.4.3/4 from:

-4- Requires: `binary_op` is expected not to have any side effects.

to:

-4- Requires: In the ranges `[first, last]` and `[result, result + (last - first)]`, `binary_op` shall neither modify elements nor invalidate iterators or subranges.

Change 26.4.4/2 from:

-2- Requires: `binary_op` shall not have any side effects.

to:

-2- Requires: In the ranges `[first, last]` and `[result, result + (last - first)]`, `binary_op` shall neither modify elements nor invalidate iterators or subranges.

243. `get` and `getline` when sentry reports failure

Section: 27.6.1.3 [lib.istream.unformatted](#) **Status:** [Ready](#) **Submitter:** Martin Sebor **Date:** May 15 2000

`basic_istream::get()`, and `basic_istream::getline()`, are unclear with respect to the behavior and side-effects of the named functions in case of an error.

27.6.1.3, p1 states that "... If the sentry object returns true, when converted to a value of type bool, the function endeavors to obtain the requested input..." It is not clear from this (or the rest of the paragraph) what precisely the behavior should be when the sentry ctor exits by throwing an exception or when the sentry object returns false. In particular, what is the number of characters extracted that `gcount()` returns supposed to be?

27.6.1.3 p8 and p19 say about the effects of `get()` and `getline()`: "... In any case, it then stores a null character (using `charT()`) into the next successive location of the array." Is not clear whether this sentence applies if either of the conditions above holds (i.e., when sentry fails).

Proposed Resolution:

Add to 27.6.1.3, p1 after the sentence

"... If the sentry object returns true, when converted to a value of type bool, the function endeavors to obtain the requested input."

the following

"Otherwise, if the sentry constructor exits by throwing an exception or if the sentry object returns false, when converted to a value of type bool, the function returns without attempting to obtain any input. In either case the number of extracted characters is set to 0; unformatted input functions taking a character array of non-zero size as an argument shall also store a null character (using `charT()`) in the first location of the array."

Rationale:

Although the general philosophy of the input functions is that the argument should not be modified upon failure, `getline` historically added a terminating null unconditionally. Most implementations still do that. Earlier versions of the draft standard had language that made this an unambiguous requirement; those words were moved to a place where their context made them less clear. See Jerry Schwarz's message [c++std-lib-7618](#).

247. `vector`, `deque::insert` complexity

Section: 23.3.4.4 [lib.vector.modifiers](#) **Status:** [Open](#) **Submitter:** Lisa Lippincott **Date:** 06 June 2000

Paragraph 2 of 23.3.4.3 [[lib.vector.modifiers](#)] describes the complexity of `vector::insert`:

Complexity: If first and last are forward iterators, bidirectional iterators, or random access iterators, the complexity is linear in the number of elements in the range [first, last) plus the distance to the end of the vector. If they are input iterators, the complexity is proportional to the number of elements in the range [first, last) times the distance to the end of the vector.

First, this fails to address the non-iterator forms of `insert`.

Second, the complexity for input iterators misses an edge case -- it requires that an arbitrary number of elements can be added at the end of a `vector` in constant time.

At the risk of strengthening the requirement, I suggest simply

Complexity: The complexity is linear in the number of elements inserted plus the distance to the end of the vector.

For input iterators, one may achieve this complexity by first inserting at the end of the `vector`, and then using `rotate`.

I looked to see if `deque` had a similar problem, and was surprised to find that `deque` places no requirement on the complexity of inserting multiple elements ([23.2.1.3](#) [lib.deque.modifiers], paragraph 3):

Complexity: In the worst case, inserting a single element into a deque takes time linear in the minimum of the distance from the insertion point to the beginning of the deque and the distance from the insertion point to the end of the deque. Inserting a single element either at the beginning or end of a deque always takes constant time and causes a single call to the copy constructor of T.

I suggest:

Complexity: The complexity is linear in the number of elements inserted plus the shorter of the distances to the beginning and end of the deque. Inserting a single element at either the beginning or the end of a deque causes a single call to the copy constructor of T.

Proposed Resolution:

*[Toronto: It's agreed that there is a defect in complexity of multi-element insert for vector and deque. For vector, the complexity should probably be something along the lines of $c_1 * N + c_2 * \text{distance}(i, \text{end}())$. However, there is some concern about whether it is reasonable to amortize away the copies that we get from a reallocation whenever we exceed the vector's capacity. For deque, the situation is somewhat less clear. Deque is notoriously complicated, and we may not want to impose complexity requirements that would imply any implementation technique more complicated than a while loop whose body is a single-element insert.]*

248. time_get fails to set eofbit

Section: 22.2.5 [lib.category.time](#) Status: [Ready](#) Submitter: Martin Sebor Date: 22 June 2000

There is no requirement that any of `time_get` member functions set `ios::eofbit` when they reach the end iterator while parsing their input. Since members of both the `num_get` and `money_get` facets are required to do so (22.2.2.1.2, and 22.2.6.1.2, respectively), `time_get` members should follow the same requirement for consistency.

Proposed Resolution:

Add paragraph 2 to section 22.2.5.1 with the following text:

If the end iterator is reached during parsing by any of the `get()` member functions, the member sets `ios_base::eofbit` in `err`.

[Toronto: Martin submitted two proposed resolutions. The LWG chose this one because it was more consistent with the way eof is described for other input facets.]

250. splicing invalidates iterators

Section: 23.2.2.4 [lib.list.ops](#) Status: [Open](#) Submitter: Brian Parker <brianp@research.canon.com.au> Date: 14 Jul 2000

Section 23.2.2.4 [lib.list.ops] states that

```
void splice(iterator position, list<T, Allocator>& x);
```

invalidates all iterators and references to list *x*.

This is unnecessary and defeats an important feature of splice. In fact, the SGI STL guarantees that iterators to *x* remain valid after splice.

Proposed Resolution:

I think that this clause (and the other splice clauses) should be reworded to- "all iterators and references remain valid, including iterators that point to elements of *x*."

[Toronto: the LWG was generally in favor of the idea behind this change. Some members, however, are concerned about the phrase "remain valid". They aren't sure that validity is sufficiently well defined, or that it means exactly what we mean to say here. Howard will provide more precise wording.]

251. basic_stringbuf missing allocator_type

Section: 27.7.1 [lib.stringbuf](#) **Status:** [Ready](#) **Submitter:** Martin Sebor **Date:** 28 Jul 2000

The synopsis for the template class `basic_stringbuf` doesn't list a typedef for the template parameter `Allocator`. This makes it impossible to determine the type of the allocator at compile time. It's also inconsistent with all other template classes in the library that do provide a typedef for the `Allocator` parameter.

Proposed Resolution:

Add to the synopses of the class templates `basic_stringbuf` (27.7.1), `basic_istream` (27.7.2), `basic_ostringstream` (27.7.3), and `basic_stringstream` (27.7.4) the typedef:

```
typedef Allocator allocator_type;
```

252. missing casts/C-style casts used in iostreams

Section: 27.7 [lib.string.streams](#) **Status:** [Ready](#) **Submitter:** Martin Sebor **Date:** 28 Jul 2000

27.7.2.2, p1 uses a C-style cast rather than the more appropriate `const_cast<>` in the Returns clause for `basic_istream<>::rdbuf()`. The same C-style cast is being used in 27.7.3.2, p1, D.7.2.2, p1, and D.7.3.2, p1, and perhaps elsewhere. 27.7.6, p1 and D.7.2.2, p1 are missing the cast altogether.

C-style casts have not been deprecated, so the first part of this issue is stylistic rather than a matter of correctness.

Proposed Resolution:

In 27.7.2.2, p1 replace

```
-1- Returns: (basic_stringbuf<charT, traits, Allocator>*)&sb.
```

with

```
-1- Returns: const_cast<basic_stringbuf<charT,traits,Allocator>*>(&sb).
```

In 27.7.3.2, p1 replace

```
-1- Returns: (basic_stringbuf<charT,traits,Allocator>*)&sb.
```

with

```
-1- Returns: const_cast<basic_stringbuf<charT,traits,Allocator>*>(&sb).
```

In 27.7.6, p1, replace

```
-1- Returns: &sb
```

with

```
-1- Returns: const_cast<basic_stringbuf<charT,traits,Allocator>*>(&sb).
```

In D.7.2.2, p1 replace

```
-2- Returns: &sb.
```

with

```
-2- Returns: const_cast<strstreambuf*>(&sb).
```

253. valarray helper functions are almost entirely useless

Section: 26.3.2.1 [lib.valarray.cons](#) and 26.3.2.2 [lib.valarray.assign](#) **Status:** [Open](#) **Submitter:** Robert Klarer **Date:** 31 Jul 2000

This discussion is adapted from message c++std-lib-7056 posted November 11, 1999. I don't think that anyone can reasonably claim that the problem described below is NAD.

These valarray constructors can never be called:

```
template <class T>
    valarray<T>::valarray(const slice_array<T> &);
template <class T>
    valarray<T>::valarray(const gslice_array<T> &);
template <class T>
    valarray<T>::valarray(const mask_array<T> &);
template <class T>
    valarray<T>::valarray(const indirect_array<T> &);
```

Similarly, these valarray assignment operators cannot be called:

```
template <class T>
    valarray<T> valarray<T>::operator=(const slice_array<T> &);
template <class T>
```

```

valarray<T> valarray<T>::operator=(const gslice_array<T> &);
template <class T>
valarray<T> valarray<T>::operator=(const mask_array<T> &);
template <class T>
valarray<T> valarray<T>::operator=(const indirect_array<T> &);

```

Please consider the following example:

```

#include <valarray>
using namespace std;

int main()
{
    valarray<double> va1(12);
    valarray<double> va2(va1[slice(1,4,3)]); // line 1
}

```

Since the `valarray` `va1` is non-const, the result of the sub-expression `va1[slice(1,4,3)]` at line 1 is an rvalue of type `const std::slice_array<double>`. This `slice_array` rvalue is then used to construct `va2`. The constructor that is used to construct `va2` is declared like this:

```

template <class T>
valarray<T>::valarray(const slice_array<T> &);

```

Notice the constructor's const reference parameter. When the constructor is called, a `slice_array` must be bound to this reference. The rules for binding an rvalue to a const reference are in 8.5.3, paragraph 5 (see also 13.3.3.1.4). Specifically, paragraph 5 indicates that a second `slice_array` rvalue is constructed (in this case copy-constructed) from the first one; it is this second rvalue that is bound to the reference parameter. Paragraph 5 also requires that the constructor that is used for this purpose be callable, regardless of whether the second rvalue is elided. The copy-constructor in this case is not callable, however, because it is private. Therefore, the compiler should report an error.

Since `slice_arrays` are always rvalues, the `valarray` constructor that has a parameter of type `const slice_array<T> &` can never be called. The same reasoning applies to the three other constructors and the four assignment operators that are listed at the beginning of this post. Furthermore, since these functions cannot be called, the `valarray` helper classes are almost entirely useless.

Proposed Resolution:

Adopt section 2 of 00-0023/N1246. Sections 1 and 5 of that paper have already been classified as "Request for Extension". Sections 3 and 4 are reasonable generalizations of section 2, but they do not resolve an obvious inconsistency in the standard.

[Toronto: it is agreed that there is a defect. A full discussion, and an attempt at fixing the defect, should wait until we can hear from valarray experts.]

254. Exception types in clause 19 are constructed from `std::string`

Section: 19.1 [lib.std.exceptions](#) **Status:** [Open](#) **Submitter:** Dave Abrahams **Date:** 01 Aug 2000

Many of the standard exception types which implementations are required to throw are constructed with a `const std::string&` parameter. For example:


```

19.1.1.5 Class out_of_range [lib.out.of.range]
namespace std {
    class out_of_range : public logic_error {
    public:
        explicit out_of_range(const string& what_arg);
    };
}

```

- 1 The class `out_of_range` defines the type of objects thrown as exceptions to report an argument value not in its expected range.

```
out_of_range(const string& what_arg);
```

Effects:

Constructs an object of class `out_of_range`.

Postcondition:

```
strcmp(what(), what_arg.c_str()) == 0.
```

There are at least two problems with this:

1. A program which is low on memory may end up throwing `std::bad_alloc` instead of `out_of_range` because memory runs out while constructing the exception object.
2. An obvious implementation which stores a `std::string` data member may end up invoking `terminate()` during exception unwinding because the exception object allocates memory (or rather fails to) as it is being copied.

There may be no cure for (1) other than changing the interface to `out_of_range`, though one could reasonably argue that (1) is not a defect. Personally I don't care that much if out-of-memory is reported when I only have 20 bytes left, in the case when `out_of_range` would have been reported. People who use exception-specifications might care a lot, though.

There is a cure for (2), but it isn't completely obvious. I think a note for implementors should be made in the standard. Avoiding possible termination in this case shouldn't be left up to chance. The cure is to use a reference-counted "string" implementation in the exception object. I am not necessarily referring to a `std::string` here; any simple reference-counting scheme for a NTBS would do.

Further discussion, in email:

...I'm not so concerned about (1). After all, a library implementation can add `const char*` constructors as an extension, and users don't *need* to avail themselves of the standard exceptions, though this is a lame position to be forced into. FWIW, `std::exception` and `std::bad_alloc` don't require a temporary `basic_string`.

...I don't think the fixed-size buffer is a solution to the problem, strictly speaking, because you can't satisfy the postcondition

```
strcmp(what(), what_arg.c_str()) == 0
```

For all values of `what_arg` (i.e. very long values). That means that the only truly conforming solution requires a dynamic allocation.

Proposed Resolution:

[Toronto: some LWG members thought this was merely a QoI issue, but most believed that it was at least a borderline defect. There was more support for nonnormative advice to implementors than for a normative change.]

256. typo in 27.4.4.2, p17: `copy_event` does not exist

Section: 27.4.4.2 [lib.basic.ios.members](#) **Status:** [Ready](#) **Submitter:** Martin Sebor **Date:** 21 Aug 2000

27.4.4.2, p17 says

-17- Before copying any parts of rhs, calls each registered callback pair (fn,index) as (*fn)(erase_event,*this,index). After all parts but exceptions() have been replaced, calls each callback pair that was copied from rhs as (*fn)(copy_event,*this,index).

The name copy_event isn't defined anywhere. The intended name was copyfmt_event.

Proposed Resolution:

Replace copy_event with copyfmt_event in the named paragraph.

257. STL functional object and iterator inheritance.

Section: 20.3.1 [lib.base](#) and 24.3.2 [lib.iterator.basic](#) **Status:** [NAD](#) **Submitter:** Robert Dick
<dickrp@venus.ee.Princeton.EDU> **Date:** 17 Aug 2000

According to the November 1997 Draft Standard, the results of deleting an object of a derived class through a pointer to an object of its base class are undefined if the base class has a non-virtual destructor. Therefore, it is potentially dangerous to publicly inherit from such base classes.

Defect:

The STL design encourages users to publicly inherit from a number of classes which do nothing but specify interfaces, and which contain non-virtual destructors.

Attribution:

Wil Evers and William E. Kempf suggested this modification for functional objects.

Proposed Resolution:

Proposed correction:

When a base class in the standard library is useful only as an interface specifier, i.e., when an object of the class will never be directly instantiated, specify that the class contains a protected destructor. This will prevent deletion through a pointer to the base class without performance, or space penalties (on any implementation I'm aware of).

As an example, replace...

```
template <class Arg, class Result>
struct unary_function {
    typedef Arg      argument_type;
    typedef Result  result_type;
};
```

... with...

```
template <class Arg, class Result>
struct unary_function {
    typedef Arg      argument_type;
```

```

        typedef Result result_type;
protected:
        ~unary_function() {}
};

```

Affected definitions:

20.3.1 [lib.function.objects] -- unary_function, binary_function

24.3.2 [lib.iterator.basic] -- iterator

Rationale:

The standard is clear as written; this is a request for change, not a defect in the strict sense. The LWG had several different objections to the proposed change. One is that it would prevent users from creating objects of type `unary_function` and `binary_function`. Doing so can sometimes be legitimate, if users want to pass temporaries as traits or tag types in generic code.

258. Missing allocator requirement

Section: 20.1.5 [lib.allocator.requirements](#) **Status:** [Open](#) **Submitter:** Matt Austern **Date:** 22 Aug 2000

From lib-7752:

I've been assuming (and probably everyone else has been assuming) that allocator instances have a particular property, and I don't think that property can be deduced from anything in Table 32.

I think we have to assume that allocator type conversion is a homomorphism. That is, if x_1 and x_2 are of type X , where $X::\text{value_type}$ is T , and if type Y is $X::\text{template rebind}<U>::\text{other}$, then $Y(x_1) == Y(x_2)$ if and only if $x_1 == x_2$.

Further discussion: Howard Hinant writes, in lib-7757:

I think I can prove that this is not proveable by Table 32. And I agree it needs to be true except for the "and only if". If $x_1 != x_2$, I see no reason why it can't be true that $Y(x_1) == Y(x_2)$. Admittedly I can't think of a practical instance where this would happen, or be valuable. But I also don't see a need to add that extra restriction. I think we only need:

$$\text{if } (x_1 == x_2) \text{ then } Y(x_1) == Y(x_2)$$

If we decide that $==$ on allocators is transitive, then I think I can prove the above. But I don't think $==$ is necessarily transitive on allocators. That is:

Given $x_1 == x_2$ and $x_2 == x_3$, this does not mean $x_1 == x_3$.

Example:

```

x1 can deallocate pointers from: x1, x2, x3
x2 can deallocate pointers from: x1, x2, x4
x3 can deallocate pointers from: x1, x3
x4 can deallocate pointers from: x2, x4

```

$$x_1 == x_2, \text{ and } x_2 == x_4, \text{ but } x_1 != x_4$$

Proposed Resolution:

[Toronto: LWG members offered multiple opinions. One opinion is that it should not be required that $x1 == x2$ implies $Y(x1) == Y(x2)$, and that it should not even be required that $X(x1) == x1$. Another opinion is that the second line from the bottom in table 32 already implies the desired property. This issue should be considered in light of other issues related to allocator instances.]

259. `basic_string::operator[]` and const correctness

Section: 21.3.4 [lib.string.access](#) **Status:** [Review](#) **Submitter:** Chris Newton <chrisnewton@btinternet.com> **Date:**

Paraphrased from a message that Chris Newton posted to comp.std.c++:

The standard's description of `basic_string<>::operator[]` seems to violate const correctness.

The standard (21.3.4/1) says that "If `pos < size()`, returns `data()[pos]`." The types don't work. The return value of `data()` is `const charT*`, but `operator[]` has a non-const version whose return type is `reference`.

Proposed Resolution:

In section 21.3.4, paragraph 1, change "`data()[pos]`" to "`*(begin() + pos)`".

260. Inconsistent return type of `istream_iterator::operator++(int)`

Section: 24.5.1.2 [lib.istream.iterator.ops](#) **Status:** [Ready](#) **Submitter:** Martin Sebor **Date:** 27 Aug 2000

The synopsis of `istream_iterator::operator++(int)` in 24.5.1 shows it as returning the iterator by value. 24.5.1.2, p5 shows the same operator as returning the iterator by reference. That's incorrect given the Effects clause below (since a temporary is returned). The ``&'` is probably just a typo.

Proposed Resolution:

Change the declaration in 24.5.1.2, p5 from

```
istream_iterator<T, charT, traits, Distance>& operator++(int);
```

to

```
istream_iterator<T, charT, traits, Distance> operator++(int);
```

(that is, remove the ``&'`).

261. Missing description of `istream_iterator::operator!=`

Section: 24.5.1.2 [lib.istream.iterator.ops](#) **Status:** [Ready](#) **Submitter:** Martin Sebor **Date:** 27 Aug 2000

24.5.1, p3 lists the synopsis for

```
template <class T, class charT, class traits, class Distance>
    bool operator!=(const istream_iterator<T,charT,traits,Distance>& x,
                    const istream_iterator<T,charT,traits,Distance>& y);
```

but there is no description of what the operator does (i.e., no Effects or Returns clause) in 24.5.1.2.

Proposed Resolution:

Add paragraph 7 to the end of section 24.5.1.2 with the following text:

```
template <class T, class charT, class traits, class Distance>
    bool operator!=(const istream_iterator<T,charT,traits,Distance>& x,
                    const istream_iterator<T,charT,traits,Distance>& y);
```

-7- Returns: `!(x == y)`.

262. Bitmask operator ~ specified incorrectly

Section: 17.3.2.1.2 [lib.bitmask.types](#) **Status:** [Ready](#) **Submitter:** Beman Dawes **Date:** 03 Sep 2000

The ~ operation should be applied after the cast to `int_type`.

Proposed Resolution:

Change 17.3.2.1.2 [[lib.bitmask.types](#)] `operator~` from:

```
bitmask operator~ ( bitmask X )
{ return static_cast< bitmask>(static_cast<int_type>(~ X)); }
```

to:

```
bitmask operator~ ( bitmask X )
{ return static_cast< bitmask>(~static_cast<int_type>(X)); }
```

263. Severe restriction on `basic_string` reference counting

Section: 21.3 [lib.basic.string](#) **Status:** [Ready](#) **Submitter:** Kevlin Henney <kevin@curbralan.com> **Date:** 04 Sep 2000

The note in paragraph 6 suggests that the invalidation rules for references, pointers, and iterators in paragraph 5 permit a reference-counted implementation (actually, according to paragraph 6, they permit a "reference counted implementation", but this is a minor editorial fix).

However, the last sub-bullet is so worded as to make a reference-counted implementation unviable. In the following example none of the conditions for iterator invalidation are satisfied:

```
// first example: "*****" should be printed twice
```

```

string original = "some arbitrary text", copy = original;
const string & alias = original;

string::const_iterator i = alias.begin(), e = alias.end();
for(string::iterator j = original.begin(); j != original.end(); ++j)
    *j = '*';
while(i != e)
    cout << *i++;
cout << endl;
cout << original << endl;

```

Similarly, in the following example:

```

// second example: "some arbitrary text" should be printed out
string original = "some arbitrary text", copy = original;
const string & alias = original;

string::const_iterator i = alias.begin();
original.begin();
while(i != alias.end())
    cout << *i++;

```

I have tested this on three string implementations, two of which were reference counted. The reference-counted implementations gave "surprising behaviour" because they invalidated iterators on the first call to non-const begin since construction. The current wording does not permit such invalidation because it does not take into account the first call since construction, only the first call since various member and non-member function calls.

Proposed Resolution:

Change the following sentence in 21.3 paragraph 5 from

Subsequent to any of the above uses except the forms of insert() and erase() which return iterators, the first call to non-const member functions operator[](), at(), begin(), rbegin(), end(), or rend().

to

Following construction or any of the above uses, except the forms of insert() and erase() which return iterators, the first call to non-const member functions operator[](), at(), begin(), rbegin(), end(), or rend().

264. Associative container `insert(i, j)` complexity requirements are not feasible.

Section: 23.1.2 [lib.associative.reqmts](#) **Status:** [Review](#) **Submitter:** John Potter **Date:** 07 Sep 2000

Table 69 requires linear time if `[i, j)` is sorted. Sorted is necessary but not sufficient. Consider inserting a sorted range of even integers into a `set<int>` containing the odd integers in the same range.

Related issue: [102](#)

Proposed Resolution:

In Table 69, in section 23.1.2, change the complexity clause for insertion of a range from "N log(size() + N) (N is the distance from i to j) in general; linear if [i, j) is sorted according to value_comp()" to "N log(size() + N, where N is the distance from i to j".

Rationale:

Testing for valid insertions could be less efficient than simply inserting the elements when the range is not both sorted and between two adjacent existing elements; this could be a QOI issue.

The LWG considered two other options: (a) specifying that the complexity was linear if [i, j) is sorted according to value_comp() and between two adjacent existing elements; or (b) changing to $K \log(\text{size}() + N) + (N - K)$ (N is the distance from i to j and K is the number of elements which do not insert immediately after the previous element from [i, j) including the first). The LWG felt that, since we can't guarantee linear time complexity whenever the range to be inserted is sorted, it's more trouble than it's worth to say that it's linear in some special cases.

265. std::pair::pair() effects overly restrictive

Section: 20.2.2 [lib.pairs](#) **Status:** [Ready](#) **Submitter:** Martin Sebor **Date:** 11 Sep 2000

I don't see any requirements on the types of the elements of the std::pair container in 20.2.2. From the descriptions of the member functions it appears that they must at least satisfy the requirements of 20.1.3 [lib.copyconstructible] and 20.1.4 [lib.default.con.req], and in the case of the [in]equality operators also the requirements of 20.1.1 [lib.equalitycomparable] and 20.1.2 [lib.less-than-comparable].

I believe that the the CopyConstructible requirement is unnecessary in the case of 20.2.2, p2.

Proposed Resolution:

Change the Effects clause in 20.2.2, p2 from

```
-2- Effects: Initializes its members as if implemented: pair() : first(T1()), second(T2())
{}

```

to

```
-2- Effects: Initializes its members as if implemented: pair() : first(), second() {}

```

Rationale:

The existing specification of pair's constructor appears to be a historical artifact: there was concern that pair's members be properly zero-initialized when they are built-in types. At one time there was uncertainty about whether they would be zero-initialized if the default constructor was written the obvious way. The core language was clarified some time ago, however, and there is no longer any doubt that the straightforward implementation is correct.

266. bad_exception::~~bad_exception() missing Effects clause

Section: 18.6.2.1 [lib.bad.exception](#) **Status:** [Review](#) **Submitter:** Martin Sebor **Date:** 24 Sep 2000

The synopsis for `std::bad_exception` lists the function `~bad_exception()` but there is no description of what the function does (the Effects clause is missing).

Proposed Resolution:

Remove the destructor from the class synopses of `bad_alloc` (18.4.2.1 [lib.bad.alloc](#)), `bad_cast` (18.5.2 [lib.bad.cast](#)), `bad_typeid` (18.5.3 [lib.bad.typeid](#)), and `bad_exception` (18.6.2.1 [lib.bad.exception](#)).

Rationale

This is a general problem with the exception classes in clause 18. The proposed resolution is to remove the destructors from the class synopses, rather than to document the destructors' behavior, because removing them is more consistent with how exception classes are described in clause 19.

267. interaction of `strstreambuf::overflow()` and `seekoff()`

Section: D.7.1.3 [depr.strstreambuf.virtuals](#) **Status:** [Open](#) **Submitter:** Martin Sebor **Date:** 5 Oct 2000

It appears that the interaction of the `strstreambuf` members `overflow()` and `seekoff()` can lead to undefined behavior in cases where defined behavior could reasonably be expected. The following program demonstrates this behavior:

```
#include <strstream>

int main ()
{
    std::strstreambuf sb;
    sb.sputc ('c');

    sb.pubseekoff (-1, std::ios::end, std::ios::in);
    return !('c' == sb.sgetc ());
}
```

D.7.1.1, p1 initializes `strstreambuf` with a call to `basic_streambuf<>()`, which in turn sets all pointers to 0 in 27.5.2.1, p1.

27.5.2.2.5, p1 says that `basic_streambuf<>::sputc(c)` calls `overflow(trait::to_int_type(c))` if a write position isn't available (it isn't due to the above).

D.7.1.3, p3 says that `strstreambuf::overflow(off, ..., ios::in)` makes at least one write position available (i.e., it allows the function to make any positive number of write positions available).

D.7.1.3, p13 computes `newoff = seekhigh - eback()`. In D.7.1, p4 we see `seekhigh = epptr() ? epptr() : egptr()`, or `seekhigh = epptr()` in this case. `newoff` is then `epptr() - eback()`.

D.7.1.4, p14 sets `gptr()` so that `gptr() == eback() + newoff + off`, or `gptr() == epptr() + off` holds.

If `strstreambuf::overflow()` made exactly one write position available then `gptr()` will be set to just before `epptr()`, and the program will return 0. But if the function made more than one write position available, `epptr()` and `gptr()` will both point past `pptr()` and the behavior of the program is undefined.

Proposed Resolution:

[Toronto: Dietmar will provide wording for a fix. The general outline is to describe the seek in terms of the put pointer, rather than using seekhigh.]

268. Typo in locale synopsis

Section: 22.1.1 [lib.locale](#) **Status:** [Ready](#) **Submitter:** Martin Sebor **Date:** 5 Oct 2000

The synopsis of the class `std::locale` in 22.1.1 contains two typos: the semicolons after the declarations of the default ctor `locale::locale()` and the copy ctor `locale::locale(const locale&)` are missing.

Proposed Resolution:

Add the missing semicolons, i.e., change

```
// construct/copy/destroy:
    locale() throw()
    locale(const locale& other) throw()
```

in the synopsis in 22.1.1 to

```
// construct/copy/destroy:
    locale() throw();
    locale(const locale& other) throw();
```

269. cstdarg and unnamed parameters

Section: 18.7 [lib.support.runtime](#) **Status:** [NAD](#) **Submitter:** J. Stephen Adamczyk **Date:** 10 Oct 2000

One of our customers asks whether this is valid C++:

```
#include <cstdarg>

void bar(const char *, va_list);

void
foo(const char *file, const char *, ...)
{
    va_list ap;
    va_start(ap, file);
    bar(file, ap);
    va_end(ap);
}
```

The issue being whether it is valid to use `cstdarg` when the final parameter before the `"..."` is unnamed. `cstdarg` is, as far as I can tell, inherited verbatim from the C standard. and the definition there (7.8.1.1 in the ISO C89 standard) refers to "the identifier of the rightmost parameter". What happens when there is no such identifier?

My personal opinion is that this should be allowed, but some tweak might be required in the C++ standard.

Rationale:

Not a defect, the C and C++ standards are clear. It is impossible to use `varargs` if the parameter immediately before `"..."` has no name, because that is the parameter that must be passed to `va_start`. The example given above is broken, because `va_start` is being passed the wrong parameter.

There is no support for extending `varargs` to provide additional functionality beyond what's currently there. For reasons of C/C++ compatibility, it is especially important not to make gratuitous changes in this part of the C++ standard. The C committee has already been requested not to touch this part of the C standard unless necessary.

270. Binary search requirements overly strict

Section: 25.3.3 [lib.alg.binary.search](#) **Status:** [Open](#) **Submitter:** Matt Austern **Date:** 18 Oct 2000

Each of the four binary search algorithms (`lower_bound`, `upper_bound`, `equal_range`, `binary_search`) has a form that allows the user to pass a comparison function object. According to 25.3, paragraph 2, that comparison function object has to be a strict weak ordering.

This requirement is slightly too strict. Suppose we are searching through a sequence containing objects of type `X`, where `X` is some large record with an integer key. We might reasonably want to look up a record by key, in which case we would want to write something like this:

```
struct key_comp {
    bool operator()(const X& x, int n) const {
        return x.key() < n;
    }
}

std::lower_bound(first, last, 47, key_comp());
```

`key_comp` is not a strict weak ordering, but there is no reason to prohibit its use in `lower_bound`.

There's no difficulty in implementing `lower_bound` so that it allows the use of something like `key_comp`. (It will probably work unless an implementor takes special pains to forbid it.) What's difficult is formulating language in the standard to specify what kind of comparison function is acceptable. We need a notion that's slightly more general than that of a strict weak ordering, one that can encompass a comparison function that involves different types. Expressing that notion may be complicated.

Here is a first attempt: the comparison function `comp` must be equivalent to a comparison function of the form `C(pi(x), y)`, and `[first, last)` must be sorted in ascending order by the comparison function `C(pi(x), pi(y))`, where `U` is a synonym for `iterator_traits<ForwardIterator>::value_type`, `x` is a value of type `U`, `y` is a value of type `T`, `C` is a strict weak ordering whose value type is `T`, and `pi` is a homomorphism from `U` to `T`.

In this notation, the existing language refers to the special case where `T` and `U` are the same type and where `pi` is the identity function.

Proposed Resolution:

[Toronto: the issue is still open; there were multiple opinions in the LWG, and several new questions were raised.]

- *Do we really want to specify what ordering the implementor must use when calling the function object? The standard gives specific expressions when describing these algorithms, but it also says that other expressions (with different argument order) are equivalent.*

- *If we are specifying ordering, note that the standard uses both orderings when describing `equal_range`.*
- *Are we talking about requiring these algorithms to work properly when passed a binary function object whose two argument types are not the same, or are we talking about requirements when they are passed a binary function object with several overloaded versions of `operator()`?*
- *The definition of a strict weak ordering does not appear to give any guidance on issues of overloading; it only discusses expressions, and all of the values in these expressions are of the same type. Some clarification would seem to be in order.*

These issues should be considered together. Andy, Beman, Bill, and Howard are interested.]

271. `basic_istream` missing typedefs

Section: 27.6.1.5 [lib.istreamclass](#) **Status:** [New](#) **Submitter:** Martin Sebor **Date:** 02 Nov 2000

Class template `basic_istream` has no typedefs. The typedefs it inherits from its base classes can't be used, since (for example) `basic_istream<T>::traits_type` is ambiguous.

Proposed Resolution:

Add the following to `basic_istream`'s class synopsis in 27.6.1.5 [lib.istreamclass](#), immediately after `public`:

```
// types:
typedef charT          char_type;
typedef typename traits::int_type int_type;
typedef typename traits::pos_type pos_type;
typedef typename traits::off_type off_type;
typedef traits          traits_type;
```

272. Missing parentheses around subexpression

Section: 27.4.4.3 [lib.iostate.flags](#) **Status:** [New](#) **Submitter:** **Date:** Martin Sebor

27.4.4.3, p4 says about the postcondition of the function: If `rdbuf() != 0` then `state == rdstate()`; otherwise `rdstate() == state | ios_base::badbit`.

The expression on the right-hand-side of the operator `==()` needs to be parenthesized in order for the whole expression to ever evaluate to anything but non-zero.

Proposed Resolution:

Add parentheses like so: `rdstate() == (state | ios_base::badbit)`.

273. Missing `ios_base` qualification on members of a dependent class

Section: 27 [lib.input.output](#) **Status:** [New](#) **Submitter:** Martin Sebor **Date:** 02 Nov 2000

27.5.2.4.2, p4, and 27.8.1.6, p2, 27.8.1.7, p3, 27.8.1.9, p2, 27.8.1.10, p3 refer to in and/or out w/o `ios_base::` qualification. That's incorrect since the names are members of a dependent base class (14.6.2 [temp.dep]) and thus not visible.

Proposed Resolution:

Qualify the names with the name of the class of which they are members, i.e., `ios_base`.

274. a missing/impossible allocator requirement

Section: 20.1.5 [lib.allocator.requirements](#) **Status:** [New](#) **Submitter:** Martin Sebor **Date:** 02 Nov 2000

I see that table 31 in 20.1.5, p3 allows `T` in `std::allocator<T>` to be of any type. But the synopsis in 20.4.1 calls for `allocator<U>::address()` to be overloaded on `reference` and `const_reference`, which is ill-formed for all `T = const U`. In other words, this won't work:

```
template class std::allocator<const int>;
```

The obvious solution is to disallow specializations of allocators on `const` types. However, while containers' elements are required to be assignable (which rules out specializations on `const T`'s), I think that allocators might perhaps be potentially useful for `const` values in other contexts. So if allocators are to allow `const` types a partial specialization of `std::allocator<const T>` would probably have to be provided.

Proposed Resolution:

Proposed resolution 1

Add the following definition of a partial specialization immediately below the definition of the primary template in 20.4.1:

```
template <class T>
class allocator<const T> {
public:
    typedef size_t      size_type;
    typedef ptrdiff_t   difference_type;
    typedef const T*    pointer;
    typedef const T*    const_pointer;
    typedef const T&    reference;
    typedef const T&    const_reference;
    typedef const T     value_type;

    template <class U> struct rebind { typedef allocator<U> other; };

    allocator() throw();
    allocator(const allocator&) throw();
    template <class U> allocator(const allocator<U>&) throw();
    ~allocator() throw();

    const_pointer address(const_reference x) const;

    const_pointer allocate(size_type, allocator<void>::const_pointer hint =
0);
    void deallocate(const_pointer p, size_type n);

    size_type max_size() const throw();
```

```

        void construct(const_pointer p, const_reference val);
        void destroy(const_pointer p);
    };

```

Proposed resolution 2

Change the text in row 1, column 2 of table 32 in 20.1.5, p3 from

any type

to

any non-const type

275. Wrong type in num_get::get() overloads

Section: 22.2.2.1.1 [lib.facet.num.get.members](#) **Status:** [New](#) **Submitter:** Matt Austern **Date:** 02 Nov 2000

In 22.2.2.1.1, we have a list of overloads for num_get<>::get(). There are eight overloads, all of which are identical except for the last parameter. The overloads are:

- long&
- unsigned short&
- unsigned int&
- unsigned long&
- short&
- double&
- long double&
- void*&

There is a similar list, in 22.2.2.1.2, of overloads for num_get<>::do_get(). In this list, the last parameter has the types:

- long&
- unsigned short&
- unsigned int&
- unsigned long&
- float&
- double&
- long double&
- void*&

These two lists are not identical. They should be, since get is supposed to call do_get with exactly the arguments it was given.

Proposed Resolution:

In 22.2.2.1.1 [lib.facet.num.get.members](#), change

```

iter_type get(iter_type in, iter_type end, ios_base& str,
              ios_base::iostate& err, short& val) const;

```

to

```
iter_type get(iter_type in, iter_type end, ios_base& str,
             ios_base::iostate& err, float& val) const;
```

276. Assignable requirement for container value type overly strict

Section: 23.1 [lib.container.requirements](#) **Status:** [New](#) **Submitter:** Peter Dimov **Date:** 07 Nov 2000

23.1/3 states that the objects stored in a container must be Assignable. 23.3.1/2 [lib.map](#) states that map satisfies all requirements for a container, while in the same time defining value_type as pair<const Key, T> - a type that is not Assignable.

It should be noted that there exists a valid and non-contradictory interpretation of the current text. The wording in 23.1/3 avoids mentioning value_type, referring instead to "objects stored in a container." One might argue that map does not store objects of type map::value_type, but of map::mapped_type instead, and that the Assignable requirement applies to map::mapped_type, not map::value_type.

However, this makes map a special case (other containers store objects of type value_type) and the Assignable requirement is needlessly restrictive in general.

For example, the proposed resolution of active library issue [103](#) is to make set::iterator a constant iterator; this means that no set operations can exploit the fact that the stored objects are Assignable.

This is related to, but slightly broader than, closed issue [140](#).

Proposed Resolution:

Remove the requirement that the objects stored in a container be Assignable from 23.1/3 and reintroduce it on a case by case basis (for vector and deque.)

Rationale:

list, set, multiset, map, multimap are able to store non-Assignables.

277. Normative encouragement in allocator requirements unclear

Section: 20.1.5 [lib.allocator.requirements](#) **Status:** [New](#) **Submitter:** Matt Austern **Date:** 07 Nov 2000

In 20.1.5, paragraph 5, the standard says that "Implementors are encouraged to supply libraries that can accept allocators that encapsulate more general memory models and that support non-equal instances." This is intended as normative encouragement to standard library implementors. However, it is possible to interpret this sentence as applying to nonstandard third-party libraries.

Proposed Resolution:

In 20.1.5, paragraph 5, change "Implementors" to "Implementors of the library described in this International Standard".

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