Doc. no.: J16/97-0110
WG21/N1148
Date: 4 November 1997
Project: Programming Language C++
Reply to: Beman Dawes
beman@esva.net

Libraries Issues for Morristown

Revision history:

- Initial version: Pre-Morristown mailing.
- Revision 1: Distributed at the beginning of the Morristown meeting. Includes new issues and revisions to proposed resolutions for existing issues.

The issues to be addressed by the library working group in Morristown fall into several categories:

- National Body comments not fully addressed in London.
- National Body comments addressed in London, but possibly needing ratification.
- New issues received since the London meeting.

For ease of management, all library issues are collected together in this document. The base document for the NB comments is ISO/IEC JTC 1/SC22 N2490 Summary of Voting on Final CD (FCD) 14882. The page numbers associated with some items refer to the printed copy of that distributed in London.

Please note that the proposed resolutions come from several sources. For several of the National Body comments, the resolutions were drafted by library working group members at the request of the LWG during the London meeting. Others were provided by the National Body. In some cases, particularly for late arriving issues, the issues list maintainer drafted a proposed resolution based on email or reflector messages.

While the LWG is only obligated to resolve the NB comment issues, the plan is to also resolve the other issues if they are non-controversial and full consensus can be reached.

National Body comments not fully addressed in London

Item 1. London editorial change needs correction: [lib.global.names] 17.3.3.1.2 Global names

An editorial change to reduce redundancy between [lex.name] 2.10 Identifiers and [lib.global.names] 17.3.3.1.2 Global names did not fully move wording from [lex.name] to [lib.global.names].

Proposed resolution:

Change [lib.global.names] from:

> — Each name that begins with an underscore and either an uppercase letter or another underscore (2.11) is reserved to the implementation for any use.

To:

> — Each name that contains a double underscore (__) or begins with underscore and an uppercase letter (2.11) is reserved to the implementation for any use.

Item 2. USA CD2-17-002 Portable C Lib function linkage

One of the tricky issues we resolved in London was the question of portable C Library function linkage.

The key point is to provide "normative encouragement" for C++ linkage, but not require it.

In reading the IEC/ISO Directives at the post-meeting editing session, it seems that the clearest approved expression of normative encouragement is the form "is recommended that..." rather than the IEC/ISO equivalent "should".

Proposed resolution:

17.3.2.2 Linkage [lib.using.linkage]

Change paragraph 2 from:

> It is unspecified whether a name from the Standard C library declared with external linkage has either `extern "C"` or `extern "C++"` linkage.

to:

> Whether a name from the Standard C library declared with external linkage has `extern "C"` or `extern "C++"` linkage is implementation defined. It is recommended that an implementation use `extern "C++"` linkage for this purpose.

Leave the footnote unchanged.

18.3 Start and Termination [lib.support.start.term]

Replace 18.3 function signature:

```
    atexit(void (*f) (void));
```

with:

```
    extern "C" int atexit(void (*f) (void));
    extern "C++" int atexit(void (*f) (void));
```

Replace 18.3 paragraph 4:

> The function `atexit()`, has additional behavior in this International Standard:
>
> — For the execution of a function registered with `atexit`, if control leaves the function because it provides no handler for a thrown exception, `terminate()`is called (18.6.3.3).

with:

> Effects: The `atexit` functions register the function pointed to by f, to be called without arguments at normal program termination.
>
> For the execution of a function registered with `atexit`, if control leaves the function because it provides no handler for a thrown exception, t`erminate`() is called(18.6.3.3).
>
> Implementation Limits: The implementation shall support the registration of at least 32 functions.
>
> Returns: The `atexit` functions return zero if the registration succeeds, nonzero if it fails.

25.4 C library algorithms [lib.alg.c.library]

Replace entire paragraph 2 with:

> The contents are the same as the Standard C library header `<stdlib.h>` with the following modifications:
>
> The function signature:

```
      bsearch(const void *, const void *, size_t, size_t,
              int (*) (const void *, const void *));
```

> is replaced by the two declarations:

```
      extern "C" void *bsearch(const void *key, const void *base,
                         size_t nmemb, size_t size,
                         int (*compar) (const void *, const void *));

      extern "C++" void *bsearch(const void *key, const void *base,
                           size_t nmemb, size_t size,
                           int (*compar) (const void *, const void *));
```

both of which have the same behavior as the original declaration.

The function signature:

```
 qsort(void *, size_t, size_t, int (*)
        (const void *, const void *));
```

is replaced by the two declarations:

```
  extern "C" void qsort(void *base, size_t nmemb, size_t size,
                        int (*compar) (const void *, const void *));

  extern "C++" void qsort(void *base, size_t nmemb, size_t size,
                        int (*compar) (const void *, const void *));
```

both of which have the same behavior as the original declaration.

[Note: Since the function argument `compar()` may throw an exception, `bsearch` and `qsort` are allowed to propagate the exception(17.3.4.8). --end note.]

SEE ALSO: ISO C subclause 7.10.5.

Item 3. Germany "Editorials" 17.2.2.1.2/2 bitmask (p162)

The description of the bitmask operators are counterintuitive and should be changed to the following text or an equivalent.

Proposed resolution:

Change the declaration of bitmask in 17.2.2.1.2 paragraph 2 to:

```
bitmask operator& (bitmask X, bitmask Y)// exposition only.
                  // int_type is an integral type
                  // capable to represent all values
                  // of bitmask
                  { return int_type(X)& int_type(Y); }
bitmask operator| (bitmask X, bitmask Y)
                  { return int_type(X)| int_type(Y); }
bitmask operator^ (bitmask X, bitmask Y)
                  { return int_type(X)^ int_type(Y); }
bitmask operator~ (bitmask X)  { return int_type(~X); }
bitmask& operator&= (bitmask& X, bitmask Y)
                  { X = X&Y; return X; }
bitmask& operator|= (bitmask& X, bitmask Y)
                  { X = X|Y; return X; }
bitmask& operator^= (bitmask& X, bitmask Y)
                  { X = X^Y; return X; }
```

Item 4. UK 626, Sweden _20451 Auto_ptr semantics; (p61)

Further proposed resolutions for this issue has arisen since the LWG discussion at the London meeting. Discussion will be reopened in a final attempt to convert NB no votes to yes votes.

Proposed resolution Part A. (choose one):

1. No change to standard, take no further action.

2. Remove the transfer of ownership semantics from auto_ptr (Make the copy constructor and operator= private.)

3. Same as 2 but rename it safe_ptr.

4. Remove auto_ptr.

5. Deprecate auto_ptr.

6. Change the semantics of auto_ptr to prohibit certain operations if the auto_ptr does not have ownership.

7. Change the arguments of auto_ptr's copy constructor and operator= to non-const.

8. Following [lib.auto.ptr] 20.4.5 paragraph 2, add non-normative text:

   [*Note:* The uses of auto_ptr include providing temporary exception-safety for dynamically allocated memory, passing ownership of dynamically allocated memory to a function, and returning dynamically allocated memory from a function. Auto_ptr does not meet the CopyConstructible and Assignable requirements for Standard Library container elements and thus instantiating a Standard Library container with an auto_ptr results in undefined behavior. *–end note*]

Proposed resolution Part B:

The committee is hereby authorized to begin development of a Technical Report specifying additional Standard Library smart pointers such as safe_ptr (like auto_ptr but no transfer of ownership) and counted_ptr (see 94-0202R1/N0589R1). The Technical Report may also specify changes to auto_ptr to improved the safety of its transfer of ownership semantics.

Item 5. Sweden _2133/b Return value of capacity() (p71), US CD2-21-004 Capacity() desc unclear (p72)

[Record keeping SNAFU; what was the resolution of these issues? May have been unintentionally skipped.]
Section: 21.3.3 [lib.string.capacity] par 8 and 9

A) In [lib.string.capacity] paragraphs 8 and 9: Is reserve() guaranteed to accept any argument, even size_type(-1)?

Proposed resolution: (none yet.)

B) The description of capacity() is unclear, it doesn't stand for itself. Maybe we should define it as:

Proposed resolution:

> Returns: a value not less than the value of res_arg of the last call of reserve(), or an unspecified value if reserve() has not been called for this object. The returned value is not less than size().

Item 6. Germany CD2-21-012 Extraction of C strings and basic_strings inconsistent re null bytes (p74)

[What was the resolution of this issue? May have fallen into the cracks between two working groups.]

The extractors for C strings, i.e. charT*, are slightly different from their counterparts for basic_string<charT>. The extraction of a C string [lib.istream::extractors] stops when a null byte is found in the next position.

The same is not mentioned as a terminating condition for the basic_string extractor [lib.string.io].

An oversight or intention?

** Discussion: Jerry: I believe stopping on NULL was iostream classic behavior, and I don't see any reason to change it. Null's aren't allowed to appear in NTBS's.

Proposed Resolution:

Add to 21.3.7.9 [lib.string.io] extractor description:

> The extraction of a basic_string shall stop when a null byte is found in the next position.

Requester: Klaus Kreft & Angelika Langer

Item 7. Germany CD2-21-013 Extraction of C strings and basic_strings inconsistent re width (p75)

[What was the resolution of this issue? May have fallen into the cracks between two working groups.]

21.3.7.9 [lib.string.io] and 27.6.1.2.3 [lib.istream::extractors]

The extractors for C strings, i.e. charT*, are slightly different from their counterparts for basic_string<charT>. The number of characters extracted is different. For C strings it is width()-1; for basic_string it is width(). Is this to leave room for a terminating '\0' in a C string?

** Discussion: Jerry: Extractor's generally ignore width. But for charT I needed some way to pass the size of the array you're storing into, and yes you need the extra char to store a null. Since there is no such limitation on the string extractor I think I would have opted for ignoring width there too.

If the argument for looking at width is to make it the same as the charT* extractor then you're right that this is an inconsistency.

Proposed Resolution:

Requester: Klaus Kreft & Angelika Langer

Item 8. Japan "Editorials" 7 21.1.1 paragraph 1 (p154)

Traits is a general technique to define requirements and constraints to templatized classes and parameters. In fact, there is an inserter-traits in the STL part of the draft. So the paragraph sounds too restrictive as it describes only the use for character objects. We recommend the following substitution.

Proposed resolution:

Change 21.1.1 paragraph 1 to:

> " --traits: Traits is a class which encapsulates a set of the defined types and functions in order to give some constraints to templatized classes. All traits appeared in clauses 21, 22 and 27 are char-traits which are special traits handling character objects in any implementation of the string and iostream libraries."

Item 9. Japan "Editorials" 8 21.1.2 paragraph 1 (p154)

Generally speaking, char-like types (especially user-defined ones) do not need to have all the methods in this requirement list.

Proposed resolution:

Add the following to 21.1.2 paragraph 1:

> "Table 37 does not require that the all members in the list should be defined. Any requirement in Table 37 is required only if the corresponding member is defined."

Item 10. Japan "Editorials" 9 Clause 21.1.3 (p154)

There is no definition of the typedef TRAIT_T, and moreover it does not appear in the rest of the draft except three times in this clause. So we recommend to simplify the clause by not using the definition and by replacing

Proposed resolution:

Remove the definition of TRAIT_T and replace:

> "CHAR_T and TRAIT_T"

by

> "the character container type and its related traits".

Item 11. Japan "Editorials" 10 21.1.3 paragraph 1 (p155)

For the templatized char traits, there is no option for the type CHAR_T except charT. But this is not explicitly stated.

Proposed resolution:

Add the following statement to 21.1.3 paragraph 1:

> "The CHAR_T coincides with the template argument charT when the char traits is defined as a templatized class."

Item 12. Japan "Editorials" 11 Clause 21.1.3 (p155)

Need some requirements upon the POS_T, OFF_T, and STATE_T.

In this clause, POS_T, OFF_T and STATE_T stand for a pair of stream-position-indicator types and their related conversion state type for a certain specialized 'traits'. We had better clarify their requirements so as to ensure for library users to attach their own traits definitions to a certain implementation of templatized iostream library. A typical description of the requirements might be similar to those in Table 89 (27.4.4) and template type fpos (27.4.3) as follows:

Proposed resolution:

After 21.1.3 paragraph 11, add:

21.1.3.1 POS_T, OFF_T, and STATE_T requirements

Operations specified in Table 38 are permitted. In that table,

-- P refers to type POS_T,
-- p and q refer to an value of type P,
-- O refers to type OFF_T,
-- o refers to a value of type OFF_T,
-- S refers to a type STATE_T,
-- s refers to a value of type STATE_T, and
-- i refers to a value of type int.

Table 38 --- POS_T, OFF_T, and STATE_T requirements

expression	return type	operational semantics	assertion/note pre/post-conditions
P(-1)			invalid positional value `p == P(-1)` post: not equal to any valid value.

P p(-1);
P p = -1;
			post: p == P(i)

P(s)			p == P(s) note: a destructor is assumed post: offset is zero

P p(s);
P p = s;
			post: p == P(s)
			offset is zero

p.state();	STATE_T	get state	post: offset is unchanged
p.state(s);	void	set state	post: p.state() == s offset is unchanged
P(o)	POS_T	converts from offset	
O(p)	OFF_T	converts from offset	

p == q	convertible to bool	== is an equivalence relation	
p != q	convertible to bool	!(p == q)	

q = p + o
p += o
	POS_T	+ offset	q-o == p

q = p - o
p -= o
	POS_T	- offset	q+o == p

o = p - q	OFF_T	distance	q+o == p

Item 13. Japan Technical 3 Clause 21.1.2 paragraph 1 Table 37 and Clause 21.1.2 paragraph 2

The base definition of the struct template

> template<class charT> struct char_traits { };

has dropped default definitions of speed-up functions such as 'find', 'move', and 'copy' which were given in CD1. They should be retained as the speed-up functions are quite useful as default definitions in some traits.

Post London comment from Robert Klarer:

in London we looked at a national body comment, Japan 3, about the speed-up functions in the char_traits template. Japan believed that these functions had been removed and wanted them reinstated. We were all puzzled by this request since no one recalled any explicit decision to remove the functions. Our puzzlement was dispelled when someone noticed that 21.1.2 paragraph 2 was written like this:

> The struct template
>
> template<class charT> struct char_traits{};
>
> shall be provided in the header <string> as a basis for explicit specializations.

It seemed that the author of the comment had inferred from the pseudo-definition of char_traits<charT> above that the char_traits template was required to be empty. We agreed to address the comment by eliminating the source of confusion by eliminating the curly braces, turning the empty definition of the char_traits template into a forward declaration.

I now think that the comment from Japan was inspired by 21.1 paragraph 4 (see below).

Proposed resolution:

Replace 21.1 paragraph 4,

> This subclause specifies a struct template, char_traits<charT>, with no members, to appear in the header <string> along with two explicit specializations of it, char_traits<char> and char_traits<wchar_t> which satisfy the requirements below.

with:

> This subclause specifies a struct template, char_traits<charT> and two explicit specializations of it, char_traits<char> and char_traits<wchar_t>, all of which appear in the header <string> and satisfy the requirements below.

Item 14. US CD2-23-013 Container requirements do not talk about allocator interface (p112)

[Also see Item 17, CD2-23-018 Container member typedefs unimplementable]

The interface for allocator<T> is not necessarily the same as that for allocator<S> when S != T. The CD does not specify which interfaces are allowed for the allocator template argument of containers. For example, is the following allowed:

```
List<T, allocator<double> > l;
```

Is it required to work?

Requester: Jerry Schwarz

Item 15. US CD2-23-015 Is X:: reverse_iterator convertable to X::const_reverse_iterator? (p113)

[Beman] This was deferred because several knowledgeable people were out of the room. Note that the prior issue, CD2-23-014, proposing conversion of container iterators to const_iterators was rejected.

If X is a containers, then X::iterator is (or ought to be) convertible to X::const_iterator. X::reverse_iterator and X::const_reverse_iterator, however, are typedefs that involve the reverse_iterator<> template, so whether or not this guarantee exists for X::reverse_iterator and X::const_reverse_iterator depends on how that template is defined.

In fact, that conversion does not exist. reverse_iterator<X::iterator> and reverse_iterator<X::const_iterator> are unrelated types, and reverse_iterator<> has no member function that would provide a conversion between them.

Proposed resolution:

Add a member template "generalized copy constructor" to reverse_iterator<>, so that reverse_iterator<U> is convertible to reverse_iterator<V> if and only if U is convertible to V.

(Generalized copy constructors are used elsewhere in the standard library; see pair<>, for example.)

Item 16. UK 699 template member function ambiguities (p105); should apply to more widely

The London "do the right thing" resolution for template ambiguities should apply to basic_string as well as the containers in clause 23, and to members append(), assign(), and replace() as well as constructors and insert(). Yet the enabling wording in [lib.sequence.reqmts] 23.1.1 paragraph 9 begins "For every sequence defined in this clause:", and goes on to deal only with constructors and insert().

Proposed resolution:

Change [lib.sequence.reqmts] 23.1.1 paragraph 9 first sentence to:

> For every sequence defined in this clause and in the Strings library, clause 21:

Change the portion which reads:

> — the member function:

```
        template <class InputIterator>
        void insert(iterator p, InputIterator f, InputIterator l)
```

> shall have the same effect as:

```
        insert(p,
               static_cast<typename X::size_type>(f),
               static_cast<typename X::value_type>(l)).
```

To:

> — the member functions in the forms:

```
        template <class InputIterator>  // such as insert()
        rt fx1(iterator p, InputIterator f, InputIterator l)

        template <class InputIterator>  // such as append(), assign()
        rt fx2(InputIterator f, InputIterator l)

        template <class InputIterator>  // such as replace()
        rt fx3(iterator i1, iterator i2, InputIterator f, InputIterator l)
```

shall have the same effect, respectively, as:

```
        fx1(p,
            static_cast<typename X::size_type>(f),
            static_cast<typename X::value_type>(l))

        fx2(static_cast<typename X::size_type>(f),
           static_cast<typename X::value_type>(l))

        fx3(i1, i2,
            static_cast<typename X::size_type>(f),
            static_cast<typename X::value_type>(l))
```

Item 17. CD2-23-018 Container member typedefs unimplementable

[Also see Item 14, US CD2-23-013 Container requirements do not talk about allocator interface]

Container reference types need to be made consistent by ensuring that reference types are in terms of T&. This will guarantee that regardless of the allocator a container is parameterized on, the reference for the container is T& and not based on allocator<any_type>::reference i.e. any_type&. It falls out of this that in the case of map and multimap the return value from operator [] () is T& which is what is expected.

Comment from Matt Austern:

> I think that item 17, issue CD2-23-018, is not so much incorrect as incomplete.
>
> It's true that we need to worry about Container::reference and Container:const_reference. What about the other nested typedefs that are defined in terms of allocators, though, like Container::pointer?
>
> The issue is similar: if X is a vector<int, allocator<double> >, then what should X::pointer be?
>
> Options.
>
> (1) Just say no. Decree that an allocator's value type must be the same as the container's value type. This is my favorite.   vector<int, allocator<double> > is pathological.
> (2) Rely on the infamous weasel words: just say that   vector<T, Allocator>::pointeris T*. We've already said that implementors who allow allocators with non-standard pointer type have to describe the semantics of using those allocators with their containers.
> (3) Say that X::pointer, too, is implementation defined.
> (4) Actually spell out what X::pointer is, in terms of rebind. This is my least favorite option, because the syntax is absolutely appalling.
>   typedef typename Allocator::template rebind<value_type>::other::pointer pointer;

Proposed resolution:

For the following:

> 23.2.1 Template class deque [lib.deque]
> 23.2.2 Template class list [lib.list]
> 23.2.4 Template class vector [lib.vector]
> 23.3.1 Template class map [lib.map]
> 23.3.2 Template class multimap [lib.multimap]
> 23.3.3 Template class set [lib.set]
> 23.3.4 Template class multiset [lib.multiset]

Change:
```
        typedef typename Allocator::reference          reference;
```

```
       typedef typename Allocator::const_reference    const_reference;
```
To:
```
       typedef T&                                     reference;
       typedef T const&                               const_reference;
```

Item 18. Japan "Editorials" 13 Clause 26.2.2 (p156)

Global operator functions 'operator +', 'operator -', 'operator *', 'operator /', 'operator ==' and 'operator !=' are declared here. But the functions are already declared in the clause 26.2.1. Furthermore, each declaration between both clauses differs a little.

Proposed resolution:

Remove the declarations in 26.2.2 for global operator functions 'operator +', 'operator -', 'operator *', 'operator /', 'operator ==' and 'operator !='.

Item 19. Japan "Editorials" 14 26.2.6 paragraph 10 (p156)

Proposed resolution:

Change:

```
  template<class T> bool operator!=(complex<T>& lhs, complex<T>& rhs);
  template<class T> bool operator!=(complex<T>& lhs, const T& rhs);
  template<class T> bool operator!=(const T& lhs, complex<T>& rhs);
```

to:

```
  template<class T> bool operator!=(const complex<T>& lhs, const complex<T>& rhs);
  template<class T> bool operator!=(const complex<T>& lhs, const T& rhs);
  template<class T> bool operator!=(const T& lhs, const complex<T>& rhs);
```

Item 20. Library typename syntax and related issues

In London, Robert Klarer committed to a review of typename syntax and related issues, to address several National Body comments regarding these issues. Here are the results of his review:

Proposed resolutions:

18.2.1 In "Header<limits> synopsis," add "template<>" to each of the 14 explicit specializations.

18.2.1.4 paragraph 2. Add "template<>" to the explicit specialization numeric_limits<float>.

20 General utilities library; in table 32, change the expression "typename X::rebind<U>::other" in the eighth row to "typename X::template rebind<U>::other."

20.3.5, paragraph 1. Change the declaration of

```
        bool operator()(const argument_type& x) const;
```
to:
```
        bool operator()(const typename Predicate::argument_type& x) const;
```

20.3.5, paragraph 3. Change the declaration of

```
        bool operator()(const first_argument_type& x
                        const second_argument_type& y) const;
```
to:
```
        bool operator()(const typename Predicate::first_argument_type& x,
                        const typename Predicate::second_argument_type& y) const;
```

20.3.6.1 Change the declaration of

```
        result_type binder1st<Operation>::operator()(const argument_type& x) const;
```
to:
```
        typename Operation::result_type
          operator()(const typename Operation::second_argument_type& x) const;
```

20.3.6.3 Change the declaration of

```
        result_type binder2nd<Operation>::operator()(const argument_type& x) const;
```
to:
```
        typename Operation::result_type
          operator()(const typename Operation::first_argument_type& x) const;
```

20.4.1 In the declaration

```
        pointer allocate(
          size_type, typename allocator<void>::const_pointer hint = 0);
```

the use of the typename keyword is redundant, though not strictly wrong. The same function is declared without the typename keyword in 20.4.1.1,
paragraph 2.

20.4.4.1 Paragraph 1 insert missing ‘>’ in Effects:

```
        typename iterator_traits<ForwardIterator>::value_type(*first);
```

20.4.4.2, paragraph 1 insert missing ‘>’ in Effects:

```
        typename iterator_traits<ForwardIterator>::value_type(x);
```

20.4.4.3, paragraph 1 insert missing ‘>’ in Effects:

```
        typename iterator_traits<ForwardIterator>::value_type(x);
```

21.3.7.9, paragraph 1.Change the first sentence to read

> Effects: Begins by constructing a sentry object k as if k were constructed by typename basic_istream<charT, traits>::sentry k(is).

21.3.7.9, paragraph 3. Change the first sentence to read

> Effects: Begins by constructing a sentry object k as if k were constructed by typename basic_ostream<charT, traits>::sentry k(os).

21.3.7.9, paragraph 5. Change the first sentence to read

> Effects: Begins by constructing a sentry object k as if by typename basic_istream<charT, traits>::sentry k(is, true).

22.2.1.2 In ctype_byname<charT>, mask is a type-dependent name and therefore needs a
typedef.

Also, add "namespace std {" at the top of the ctype_byname template.

22.2.1.4 Change the first line of ctype_byname<char> from

```
        template <> class ctype_byname<char> : public ctype<charT> {
```
to
```
        template <> class ctype_byname<char> : public ctype<char> {
```

Also, add "namespace std {" at the top of the subsection.

22.2.3.2 To the template numpunct_byname<charT>, add typedefs for char_type and string_type:

```
        typedef charT                       char_type
        typedef basic_string<charT>         string_type
```

22.2.4.2 To the template collate_byname<charT>, add a typedef for string_type:

```
        typedef basic_string<charT>         string_type
```

Also, finish the namespace declaration with a closing "}".

22.2.5.2 To the template time_get_byname<charT, InputIterator>, add typedefs for iter_type and dateorder.

22.2.5.4 To the template time_put_byname<charT, OutputIterator>, add a typedef for iter_type.

22.2.6.3, paragraph 3 Change du_grouping to do_grouping.

22.2.6.4 To the template moneypunct_byname<charT, bool>, add typedefs for string_type and pattern.

22.2.7.2 To the template messages_byname<charT, bool>, add typedefs for string_type and catalog.

23.1, paragraph 9 The typename keyword appears four times in this paragraph. Remove it as it is redundant each time.

23.2 In "Header <queue> synopsis," change

```
        template <class T, class Container = vector<T>,
            class Compare = less<Container::value_type> >
```

to

```
        template <class T, class Container = vector<T>,
            class Compare = less<typename Container::value_type> >
```

23.2.1.1 paragraph 2 Change

```
        explicit deque(size_t n, const T& value = T(),
            const Allocator& = Allocator());
```

to

```
        explicit deque(size_type n, const T& value = T(),
            const Allocator& = Allocator());
```

23.2.3.1 In the template queue<class T, class Container>, change the typedef of container_type from

```
        typedef typename Container      container_type;
```

to

```
        typedef Container               container_type;
```

23.2.3.2 Change the declaration of priority_queue<class T, class Container, class Compare> to

```
        template <class T, class Container = Vector<T>,
            class Compare = less<typename Container::value_type> >
```

Also, remove the typename keyword from the typedef of container_type.

23.2.3.3 Remove the typename keyword from the typedef of container_type.

23.3.5.1 paragraph 2 Change the declaration of the bitset constructor from

```
        template <class charT, class traits, class Allocator>
        explicit
        bitset(const basic_string<charT, traits, Allocator>& str,
              basic_string<charT, traits, Allocator>::size_type pos = 0,
              basic_string<charT, traits, Allocator>::size_type n =
                basic_string<charT, traits, Allocator>::npos);
```
to
```
       template <class charT, class traits, class Allocator>
       explicit
       bitset(const basic_string<charT, traits, Allocator>& str,
              typename basic_string<charT, traits, Allocator>::size_type pos = 0,
              typename basic_string<charT, traits, Allocator>::size_type n =
                basic_string<charT, traits, Allocator>::npos);
```

24.2 The declaration of distance should be changed from

```
       template <class InputIterator>
         iterator_traits<InputIterator>::difference_type
         distance(InputIterator first, InputIterator last);
```
to
```
       template <class InputIterator>
         typename iterator_traits<InputIterator>::difference_type
         distance(InputIterator first, InputIterator last);
```

24.3.1 paragraph 2 Change the definition of the iterator_traits template from

```
       template<class Iterator> struct iterator_traits {
         typedef Iterator::difference_type difference_type;
         typedef Iterator::value_type value_type;
         typedef Iterator::pointer pointer;
         typedef Iterator::reference reference;
         typedef Iterator::iterator_category iterator_category;
       };
```
to
```
       template<class Iterator> struct iterator_traits {
         typedef typename Iterator::difference_type difference_type;
         typedef typename Iterator::value_type value_type;
         typedef typename Iterator::pointer pointer;
         typedef typename Iterator::reference reference;
         typedef typename Iterator::iterator_category iterator_category;
       };
```

paragraph 3 Change the example from

```
       template <class BidirectionalIterator>
       void reverse(BidirectionalIterator first, BidirectionalIterator last) {
         iterator_traits<BidirectionalIterator>::difference_type n =
           distance(first, last);
         --n;
         while(n > 0) {
           iterator_traits<BidirectionalIterator>::value_type tmp = *first;
           *first++ = * --last;
           *last = tmp;
           n -= 2;
         }
       }
```
to
```
       template <class BidirectionalIterator>
       void reverse(BidirectionalIterator first, BidirectionalIterator last) {
         typename iterator_traits<BidirectionalIterator>::difference_type n =
           distance(first, last);
         --n;
         while(n > 0) {
           typename iterator_traits<BidirectionalIterator>::value_type tmp = *first;
```

```
    *first++ = * --last;
    *last = tmp;
    n -= 2;
  }
}
```

24.3.4 paragraph 3 Change the declaration of distance from

```
template<class InputIterator>
  iterator_traits<InputIterator>::difference_type
    distance(InputIterator first, InputIterator last);
```

to

```
template<class InputIterator>
  typename iterator_traits<InputIterator>::difference_type
    distance(InputIterator first, InputIterator last);
```

24.4.1.1 Change the reverse_iterator template from

```
template <class Iterator>
class reverse_iterator :
  public iterator<iterator_traits<Iterator>::iterator_category,
                iterator_traits<Iterator>::value_type,
                iterator_traits<Iterator>::difference_type,
                iterator_traits<Iterator>::pointer,
                iterator_traits<Iterator>::reference> {
protected:
  Iterator current;
public:
  typedef Iterator iterator_type;
  reverse_iterator();
  explicit reverse_iterator(Iterator x);

  Iterator base() const;    // explicit
  Reference operator*() const;
  Pointer   operator->() const;

  reverse_iterator& operator++();
  reverse_iterator& operator++(int);
  reverse_iterator& operator--();
  reverse_iterator& operator--(int);

  reverse_iterator  operator+ (difference_type n) const;
  reverse_iterator& operator+=(difference_type n);
  reverse_iterator  operator- (difference_type n) const;
  reverse_iterator& operator-=(difference_type n);
  reference operator[](difference_type n) const;
};
```

to

```
template <class Iterator>
class reverse_iterator :
  public iterator<typename iterator_traits<Iterator>::iterator_category,
                typename iterator_traits<Iterator>::value_type,
                typename iterator_traits<Iterator>::difference_type,
                typename iterator_traits<Iterator>::pointer,
                typename iterator_traits<Iterator>::reference> {
protected:
  Iterator current;
public:
  typedef Iterator iterator_type;
  typedef typename iterator_traits<Iterator>::difference_type difference_type;
  typedef typename iterator_traits<Iterator>::reference reference;
```

```
   typedef typename iterator_traits<Iterator>::pointer pointer;

   reverse_iterator();
   explicit reverse_iterator(Iterator x);

   Iterator base() const;    // explicit
   reference operator*() const;
   pointer   operator->() const;

   reverse_iterator& operator++();
   reverse_iterator& operator++(int);
   reverse_iterator& operator--();
   reverse_iterator& operator--(int);

   reverse_iterator  operator+ (difference_type n) const;
   reverse_iterator& operator+=(difference_type n);
   reverse_iterator  operator- (difference_type n) const;
   reverse_iterator& operator-=(difference_type n);
   reference operator[](difference_type n) const;
 };
```

(the differences are: the addition of typename keywords where appropriate, the addition of typedefs for difference_type, pointer and reference, replacement of "Reference" with "reference", "Pointer" with "pointer").

24.5.3 The istreambuf_iterator template is too wide for the page in my hardcopy. Add a line break

25 Algorithms library. In "Header <algorithm> synopsis," change the declarations of count and count_if from

```
template<class InputIterator, class T>
  iterator_traits<InputIterator>::difference_type
    count(InputIterator first, InputIterator last, const T& value);
template<class InputIterator, class Predicate>
  iterator_traits<InputIterator>::difference_type
    count_if(InputIterator first, InputIterator last, Predicate pred);
```

to

```
template<class InputIterator, class T>
  typename iterator_traits<InputIterator>::difference_type
    count(InputIterator first, InputIterator last, const T& value);
template<class InputIterator, class Predicate>
  typename iterator_traits<InputIterator>::difference_type
    count_if(InputIterator first, InputIterator last, Predicate pred);
```

25.1.6 Make the same changes for count and count_if as in "Header <algorithm> synopsis.

26.2.3 Add "template<>" to the explicit specializations complex<float>, complex<double> and complex<long double>.

26.2.5 For the sake of consistency, change

```
complex<T>& operator+=(const T& rhs);
```

to

```
template <class T>
complex<T>& operator+=(const T& rhs);
```

Also do this for operator-=(), operator*=() and operator/=.

Fix the Troff bug in the typedef of traits_type.

27.8.1.3 paragraph 2 Change: ::fopen(s, modstr)
to: std::fopen(s, modstr).

27.8.1.3 paragraph 6 Change: ::fclose(file) to: std::fclose(file)

and change: If any of the calls to overflow or ::fclose fails...
to: If any of the calls to overflow or std::fclose fails....

27.8.1.4 paragraph 11 Change "::fseek(file, distance*off, whence)" to "std::fseek(file, distance*off, whence)."

27.8.1.5 Change: void open(const char* s, openmode mode = in);
to: void open(const char* s, ios_base::openmode mode = in);

27.8.1.6 paragraph 1Change: explicit basic_ifstream(const char* s, openmode mode = in);
to: explicit basic_ifstream(const char* s, ios_base::openmode mode = in);

27.8.1.7 paragraph 2 Change: void open(const char* s, openmode = in);
to: void open(const char* s, ios_base::openmode = in);

27.8.1.9 paragraph 1Change: explicit basic_ofstream(const char* s, openmode mode = out);
to: explicit basic_ofstream(const char* s, ios_base::openmode mode = out);

paragraph 2 Change: void open(const char *, openmode mode = out);
to: void open(const char *, ios_base::openmode mode = out);

Annex D Compatibility features.

D.7.3.1 paragraph 2 Change: strstreambuf(s,n,s+::strlen(s))
to: strstreambuf(s,n,s+std::strlen(s))

D.7.4 Class strstream has the following typedefs

```
typedef typename char_traits<char>::int_type int_type;
typedef typename char_traits<char>::pos_type pos_type;
typedef typename char_traits<char>::off_type off_type;
```

the typename keyword is redundant but harmless in each case. Make no change.

D.7.4.1 paragraph 2 Change: strstreambuf(s,n,s+::strlen(s))
to: strstreambuf(s,n,s+std::strlen(s))

National Body comments possibly needing ratification

Item 21. Ratification of certain London changes

Some London changes to WP may not be fully reflected in London resolutions due to expedited London voting procedures such as voting on troff diffs.

Proposed resolution:

The resolution of the following issues and resulting changes to the WP are hereby ratified:

- US CD2-23-007 Can library containers be instantiated on incomplete types? (p111) [No]
- UK 677 Container requirements table 66 Should "distance type" be "difference type"? (p113) [Yes. Also in two places in lib.iterator.traits para 1]
- UK 691(p114) [Add typename to two 23.3.1 [lib.map] typedefs]
- US CD2-23-011(p114) Lifetime of iterators and references into containers [clarification added]
- UK 680 (p115) [23.1.2 table 70 typo corrected]
- US CD2-23-006(p115) Impact of insert() and erase() on containers not specified [lib.associative.reqmts requirement added]

Item 22. Template class codecvt<intenT, externT, stateT> change questioned

22.2.1.t now defines the member types:

```
typedef internT intern_type;
typedef externT extern_type;
```

in place of:

```
typedef internT from_type;
typedef externT to_type;
```

I can find no resolution that calls for this change. –pjp

Proposed resolution: The change to the WP is hereby ratified.

Item 23. Template class reverse_iterator<Iterator> iterator_type questioned

24.4.1.1 Template class reverse_iterator<Iterator> now defines the member type:

```
typedef Iterator iterator_type;
```

I can find no resolution that calls for this addition. –pjp

Proposed resolution: The change to the WP is hereby ratified.

New issues received since the London meeting

Item 24. Resolution of CD2-20-006 Incomplete; mem_fun adaptors can't be used with const member functions

In London four operator() functions where made const. This, however, did not address the larger issue brought forward by 20-006. Matt Austern has asked that the larger issue be reconsidered in Morristown. He has provide a fresh description and proposed resolution:

Suppose that you have a vector of vectors, and that you want to make a list of those vectors' sizes. It looks like you can do this using existing library components: all you need is to call size() to each element of a range, and that's just what the mem_fun adaptors are for. So you just write something like this.

```
   transform(V.begin(), V.end(), Dest.begin(),
            mem_fun_ref(vector<X>::size));
```

Except that it doesn't work; it won't compile.

(1) The mem_fun_ref_t adaptor expects an argument of type S (T::*f)(), and we're not giving it an argument of that form. The type of vector<X>::size is size_t (vector<X>::*) const, not size_t (vector<X>::*).

(2) You can't pass a const argument to a mem_fun_ref_t. That is, if F is a mem_fun_ref_t, and x is a const object, you can't write F(x). No surprise: a mem_fun_ref is a wrapper for a non-const member function, and can't call a non-const member function on a const object.

The problem is that pointer-to-const-member-function and pointer-to-non-const-member-function are completely unreleated types; there is no way to convert from one to the other, and there is no way to unify them as a single template.

Proposed resolution:

Add four more mem_fun adaptors. They would be identical to the existing four, except that they are adaptors for const member functions instead of non-const member functions. Also overload the four helper funtions mem_fun, mem_fun_ref, mem_fun1, and mem_fun1_ref, so that (for example) mem_fun(X) would create a mem_fun_t if X is a non-const member function and would create a const_mem_fun_t if X is a const member function.

As usual with tiny little adaptors like this, the code is much simpler and clearer than the description. It has been tested.

```
template <class S, class T>
class const_mem_fun_t : public unary_function<const T*, S> {
public:
  explicit const_mem_fun_t(S (T::*pf)() const) : f(pf) {}
  S operator()(const T* p) const { return (p->*f)(); }
private:
  S (T::*f)() const;
};

template <class S, class T>
class const_mem_fun_ref_t : public unary_function<T, S> {
public:
  explicit const_mem_fun_ref_t(S (T::*pf)() const) : f(pf) {}
  S operator()(const T& r) const { return (r.*f)(); }
private:
  S (T::*f)() const;
};

template <class S, class T, class A>
class const_mem_fun1_t : public binary_function<const T*, A, S> {
public:
  explicit const_mem_fun1_t(S (T::*pf)(A) const) : f(pf) {}
  S operator()(const T* p, A x) const { return (p->*f)(x); }
private:
  S (T::*f)(A) const;
};

template <class S, class T, class A>
class const_mem_fun1_ref_t : public binary_function<T, A, S> {
public:
  explicit const_mem_fun1_ref_t(S (T::*pf)(A) const) : f(pf) {}
  S operator()(const T& r, A x) const { return (r.*f)(x); }
private:
  S (T::*f)(A) const;
};

template <class S, class T>
inline const_mem_fun_t<S,T> mem_fun(S (T::*f)() const) {
  return const_mem_fun_t<S,T>(f);
}

template <class S, class T>
inline const_mem_fun_ref_t<S,T> mem_fun_ref(S (T::*f)() const) {
  return const_mem_fun_ref_t<S,T>(f);
}

template <class S, class T, class A>
inline const_mem_fun1_t<S,T,A> mem_fun1(S (T::*f)(A) const) {
  return const_mem_fun1_t<S,T,A>(f);
}

template <class S, class T, class A>
inline const_mem_fun1_ref_t<S,T,A> mem_fun1_ref(S (T::*f)(A) const) {
  return const_mem_fun1_ref_t<S,T,A>(f);
}
```

Item 25. Fallout from CD2-27-042 (22.2.2.2)

In London we passed Issue CD2-27-042, changing the operator<<(void*) to operator<<(const void*) so that const pointers could be printed.

Unfortunately I just noticed that this did not go far enough, the same changes need to be made to the num_put::put() and num_put::do_put() member functions that currently take a void*.

The Plauger library already has the correct prototypes. --Judy Ward

Proposed resolution:

In four places in Section 22.2.2.2, change: `void* val`
to: `const void* val`.

Item 26. [lib.limits] 18.2.1 Numeric limits specializations need to define static member constants

Is there a reason that min() and max() are static member functions, while min_exponent and max_exponent() are static member variables?

In the case of these numeric_limits members, the min() and max() values for some numeric type aren't necessarily representable in the

```
  template <>
    struct numeric_limits<Num> {
      ...
      static const Num min = 0;
      ...
    };
```

notation, and so might as well be functions, hopefully in-line. Making them static would risk initialization-order problems.

However, the exponents were assumed to be representable as integer constants; making them static member constants allows them to be used as compile-time constants, e.g. to dimension an array big enough to hold one.

[Nathan Myers] Unfortunately I don't think the draft actually requires specializations to define the exponents and other static member constants in the notation above. It should.

[Beman Dawes] 18.2.1 paragraph 3 already says:

> For all members declared static const in the numeric_limits template, specializations shall define these values in such a way that they are usable as integral constant expressions.

Proposed resolution:

No change to WP.

Item 27. [lib.limits] 18.2.1 Numeric limits has_denorm undeterminable until runtime

Matt Austern: numeric_limits<>::has_denorm is a constant of type bool. That's a problem, though: it isn't necessarily possible to determine whether or not floating-point types have denormalized types until run time.

I can see three options.
 (1) Change this from a constant to a function.
 (2) Change the type from bool to an enum, something along the lines of float_round_style. There would be three values: true, false, or don't know.
 (3) Much like option (2), but lump "don't know" in with either "true" or "false". That is, decide that has_denorm is true if the type might have denormalized values, or else that has_denorm is false if the type might not have denormalized values.

Nathan Myers: I believe this was on the issues list at one time. It's clear that numeric_limits<> has some responsibility to respect the reality of floating point hardware. The proposal that I thought was in the issues
list was a variation on Matt's (2): has, doesn't have, or might have; there was an accompanying function to report the current state, in the last case.

As I did not participate in most of the numerics issues discussions, I don't know whether this issue was simply closed with no action.

Item 28. Missing public in ostreambuf_iterator (24.5.4)

Proposed resolution:

In [lib.ostreambuf.iterator] 24.5.4 Template class ostreambuf_iterator change:

```
class ostreambuf_iterator :
  iterator<output_iterator_tag,void,void,void,void>{
```

to:

```
class ostreambuf_iterator :
  public iterator<output_iterator_tag,void,void,void,void>{
```

Item 29. basic_string::reserve() unclear

The current description of reserve() is causing confusion.

Proposed resolution:

To the description of reserve() in [lib.string.capacity] 21.3.3 basic_string capacity, add:

> Notes: Calling reserve() with a res_arg argument less than capacity() is in effect a non-binding shrink request. A call with res_arg <= size() is in effect a non-binding shrink-to-fit request.

Item 30. num_get::get const?

22.2.2.1 num_get<charT, InIt>::get is overloaded on void *&, not const void *&. It is now not directly usable by 27.6.1.2.2 operator>>(const void *), since it was changed July '97 from void *.

Proposed resolution (choose one):

1) Change the signature of num_get::get to const void *&.

2) Change the extractor signature back to void *&.

Item 31. Containers overspecify size_type and difference_type

Containers generally overspecify the defined member types size_type and difference_type. These are not necessarily synonyms for the corresponding types in the Allocator parameter type. size_type and difference_type relate to the differences between iterators, which often designate objects of some type X other than the (parameter) type T of the values stored in the container. The relevant allocator types are thus members of Allocator::rebind<X>, not Allocator. (Note that Allocator should be the same as Allocator::rebind<T>.) These types should be implementation defined, just like the iterator types.

Proposed resolution:

> For all containers, change size_type and difference_type typedefs to implementation defined.

Item 32. vector<bool> pointer and const_pointer problem

23.2.5 vector<bool> requires the member type pointer to be a synonym for Allocator::pointer, and the member type const_pointer to be a synonym for Allocator::const_pointer. Both should be implementation defined. vector<bool> uses special iterators to access its elements, rendering these conventional pointers inappropriate and useless.

Proposed resolution:

> Change 23.2.5 vector<bool> pointer and const_pointer typedefs to implementation defined.

Item 33. Input iterator assignment return type

24.1 Input iterator assignment is required to return type T, not a reference T& as for all other iterators besides output iterators. The only two input iterators defined by the library, istream_iterator and istreambuf_iterator, have default copy constructors, which return T&.

Proposed resolution:

> Change the table to an input iterator assignment return type of T.

Item 34. Input iterator operator return type*

24.1 Input iterator operator* is required to return type T, not a reference T& as for all other iterators besides output iterators. istream_iterator returns const T&, however, and istreambuf_iterator should be permitted to do so.

Proposed resolution:

> Change the table to an input iterator operator* return type of ‘‘convertible to T.”

Item 35. Input iterator operator-> missing return type

24.1 Input iterator operator-> has no required return type.

Proposed resolution:

> Change the table to an input iterator operator-> return type of const U&.

Item 36. Template class fpos<class stateT> ambiguous constructors

27.4.3 Template class fpos<class stateT> requires constructors that can be ambiguous:

```
fpos(stateT);
fpos(streamoff);
```

The first endeavors to initialize the stored stateT, while the latter determines the stored offset. Typically, stateT is mbstate_t, which is an integer type on many Unix systems. It can be exactly the same type as streamoff, leading to an unavoidable conflict in the specification, or it can be slightly different, leading to error-prone code.

The fix is to eliminate the first constructor. The class already defines the member functions:

```
stateT state() const;
state(stateT);
```

for setting and reading the stored stateT object. The draft also permits the addition of other, more useful, constructors, such as:

```
fpos(stateT, fpos_t);
```

for implementations that need them, so there is little need for a constructor that sets the stored stateT object.

Proposed resolution:

> From 27.4.3 Template class fpos<class stateT>, remove the constructor fpos(stateT);

Item 37. Typedef event_callback invalid throw()

27.4.2 [lib.ios.base] Class ios_base Typedef event_callback had a throw() qualifier added in London, which is invalid.

Proposed resolution:

To 27.4.2 [lib.ios.base] add words to the effect that:

Functions pointed to by a pointer of type event_callback cannot throw.

Item 38. basic_filebuf::seekpos must call unshift

27.8.1.4 basic_filebuf::seekpos must call unshift for the codecvt facet before it alters the file position indicator. (I was wrong in my opposition to this behavior at the Nashua meeting.--pjp)

Proposed resolution:

To 27.8.1.4 add wording to the effect that:

basic_filebuf::seekpos must call unshift for the codecvt facet before it alters the file position indicator.

Item 39. Where is basic_iostream declared?

[Matt Austern] Which header basic_iostream is declared in? My best guess is <istream>, but it's not included in the <istream> header synopsis.

[Jerry Schwarz] I vaguely remember putting it in istream a long time ago.

[pjp] basic_iostream is described under istream, as it should be, but it is indeed missing from the synopsis for that header. ios_base is described under ios, as it should be, and is present in the synopsis for that header.

[see [lib.iostreamclass] 27.6.1.5 Template class basic_iostream]

Proposed resolution:

To the Header <istream> synopsis in [lib.iostream.format] 27.6 Formatting and manipulators add:

```
template <class charT, class traits = char_traits<charT> >
  class basic_iostream;
typedef basic_iostream<char>     iostream;
typedef basic_iostream<wchar_t>  wiostream;
```

Item 40. Clause 27 Inconsistent with regards to typedefs

Proposed resolution:

27.6.1.1 Add typedefs of char_type, int_type, pos_type, off_type and traits_type to the basic_istream template. The typedefs are already there, but they're inside comments.

27.6.2.1 Add typedefs of char_type, int_type, pos_type, off_type and traits_type to the basic_ostream template. The typedefs are already there, but they're inside comments.

27.7.1 Add typedefs of char_type, int_type, pos_type, off_type and traits_type to the basic_stringbuf template. The typedefs are already there, but they're inside comments.

27.7.2 The template basic_istringstream shows a number of typedefs in commentary lines. These can stay in comments, since they're not actually used in this subclause. No change is required, but the typedefs may be "uncommented" for consistency with the other class templates.

27.7.3 The template basic_ostringstream shows a number of typedefs which are not in commentary lines. These can be put into comments, since they're not actually used in this subclause, or they can be left as is.

27.7.4 The template basic_stringstream shows a number of typedefs which are not in commentary lines. These can be put into comments, since they're not actually used in the subclauses which describe this class template, or they can left as is.

27.8.1.1 Add typedefs of char_type, int_type, pos_type, off_type and traits_type to the basic_filebuf template. The typedefs are already there, but they're inside comments.

27.8.1.5 The template basic_ifstream shows a number of typedefs in commentary lines. These can stay in comments, since they're not actually used in this subclause. No change is required, but the typedefs may be "uncommented" for consistency with the other class templates.

27.8.1.8 The template basic_ofstream shows a number of typedefs in commentary lines. These can stay in comments, since they're not actually used in this subclause. No change is required, but the typedefs may be "uncommented" for consistency with the other class templates.

27.8.1.11 The template basic_fstream shows a number of typedefs in commentary lines. These can stay in comments, since they're not actually used in this subclause. No change is required, but the typedefs may be "uncommented" for consistency with the other class templates.

Item 41. Library uses illegal default args

14.1 paragraph 8 says:

> The set of default template-arguments available for use with a template in a translation unit shall only be provided by the first declaration of the template in that translation unit.

This is especially a problem in <iosfwd>, but may affect other parts of the library as well. For example, the following declaration may be found in <iosfwd>:

```
template <class charT, class traits = char_traits<charT> >
  class basic_ios;
```

The basic_ios template is defined in <ios>:

```
#include<iosfwd>
namespace std {
      /* ... */

      template <class charT, class traits = char_traits<charT> >
        class basic_ios;

      /* ... */
}
```

The definition of basic_ios in <ios> is ill-formed because it restates the default template-arguments which are already provided by the forward declaration in <iosfwd>.

[John Spicer] > There is a good reason for this rule (or a rule like this) for function templates. It gets messy if you permit default arguments to be added after the template has been referenced.

There is not a good reason for the rule for classes. The WP inadvertently got changed to have this rule apply to classes, and we decided not to change it back because we thought the restriction was harmless. The previous rule for classes was the same as the usual rules for nontemplate functions (i.e., that you can't redeclare a default argument but you can add). Presumably, this would fix the library problem as the default argument could be placed on the definition of the class, and not on any of the forward declarations.

[Josee] This looks like an issue that core and library should discuss together in Morristown.

Proposed resolution (assuming core changes the language rules per John Spicer above):

Remove default template arguments from all forward declarations, such as in <iosfwd>.

Item 42. D.7 Description of freeze() is contradictory

In section D.7 "char* streams", the description and use of the "freeze" member function is contradictory, and there is also an inconsistently-implemented change from classic iostreams.

The state variable "frozen" means that the stream buffer is fixed and cannot be changed. Various operations that set or depend on "frozen" assume the "freeze" operation by default sets "frozen", as the name of the function implies. The description of "freeze" says the default operation is to clear "frozen" (inconsistent with its use), and that a "false" argument to "freeze" sets "frozen" (inconsistent with the name of the function). If the description of "freeze" is reversed, it then corresponds to classic iostream behavior, and becomes consistent with its use throughout section D.7. But wait -- there's more.

In classic iostreams, the freeze function had an int parameter. The function now appears in three versions in three subsections of D.7. In D.7.1 it has a bool parameter, but with a default argument value of 1 instead of "true". In D.7.3 and D.7.4 it has an int parameter (with a default value of 1).

By changing the one instance of a bool parameter type back to int, and by reversing the sense of the description of "freeze", the draft will become internally consistent, and will correspond to classic iostreams.

The other possibility is to leave this freeze's parameter type bool, change its default value to "true", and change the other two instances of "freeze" to bool with default "true" as well.

I suggest the changes above as being the minimum needed change, and because it corresponds to classic iostreams. If someone thinks the larger change (making everything bool) is appropriate, I have no objection, particularly if some implementations currently use bool.

[pjp] Microsoft VC++ has defined the parameter to freeze as type bool since V4.2 was released in July 1996. It does so because the committee once voted to change int to bool. I'm away from my office, so I can't cite chapter and verse as is my wont, but I'm comfortable that this is the case.

Thus, I believe the proper fix is to consider this an incomplete edit that needs to be completed to satisfy the stated will of the committee. It also aligns the draft with an existing practice that we encouraged with an earlier vote.

Proposed resolution (choose one):

Option 1:

Change D.7.1 Class strstreambuf from:	void freeze(bool freezefl = 1);
To:	void freeze(int freezefl = 1);
Change D.7.1.2 Member functions from:	void freeze(bool freezefl = 1); ... - If freezefl is false, the function sets frozen in strmode. - Otherwise, it clears frozen in strmode.
To:	void freeze(int freezefl = 1); ... - If freezefl is zero, the function clears frozen in strmode. - Otherwise, it sets frozen in strmode.

Option 2:

Change [depr.strstreambuf] D.7.1 Class strstreambuf from:

```
void freeze(bool freezefl = 1);
```

to:
```
void freeze(bool freezefl = true);
```

Change [depr.strstreambuf.members] D.7.1.2 Member functions from:

```
void freeze(bool freezefl = 1);
```
...
- If freezefl is false, the function sets frozen in strmode.

to:
```
void freeze(bool freezefl = true);
```

...
- If freezefl is true, the function sets frozen in strmode.

Change [depr.ostrstream] D.7.3 Class ostrstream from:

```
        void freeze(int freezefl = 1);
```

to:

```
        void freeze(bool freezefl = true);
```

Change [depr.ostrstream.members] D.7.3.2 Member functions from:

```
        void freeze(int freezefl = 1);
```

to:

```
        void freeze(bool freezefl = true);
```

Change [depr.strstream] D.7.4 Class strstream from:

```
        void freeze(int freezefl = 1);
```

to:

```
        void freeze(bool freezefl = true);
```

Change [depr.strstream.oper] D.7.4.3 strstream operations from:

```
        void freeze(int freezefl = 1);
```

to:

```
        void freeze(bool freezefl = true);
```

Item 43. 20.2.2 pair<> missing copy constructor

20.2.2, paragraph 1.The pair template has the member

template <class U, class V> pair(const pair<U, V> & p);

but does not have a non-template copy constructor. According to footnote 105 in 12.8, the compiler will synthesize a copy constructor for each instantiation of the pair template. Is this what we want or should we declare a copy constructor for pair?

Proposed resolution:

Add a copy constructor to the pair template:

```
        pair(const pair & p);
```

Item 44. Return from first call to std::set_new_handler unclear

[lib.set.new.handler] 18.4.2.3 set_new_handler specifies **Returns:** the previous `new_handler`. No mention is made of any special case for the first call.

Proposed resolution:

Change [lib.set.new.handler] 18.4.2.3 set_new_handler returns specification to:

> 0 on the first call, the previous `new_handler` on subsequent calls.

Delete [lib.new.handler] 18.4.2.2 paragraph 3, which reads:

> **Default behavior:** The implementation’s default `new_handler` throws an exception of type `bad_alloc`.

Item 45. iterator_traits<> underspecification for nested types

Matt Austern writes:

Exactly how should iterator_traits<I>::value_type, iterator_traits<I>::pointer, and iterator_traits<I>::reference be defined when I is a constant iterator?

Here's an example of why this question is important. (It's like accumulate but without the initial value, and it works only with non-empty ranges.)

```
  template <class InputIterator>
  iterator_traits<InputIterator>::value_type
  sum(InputIterator first, InputIterator last) {
    iterator_traits<InputIterator>::value_type result = *first++;
    while (first != last)
      result = result + *first++;
    return result;
  }
```

I claim that this is a perfectly reasonable algorithm. I claim, further, that it's reasonable for InputIterator to be a constant iterator. (It had better be! The Input Iterator requirements don't include modification.)

But now look at what happens if you pass sum() a pair of const int*'s. This will use the pointer specialization of iterator_traits, so iterator_traits<const int*>::value_type is const int, not int. Therefore sum() will fail miserably.

If value_type is to be of any use then it's pretty clear that the value type of a constant pointer to T must be the unadorned T, not a type that's cv-qualified. For other reasons (think about the way that the reverse_iterator template works) it's clear that the pointer and reference types of a constant iterator do, by contrast, have to be const qualified.

There are two distinct issues here.
(1) These requirements on const qualification should be stated somewhere. (Probably in the iterator_traits discussion.)
(2) At present, iterator_traits doesn't satisfy those requirements when it comes to pointers.

I consider both of those problems to be serious. We don't have much time left, though, and if I had to choose one to fix in the standard it would be (2). Problem (1) can be addressed with books and articles, so I don't think it would be disastrous if we waited to fix it until after the standard came out.

Problem (2) is more serious, but fortunately it's very easy to fix: just add a second iterator_traits specialization.

This is a simple and low risk change. We have already done it in the SGI implementation.

[lib.iterator.traits] 24.3.1 Iterator traits currently contains the following:

```
template <class Iterator>
struct iterator_traits {
  typedef typename Iterator::iterator_category iterator_category;
  typedef typename Iterator::value_type        value_type;
  typedef typename Iterator::difference_type   difference_type;
  typedef typename Iterator::pointer           pointer;
  typedef typename Iterator::reference         reference;
};

template <class T>
struct iterator_traits<T*> {
  typedef random_access_iterator_tag iterator_category;
  typedef T                          value_type;
  typedef ptrdiff_t                  difference_type;
  typedef T*                         pointer;
  typedef T&                         reference;
};
```

Proposed resolution:

To [lib.iterator.traits] 24.3.1 Iterator traits add the following specialization:

```
template <class T>
struct iterator_traits<const T*> {
  typedef random_access_iterator_tag iterator_category;
  typedef T                          value_type;
  typedef ptrdiff_t                  difference_type;
  typedef const T*                   pointer;
  typedef const T&                   reference;
};
```

Item 46. Stream iterators missing default argt for charT

Bjarne Stroustrup writes:

I have been using a version of the original STL where the definitions of ostream_iterator is

```
	template <class T>
	class ostream_iterator : public output_iterator {
	protected:
		ostream* stream;
		char* string;
		// ...
	};
```

I now encountered a standards conforming version that looks like this:

```
	template <class T, class Ch, class Tr = char_traits<Ch> >
	class ostream_iterator : public
iterator<output_iterator_tag,void,void,void,void> {
	public:
		typedef Ch char_type;
		typedef Tr traits_type;
		typedef basic_ostream<Ch,Tr> ostream_type;
		// ...
	};
```

and I find that all of my code is broken. For example:

```
	istream_iterator<string> is(cin); // error: too few template arguments
```

It should be:

```
	istream_iterator<string,char> is(cin);
```

This is particularly embarrassing for widely read/distributed tutorial material.

Tough, you might say, but it appears to be needlessly broken by an incomplete conversion to templatized streams. I think the proper definition is:

```
	template <class T, class Ch = char, class Tr = char_traits<Ch> >
	class ostream_iterator
	   : public  iterator<output_iterator_tag,void,void,void,void> {
	public:
		typedef Ch char_type;
		typedef Tr traits_type;
		typedef basic_ostream<Ch,Tr> ostream_type;
		// ...
	};
```

We provided support for the most common case in the form of default template parameters and typedefs in many places. Why not here? An oversight?

Proposed resolution:

Change [lib.iterator.synopsis] 24.2 Header <iterator> synopsis and [lib.ostream.iterator] 24.5.2 Template class ostream_iterator from:

```
template <class T, class charT, class traits = char_traits<charT> >
```

to:

```
template <class T, class charT = char, class traits = char_traits<charT> >
```

Change [lib.iterator.synopsis] 24.2 Header <iterator> synopsis and [lib.istream.iterator] 24.5.1 Template class istream_iterator from:

```
template <class T, class charT, class traits = char_traits<charT>,
class Distance = ptrdiff_t>
```

to:

```
template <class T, class charT = char, class traits = char_traits<charT>,
class Distance = ptrdiff_t>
```

Item 47. For_each overspecified?

Bjarne Stroustrup writes:

for_each() is specified to require that the operation does not change the value of a sequence element? I think that is an overspecification. That requirement is unnecessary for many sequences given an obvious implementation and force people to use transform where the operation must produce a result. Since the requirement is not checked, there must be much code around that violate it in the obvious and reasonable way. Further, where did that requirement come from? Is it simply a semi-accidental over simplification or is there a deeper issue that I am missing?

Item 48. Tighten constraints on signal handler

Jonathan Schilling writes:

In London we added constraints on signal handlers to 18.7 /5 [lib.support.runtime]. The new language says:

> The common subset of the C and C++ languages consists of all declarations, definitions, and expressions which may appear in a well formed C++ program and also in a conforming C program. A POF ("plain old function") is a function which uses only features from this common subset, and which does not directly or indirectly use any function that is not a POF. All signal handlers shall have C linkage. A POF which could be used as a signal handler in a conforming C program does not produce undefined behavior when used as a signal handler in a C++ program. The behavior of any other function used as a signal handler in a C++ program is implementation defined.

I think this language may not go far enough. Wouldn't something like f() in this:

```
#ifndef __cplusplus// perhaps from a header somewhere
int throw;
#endif

int global = 1;

int f() {
        return global ? throw - 1 : 0;          // does a throw in C++
}
```

qualify as a POF, since it can appear in both a well-formed C++ and conforming C program? But of course we don't want to allow f() to be used as a signal handler in C++.

Proposed resolution:

Add to the first sentence of the WP language above:

> [... and also in a conforming C program], and which have the same semantic behavior in both a C++ and C program.

Item 49. 25.2.2 container::swap() and references

Nico Josuttis writes:

A private discussion with Dave, Greg and Nathan has convinced me that it is not necessary to change the description of the (re)allocation policy of vectors in general (some clarifications might help but at this stage they might be more danger than enhancement).

BUT while making this discussion it seems that we have one aspect not specified in the draft that IMHO should be specified: It seems that we don't specify anything about what happens with references, pointers and iterators in general and with the allocated memory in detail when for a container swap() gets called. This probably means that actually all bets are off.

Therefore it probably would be an enhancement to write down what typically does happen. (See proposal below)

With the following supplement for vector (and string?):
>   Does not reallocate memory. The memory used by the two
>   arguments, and their capacities, are swapped. [All references...&c.]
(Thanks, Nathan)

If such a completion makes sense, one interesting question occurs: Should this be specified individually for every standard container or in general in a general section? I guess that making it a requirement would be a problem as it is probably too late and too dangerous (Dave pointed out an example which would break such a requirement). But may be we could/should formulate it as optional sequence operation thus giving a hint that if possible it would be fine to behave like that.

Proposed resolution:

Add to [lib.alg.swap] 25.2.2 Swap, after paragraph 3:

> Notes: After swap(), all references, pointers and iterators to the elements of a swapped container refer to the same elements as they did before swap(). That means that they swap the sequence they refer to accordingly.

Item 50. 27 + appendix D editorial boxes [all relate to I/O]

These boxes should be reviewed and removed.

Item 51. basic_ifstream & basic_ofstream prototype errors

The constructor prototype in 27.8.1.8 is correct:

```
explicit basic_ofstream(const char* s,
                        ios_base::openmode mode =
                        ios_base::out | ios_base::trunc);
```

However, there are several errors in other basic_ofstream prototypes.

Proposed resolution:

In 27.8.1.8 [lib.ofstream] change:

```
void open(const char* s,ios_base::openmode mode = out | trunc);
```

to:

```
void open(const char* s,ios_base::openmode mode =
                        ios_base::out | ios_base::trunc);
```

In 27.8.1.9 change paragraph 1:

```
        explicit basic_ofstream(const char* s, openmode mode = out);
```
to:
```
        explicit basic_ofstream(const char* s, openmode mode =
                                   ios_base::out | ios_base::trunc);
```

In 27.8.1.10 change paragraph 2:

```
        void open(const char* s, openmode mode = out);
```
to:
```
        void open(const char* s, openmode mode =
                                   ios_base::out | ios_base::trunc);
```

In 27.8.1.5 [lib.ifstream] change:

```
        void open(const char* s, openmode mode=in);
```
to:
```
        void open(const char* s, openmode mode=ios_base::in);
```

In 27.8.1.6 change paragraph 1:

```
        explicit basic_ifstream(const char* s, openmode mode = in);
```
to:
```
        explicit basic_ifstream(const char* s, openmode mode = ios_base::in);
```

In 27.8.1.7 change paragraph 2:

```
        void open(const char* s, openmode mode = in);
```
to:
```
        void open(const char* s, openmode mode = ios_base::in);
```

Item 52. cin doesn't have dec bit set

Bjarne Stroustrup in Message c++std-lib-6386:

Reading using

```
        int i;
        cin >> i;
```

using default settings, can I read hexadecimal and octal? That is,

        99

is decimal and reads 99. Can I do

        011

and get 9?

        0x1f

and get 31?

My suggested answer is 'yes' and that agrees with the couple of implementations, I have tried.

Jerry Schwarz comments:

Oh dear. I investigated a little further and what's happening in iostream classs is that cin is initialized in a different place from other streams (because of the withassign business) and it doesn't have the dec bit set in flags. I guess this is another bug that became a feature.

At this point I'm certainly in favor of making the standard agree with existing practice.

Item 53. Core vs library editorial conflicts

Josee Lajoie reports on CD2 comment 23/10:

The text in the library section 18.4 is incorrect and needs to be changed to match the rules for placement delete in 5.3.4. I view this as editorial.

Josee Lajoie reports on CD2 comment 23/11:

I'll also take this. This is a comment on the function unexpected. Here again, the text in the library section 18.6.2.2 must be changed to match the rules in 15.5.2. I also view this as editorial.

Item 54. Invalid vector<bool> partial specialization

23.2.5 vector<bool> partial specialization has a default template argument, which is disallowed (and unnecessary).

Proposed resolution:

In 23.2.5 change:

```
template <class Allocator> class vector<bool, Allocator> {
```

to:

```
template <class Allocator> class vector<bool> {
```

[End of Morristown Issues List]