Doc No: X3J16/96-0185 WG21/N1003
Date: October 23, 1996
Project: Programming Language C++
Ref Doc:
Reply to: TC X3J16

****

Responses to Official Public Review Comments

****

1- Comment from Daniel Louis Miller / DSC Communications
Received a hardcopy only.
Address: DSC Communications Corp
P.O. Box 796367
Dallas, TC 75379-6367

Oddity 1:
The switch statement:

```
switch (condition) statement
```

'statement' should be a variant of compound statement.

-> Rejected.
-> Forcing switch to use {} breaks C compatibility.

Oddity 2:
Change the syntax for try and catch blocks to be:

```
try-block:
  try statement handler-seq
handler:
  catch (exception-declaration) statement
```

-> Rejected.
-> The C++ scoping mechanism uses braces; it was a deliberate choice
-> to have try and catch blocks use braces; braces are needed with try
-> blocks to properly associate a catch block with its associated try
-> block (this solves the same problem as the dangling else problem
-> with the if statement).

2- Comment from Stephen Bard
Address:
Mr. Stephen Bard
Microsoft Corp.
One Microsoft Way
Redmond, WA 98052-6399

Add an additional 'clean' clause to the try/catch mechanism.

-> Rejected, request for an extension

3- Comment from Bryant Harris / Edge Research
Address:
Bryant Harris
Edge Research
1 Harbour Place Suite # 553
Portsmouth, NH 03801

Add a mechanism to C++ for implied member function calls.

-> Rejected, request for an extension

4- Comment from Peter Durham / Microsoft
Received by email
email address: peterdur@microsoft.com

Add a deferred assignment operator.

-> Rejected, request for an extension

5- Comment from Allen B. Taylor
Received by email
email address: allen.taylor@prior.ca or ataylor@spar.ca

5.1 delete and arrays

The syntax "delete x" should recognize whether or not it is deleting an array and call operator delete() or operator delete[]() appropriately.

```
->   Rejected.
->   The committee suggests that users who want only one way of writing
->   new and delete expressions use the following coding practice:
->     T* p = new T[1];
->     delete[] p;
```

5.2 Global delete should zero its pointer argument

The global delete operator should zero the pointer passed as an argument, thus allowing future attempts to delete via that pointer benign. Note that this would change the function declarations of the delete operators from void operator delete(void *) to void operator delete(void *&) and from void operator delete[](void *) to void operator delete[](void *&).

->   Rejected.
->   This breaks existing code.

5.3 Meaning of for statement

->   Accepted: see 6.5.3[stmt.for]

5.4 Arbitrary precision type or Binary Coded Decimal Type

->   Rejected, request for extension

5.5 Thread class

->   Rejected, request for extension

5.6 Renew operator

->   Rejected, request for extension

6- Comment from John Mulhern / Siemens Corporation
Received by email
email address: jmulhern@empros.com

[lib.basic.string]

Rather than pointing out each syntax error in the declaration of class basic_string, I would point out the general error thoughout this section. Except where 'basic_string' is the name of a constructor or destructor, 'basic_string' must be modified to: 'basic_string< charT, traits, Allocator >'. This error occurs again in sections: [lib.string::assign] (the last function), [lib.string::assign] (the last function), [lib.string::remove] (the last two functions), and in numerous places througout the text of [lib.string::replace] Any decent word processor can find all the occurances of this error.

-> Accepted.

[lib.string.cons] (and lots of other places...)

The very last line of the section: There is no basic_string(charT) constructor. Unless it was the commitee's intent have such a constructor, this leads to errors throughout the rest of [lib.strings]. I noticed this error in:

```
    [lib.string.cons] last line at end '( c )' s/b '( 1, c )'
    [lib.string.op+=] last line at end '( c )' s/b '( 1, c )'
    [lib.string::append] append( size_type, charT ) arguments to the
                return value are given backwards: '( c, n )' s/b '( n, c )'.
    [lib.string::assign] assign( size_type, charT ) arguments to the
                return value are given backwards: '( c, n )' s/b '( n, c )'.
    [lib.string::insert] insert( size_type, size_type, charT ) arguments to
                the return value are given backwards: '( c, n )' s/b '( n, c )'.
    [lib.string::replace] replace( size_type, size_type, charT ) arguments to
                the return value are given backwards:'( c, n )' s/b '( n, c )'.
    [lib.string::find] last line: '( c )' s/b '( 1, c )'.
    [lib.string::rfind] last line: '( c, n )' s/b '( 1, c )'.
    [lib.string::find.first.of] last line: '( c )' s/b '( 1, c )'.
    [lib.string::find.last.of] last line: '( c )' s/b '( 1, c )'.
    [lib.string::find.first.not.of] last line: '( c )' s/b '( 1, c )'.
    [lib.string::find.last.not.of] last line: '( c )' s/b '( 1, c )'.
    [lib.string::op+] operator+( charT, basic_string<...>& ) in the
                'Returns:' line, the constructor argument must be '( 1, lhs )'
    [lib.string::op+] operator+( basic_string<...>&, charT ) in the
                'Returns:' line, the constructor argument must be '( 1, rhs )'
-> Accepted.

  [lib.string::replace]
  In the 'Effects:' section for the first replace() function, in the
  first sentence, remove the '&' from in front of the name 'pos1'.
-> Accepted.

  [lib.string::compare]
  The first compare() function in this section must be declared 'const'
  as it was declared in [lib.basic.string].
-> Accepted.

  [lib.string.cons]
  For explicit basic_string( Allocator& Allocator() ), Table 38, it
  seems to me that the required value for data() should be '0' because
  the size() == 0, following the requirements given in section
  [lib.string.ops].  The capacity should, however, be left unspecified.
  I can not think of any circumstance in which data() would be other
  than zero for a string of length zero.  On the other hand, I can well
  imagine code expecting a zero-pointer from data() when the string
  size() is zero.  c_str() returns a traits::eos() terminated, zero
  length string for a string of size() == 0.  The standard should be
  more clear that that is the case for c_str() since this is what
  programmers will expect and indeed need.
-> Rejected.
-> The semantics are required for proper integration with STL.

  [lib.basic.string]
     size_type copy( charT*, size_type, size_type )
  This function is not declared to be 'const', but the function is
  indeed 'const' with respect to *this.  As the copy() function might be
  especially useful with const strings, I believe the copy() function
  should be declared const.
-> Accepted.

  [lib.string.op+]
  In the 'Returns' section for the first function, i.e. 'Returns:
  lhs.append( rhs )', surely the committee doesn't intend what is
  written there.  It should be something like 'Returns:
  basic_string<...>( lhs, rhs );' but of course the concatenating
  constructor is not part of the basic_string public interface.  It may
  well be part of the private interface.
-> Accepted.

  [lib.string::find]
  Missing a comma between ( s, n ) and pos on the 'Returns' line for
```

```
size_type find( const charT*, size_type, size_type ) const;
```

-> Accepted.

[lib.string::rfind]
The default value for the ’pos’ argument should be ’0’ and not the stated ’npos’. This applies to the 1st, 3rd and 4th versions of rfind as presented. ’pos’ refers to the offset into *this from the beginning. rfind() searches from its last character to at(pos) just as find() searches from at(pos) to the last character.
In the ’Effects’ section, the first condition must be identical to find()’s first condition, i.e. ’pos <= xpos and xpos + str.size() <= size()’

-> Rejected. The stated semantics of rfind are those desired.

[lib.string::find.last.of]
The default value for the ’pos’ argument should be ’0’ and not the stated ’npos’. This applies to the 1st, 3rd and 4th versions of find_last_of as presented. ’pos’ refers to the offset into *this from the beginning. find_last_of() searches from its last character to at(pos) just as find_first_of() searches from at(pos) to the last character. In the ’Effects’ section, the first condition must be identical to find_first_of()’s first condition, i.e. ’pos <= xpos and xpos < size()’.

-> Rejected. The stated semantics of find_last_of are those desired.

[lib.string::find.last.not.of]
The default value for the ’pos’ argument should be ’0’ and not the stated ’npos’. This applies to the 1st, 3rd and 4th versions of find_last_not_of() as presented. ’pos’ refers to the offset into *this from the beginning. find_last_not_of() searches from its last character to at(pos) just as find_first_not_of() searches from at(pos) to the last character.

-> Rejected. The stated semantics of find_last_not_of are those desired.

In the ’Effects’ section, the first condition must be identical to find_first_not_of()’s first condition, i.e. ’pos <= xpos and xpos < size()’.

-> Rejected. The condition is stated correctly for the desired semantics.

[Section 21.1.1.10.8] (which bears no other identifier...)
operator>>(): It seems to me that, to be useful, operator>>() must eat zero or more delimiters specified by basic_string<...>::traits::is_del() prior to reading each string. This should be specifed in the standard, to prevent varying implementations. If that is not the committee’s intent, it should be explicitly stated in the standard what the intent is.

-> Accepted.

[lib.string::remove]
This is the only user experience I have to date concerning my implementation of <string> (which I’m still testing). My compiler, xlC, and many others, has trouble with resolving overloading of the calls

```
remove( 6 );
remove( iterator );
```

because iterators for basic_string<char> are of type char* and char*/int overloading is unresolvable. Now, remove(6) calls remove(size_type, size_type) but xlC throws up on the char*/int overload and so never finds remove(size_type, size_type). I’m not sure what you could do to remedy that situation. It is nice to say remove() to erase the entire string. Perhaps

```
basic_string<....>& remove(); and
basic_string<....>& remove( size_type, size_type ) with no defaults given.
```

Perhaps xlC is misbehaving and this isn’t a problem. Perhaps this problem exists elsewhere and I haven’t encountered it yet in user experience.

-> Not a problem.

[lib.string::insert]
iterator insert(iterator p, size_type n, charT c = charT()); There is no ’Returns’ line for this function. Presumably, this should be ’Returns: p’.

-> Rejected. The correct return value is void.

[lib.string.cons]
Nit picking. The template constructor:

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

Compilers will probably like this better if the argument names are ’first’ and ’last’ rather than ’begin’ and ’end’. This would also be consistent with usage everywhere else in the standard with regard to iterators. As a side benefit, the contents of box labelled Table 43 would then make consistent sense. The ’Notes:’ section, needless to say, doesn’t make any sense as printed in the draft standard.

-> Editorial.

[lib.string::find]
[lib.string::rfind]
[lib.string::find.first.of]
[lib.string::find.last.of]
[lib.string::find.first.not.of]
[lib.string::find.last.not.of]
For all of these functions there should some comment in the standard which says that ’pos is the minimum of pos and size()’ thereby dealing with the otherwise unconstrained argument pos. These functions do not throw exceptions for pos > size().

[lib.exception]
Yes, this not part of [lib.strings] but I had to implement <stdexcept> in order to implement <string>. The copy constructor cannot return any value, so ’exception& exception(const exception&) throw();’ should be ’exception(const exception&) throw();’.

-> Rejected. The current semantics are those intended by the LWG.
-> If pos >= size(), then nothing is found.

7- Comment from Marc Shepherd
Received by email
email address: mshepherd@mhfl.sbi.com

1 General Comments

In general, I find the text to be of an inferior quality to the ISO C Standard. The C Standard’s clear division between syntax, semantics, and examples, is lacking. Notes and examples are mingled with normative text and are not always clearly delineated. There has obviously been an attempt to relegate non-normative comments to "notes," but the document is very uneven in this regard. I have pointed out some of the inconsistencies, but the entire text needs a thorough going-over, especially the Library clauses. Also, the notation used to call out non-normative text (using square brackets) is particularly ugly.

-> Rejected.
-> The committee prefers that the style of the C++ Standard be closer
-> to the style used in the ARM [C++ Annotated Reference Manual, by
-> Margaret Ellis and Bjarne Stroustrup], manual with which many C++

-> programmers are familiar. Also, the division used in the C Standard
-> has its drawbacks: the constraints and the semantics rules are
-> described separately causing some text to be duplicated in both
-> places.

The Committee seems undecided between the words "implementation" and "processor." I prefer "implementation," since it's what the C Standard uses, and I've flagged occurrences of the later as errors.

-> Editorial.

The text is cavalier in its use of the word "shall" - often, "shall" conveys a requirement on the implementation but sometimes, it conveys a requirement on the programmer. I have pointed out a few such inconsistencies, but the entire document needs a thorough review.

-> Editorial.

A non-normative annex similar to Annex G of the ISO C Standard, which summarizes all the kinds of unspecified, undefined and implementation-defined behaviors, would be helpful.

-> Editorial.

2 Namespaces

Lack of Prior Art.

Complexity.

Pervasiveness.

Syntactic Confusion.

Namespaces and classes share use of the :: operator for scope resolution. Yet, the operator works very differently for namespaces and classes even in cases that have an appearance of similarity.

Lack of Balance between the Problem and the Solution.

High Probability of Error.

Durability of Standards.

-> These comments were considered before namespaces were adopted. The
-> committee decided that the benefits of namespaces outweigh the
-> concerns described here.
-> Re, Syntactic confusion:
-> The committee has considered this comment in greater details and
-> modified the name look up rules for namespace members after the '::'
-> operator. See 3.4.2.2 [namespace.qual].

3 New-Style Cast Operators

static_cast

Since static_cast is a new concept, there can be no issues of backward-compatibility. Now is the time to define it sensibly. I would recommend farther restricting static_cast as follows:

1. A static_cast between two integral types is well-formed iff all possible values of the source type can be represented in the destination type.

   For example, on a machine with 16-bit shorts and 32-bit ints and longs, a static_cast from a long to an int would be well-formed, but a static_cast from an int to a short would be not (since, on such a machine, not all ints have a well-behaved conversion to short.)

2. A static_cast between two floating point types is well-formed iff all possible values of the source type have a well-defined conversion to the destination type (as described in 4.8). The intent is to prohibit use of static_cast where there is a

possibility of undefined behavior. Because this is a compile-time check, a conversion that is possibly undefined should be disallowed as a static_cast.

3. A static_cast from a floating point type to an integral type is well-formed iff all possible values of the source type can be represented (after truncation) in the destination type. The intent, again, is to disallow a conversion that could be undefined (as described in 4.9).

4. A static_cast from an integral type to a floating point type is always well-formed.

5. A static_cast between an integral type and bool, in either direction, is always well-formed.

6. A static_cast from an integral type to an enumeration type is always ill-formed. (This follows the pattern of disallowing conversions that might result in an undefined/unspecified value.)

7. A static_cast from an enumeration type to an integral type is well-formed iff the destination type is large enough to hold all possible values of the enumeration.

8. A static_cast from bool to a floating point type is always well-formed.

9. A static_cast from a floating point type to bool is always ill-formed. (It is ill-formed, because such a conversion requires an exact comparison of the floating point value to zero. And, as all programmers should know, the result of a floating point operation should never be compared to an exact value, because of the vagaries of round-off errors.)

These restrictions allow static_cast to live up to its billing as a vehicle for well-behaved casts only.

-> Rejected, request for an extension.
-> The only other choice for these conversions is reinterpret_cast.
-> That would be a bad choice since reinterpret_cast is reserved
-> for extremely dangerous operations; the intent is that you should
-> be willing to grep for all instances of reinterpret_cast in a
-> program and know that each instance is one which really must be
-> thoroughly understood. Since the above conversions are so common,
-> requiring reinterpret_cast for them would ruin the careful
-> distinction between common and highly unusual casts.

dynamic_cast

Given the farther restrictions to static_cast described above, there is a natural and important extension to dynamic_cast. I would recommend permitting enumeration and built-in numeric types (and references to them) to be the destination type of a dynamic_cast.

-> Rejected, request for an extension.

4 C Linkage

Subclause 7.5 says that "Every implementation shall provide for linkage to functions written in the C programming language." I believe this requirement is short-sighted. It is true that, today, C and C++ coexist on most platforms, but there is no assurance that this will always be true. A vendor implementing C++ on a platform that has no existing C support should not be obligated to provide such support. Naturally, many vendors will choose to do so, but this should be a "quality of implementation" issue, not a normative requirement.

I would revise 7.5 to say:

1. Whether an implementation provides C linkage is implementation-defined.

2. If an implementation does provide C linkage, the string-literal in the linkage-specification shall be spelled "C", and the semantic rules specified in 7.5 must be observed.

-> Rejected.
-> All conforming C++ implementations are required to accept a
-> declaration of the form 'extern "C" declaration' The meaning of
-> such a declaration is implementation-defined, i.e. an implementation
-> is free to ignore it.

5 C Linkage (Library)

Implementations can leave out Standard C Library components that have fully-functional C++ replacements: stdio.h and locale.h

-> Rejected, breaks C compatibility.

6 Translation Limits

-> Rejected.
-> The Annex B on translation limits has been considered very
-> carefully by the committee and is the best compromise that was
-> acceptable to the majority of committee members.

Library

7 Library - Is it too much?

-> Rejected.
-> The committee believes that the functionality provided by the library
-> is necessary in the first version of the C++ standard.

8 Exception Specifications

The use of exception specifications in the Library clauses is incomplete and inconsistent. It is too soon to tell how extensively C++ exceptions will be used, but I believe a user would want an iron-clad promise about what exceptions a library function might throw. Therefore, unless there is a good reason to the contrary, every Library function (except those inherited from C) should have an exception specification.

-> Rejected.
-> The exception specification policy for the library has been
-> discussed at length at several meetings. The current policy is the
-> one the majority of the committee members prefers.

9 Granularity

"Granularity" means: do the headers contain too many or too few declarations? Headers that contain too many declarations increase compile times unnecessarily. But, headers that contain too few declarations confuse the programmer, and increase the number of #includes that have to be coded to produce a working program.

For the most part, I think the Committee got the granularity decisions right, with the following exceptions:

- <utility>: This header contains two things: the comparison operator templates, and the pair/make_pair templates. These seem unrelated. Also, some programmers may want pair, but will not want the library to define comparison operators on their behalf. I would advise splitting <utility> into two headers: <pair> and <comparison>.

- o <memory>: This header contains three things: the default allocator, various raw memory operations, and the auto_ptr template. auto_ptr does not seem to belong here (and has a broad applicability that goes beyond the default allocator and memory operations). At a bare minimum, auto_ptr should be in its own header. Splitting the default allocator into its own header would also be clearer.

- o <iterator>: I suspect the stream iterators will be less-often used than the other iterator types. Yet, declaring the stream iterators in the same header requires <iterator> to include <ios>, <iosfwd> and <streambuf>. Removing the stream iterators into a new header - say, <ioiterator> will substantially improve the compile time of many programs.

- o <algorithm>: This header is far too large: to get one algorithm, the user is required to compile all of them. Clause 25 suggests a natural division into: non-modifying sequence algorithms; mutating sequence algorithms; sorting and searching. At least this much separation should be provided.

-> Major changes in header content were rejected.
->
-> The stream iterators question was rejected.
->
-> The <utility> comparison operators question has been resolved
-> by placing these operators in a sub-namespace.

10 Container Concerns

1. Hash structures should be added to the Library. I understand the Committee rejected hash structures at a recent meeting-not out of any technical deficiency, but out of a desire to Get the Standard Out. The reality, however, with two sound implementations widely available by anonymous ftp, is that hash structures are likely to be incorporated in all the commercial STL implementations anyway. The Committee might as well accept this, and add them to the Standard.

-> Rejected.
-> The committee does not wish to increase the size of the library at
-> this point in the standard process.

2. Container elements need not be ordered. Table 50 ("Container Requirements") says that <, <=, > and >= operations must be defined for all objects placed into a container. This requirement is too strict. Ordering should be required only if the user instantiates an algorithm that needs it.

-> Rejected.
-> The comment is not correct. The requirements in Table 50
-> ("Container Requirements") specify requirements on the container
-> itself, *not* the elements of the container.

3. Valarray should meet the requirements of a Container. I have already stated that the valarray (and related classes) are not of sufficient utility to belong in this Standard. But, if valarray remains in the Standard, it should at least meet the requirements of a Container,(by providing begin and end methods, for example). As it now stands, valarray looks like it came out of a different Committee.

-> Rejected.
-> Note that a certain amount of cleanup of valarray has been done.

11 Iostream Concerns

The description of the input/output classes appears to be in a state of flux. In particular, there is a confusion about the division of duties between the ios_base and basic_ios classes. (It’s bound to confuse people when the Standard contains two classes with such similar names.) Since I believe the Committee is aware of the numerous errors, I will not comment any farther.

-> Accepted.
-> The committee believes that the division of duties between these two
-> classes has been improved, and makes sense now.
-> The non template class ios_base is in charge of the data, and
-> functions that does not require the charT and traits template
-> parameters. The class basic_ios is in charge of the functions and
-> data, that make use of the charT and traits template parameters.

A larger concern is the elimination of the bi-directional I/O classes (e.g. iostream, fstream). These have been a part of C++ implementations for many years now; removing them indicates a disrespect for existing code. I was especially astonished to read (in comp.std.c++) that these classes were removed so that implementors would not be compelled to use multiple inheritance in the Standard Library. Imagine that: a Committee afraid of its own language!

-> Accepted.
-> Bi-directional streams were reinstated in the library.

Annex D confuses me. It is labeled "normative," but in a couple of places it says that an implementation may provide certain functions. If implementations are required to provide these functions, they should be described in Clause 27 with the rest of the I/O library (with deprecated features labeled as such). Annex D should be just a summary, not the complete description, of such features.

-> Editorial.

12 Clause-by-clause Comments

Clause 1.

Definitions (1.3).

o Implementation-defined behavior. The text says that "the range of possible behaviors is delineated by the standard." In general, this is not so. The sentence should be stricken.
-> Accepted.

o Implementation limits. Since the current draft prescribes no such limits, this definition is unnecessary. (However, in my opinion, implementations should be prescribed - see above.)
-> Rejected.

o Multibyte character. This definition uses the terms "extended character set" "and "basic character set," neither of which is defined.
-> Character discussion in clause 2 is now extensively revised. We
-> followed precedent of the C standard in defining "basic source
-> character set" precisely, then describing, rather than defining,
-> "basic execution character set".

o Implementation. This word is not defined, but should be.

-> Editorial.

o POD, PODS. These two terms appear frequently in the document, but are not defined until a footnote on the first page of Clause 9. They should be defined here.

-> Editorial.

Processor compliance (1.7).

This sub-clause should be called "Implementation Compliance," and the word "processor" changed to "implementation" throughout. Paragraph 4 uses the term "resource limits," which is never defined. Paragraph 2 uses the term "diagnosable errors," which is never defined. Paragraph 5 states that the notes, footnotes and non-normative annexes are not part of the normative Standard. While I agree with this, the last paragraph of a section called "Processor Compliance" is"not the right place to state it.

-> Editorial.

Program execution (1.8).

The first paragraph uses the words "semantic descriptions," but the document is not structured in such a way that "semantic descriptions" (as opposed to other kinds of descriptions) are easily identified. The word "processor(s)" appears several times in this sub-clause. Paragraph 3 says that, in each case of unspecified behavior, the Standard defines a set of allowable behaviors; I am not sure that it actually does this in every case.

-> Editorial.

Clause 2.

Phases of translation (2.1).

This sub-clause contains three misuses of the word "shall":

o "A source file that is not empty shall end in a new-line character, which shall not be immediately preceded by a backslash character."

o "A source file shall not end in a partial preprocessing token or partial"comment."

These would be better rewritten as:

o "If a source file does not end in a new-line character, or ends in a new-line character preceded by a backslash character, the translation unit is ill-formed."

o "If a source file ends in a partial preprocessing token or partial comment, the translation unit is ill-formed."

-> Editorial.

Preprocessing token (2.3).

The grammar summary at the beginning of this section mentions the non-terminal symbols header-name and preprocessing-op-or-punc, but never defines them. (The descriptions from the ISO C Standard would suffice.)

-> Editorial.
-> "preprocessing-op-or-punc" is described in 2.9 [lex.key].
-> header-name needs to be defined.

Keywords (2.8).

Paragraph 3 presents rules about identifiers, not keywords, and

should be moved to the preceding sub-clause. Paragraph 4 begins a discussion of a new topic - preprocessing tokens - and should thus begin a new sub-clause. Paragraph 5 presents possibly useful information, but seems to have been dropped in by mistake (having nothing to do with the subject of "keywords").
-> Editorial.

Character literals (2.9.2).

Paragraph 1 uses the phrase "machine’s character set," which is not defined. The table of escape sequences should be accompanied by the more rigorous definitions provided in Clause 5.2.2 ("Character display semantics") in ISO C.
-> Accepted.
-> This has been replaced by "execution character set".

Paragraph 4 says that an octal escape sequence consists of a backslash followed by one or more octal digits. ISO C limits the escape sequence to no more than three octal digits. I can see no justification for such a difference between the two languages.
-> Accepted: See 2.10.2 [lex.ccon].

Paragraph 4 also says that "the value of a character literal is implementation-defined if it exceeds that of the largest [emphasis added] char (for ordinary literals) or wchar_t (for wide literals)." Since char objects may be represented as signed quantities, the use of the word "largest" would seem to be an error, since it fails to consider negative numbers. Perhaps a better wording would be something like this: "The value of a character-literal is implementation defined if it falls outside the implementation-defined range of char objects (for ordinary literals) or wchar_t (for wide literals)."
-> Editorial.

String literals (2.9.4).

The equivalent section in the ISO C Standard is considerably more rigorous. Unless there is a good reason for the two to be different, I would recommend copying the ISO C wording.
-> Editorial.

Clause 3.

One definition rule (3.2).

This subclause appears to be incomplete. In particular, there is no definition of what the rule means for templates or inline functions.
-> Accepted: See 3.2 [basic.odr].

Explicit qualification (3.3.7).

The entire subclause is a non-normative note. Either the subclause should be deleted, or it should contain normative content.
-> Editorial.

Elaborated type specifier (3.3.8).

The organization of this subclause needs to be improved: the first sentence exactly repeats the first sentence of the second paragraph of 3.3.6. A note at the end of this subclause says that the scope of class names introduced in elaborated-type-specifiers is described in 7.1.5.3. The description should be moved here.
-> Editorial.

Point of declaration (3.3.9).

This subclause begins to discuss "point of declaration," but ends with a note containing forward references for enumerators, friends, and elaborated type specifiers. The related material should be brought into one place.
-> Editorial.

Main function (3.6.1).

The description uses the phrase "null-terminated multibyte strings (NTMBSs)," but this is nowhere defined.
-> Now defined, in 17.2.2.1.3.2 [lib.multibyte.strings].

A note merely recommends that additional arguments be added after argv. I think this should be a requirement.
-> Rejected.

Paragraph 3 misuses the word "shall." Here is a proposed rewording:

> A program that calls or takes the address of the main() function, or declares it static or inline, is ill-formed.

-> Editorial.

Static storage duration (3.7.1).

The statement that "the storage for these objects can last for the entire duration of the program" is poorly phrased. Why do you say use the word "can?" What other possibility is there? Also, the phrase "entire duration" is redundant.
-> Editorial.

Types (3.9).

In the first sentence, change "processors" to "implementations."
-> Editorial.

Type names (3.9.4).

This subclause contains no non-normative content. It should be given normative content or deleted.
-> Editorial.

Clause 4.

This clause begins with a non-normative "note." I think this is poor style. Notes should comment on what has already been said, and therefore should not appear at the beginning of a clause, where nothing has yet been said.
-> Editorial.

Clause 5.

Footnote 36 says that "the left operand of += shall not have type bool. This appears to be a normative requirement, so why is it buried in a footnote?
-> Editorial.

Paragraph 7 says, "A reference can be thought of as a name of an object." This appears to be a non-normative comment, and should be a "note."
-> Editorial.

Paragraph 10 contains the words, standard conversion are applied. Pluralize "conversion" or change "are" to "is."
-> Editorial.

In Paragraph 12, change "processor" to "implementation."
-> Editorial.

Primary expressions (5.1).

Paragraph 6 says that "A id-expression is a restricted form of a primary-expression that can appear after . and -> (5.2.4)." This is misleading -- an id-expression can appear in many other contexts, as well. Also, the sentence should begin with "An," not "A."
-> Editorial.

Function call (5.2.2).

The note in paragraph 8 is ambiguous. It refers to "the rules above," but there are so many rules that it is difficult to infer what rules are meant.
-> Editorial.

Class member access (5.2.4).

Paragraphs 3 and 4 both use the term "context in which the entire postfix-expression occurs." The word "context" is used in this way throughout the document, but is never defined. A note in Paragraph 3 uses the word "injection" as if the reader will know what is meant. However, this appears to be its first appearance, and I don’t recall seeing a definition.
-> 3.4 [basic.lookup] paragraph 2 defines what it means for a name
-> to be ’looked up in the context of an expression’.
-> The note now refers to clause 9, where paragraph 2 describes how a
-> class name is inserted into its own scope.

Increment and decrement (5.2.5).

A non-normative note says that incrementing an object of type bool is deprecated. This should be a normative statement, and a list of all such deprecated behaviors should be provided in an Annex.
-> Editorial.

Delete (5.3.5).

In the syntax for delete-expression, delete, [and] are separated by spaces, suggesting that they are distinct tokens, but Clause 2 says that delete [] is a single token.
-> Rejected.
-> Clause 2 was in error (and has been corrected). delete, [and] are
-> separated tokens.

Relational operators (5.9).

This subclause begins with a non-normative note, which is poor style. Also, the statement in the note that the relational operators group left-to-right seems to be important, and perhaps doesn’t belong in a "note" at all.
-> Editorial.

The last dash under paragraph 2 says that "Other pointer comparisons are implementation-defined." Do we really want to force implementations to define such? Perhaps this should be "undefined."
-> The committee voted to have these conversions be "unspecified".

Clause 7.

Paragraph 7 says:

> Only in function-definitions (8.4) and in function declarations

for constructors, destructors, and type conversions can the decl-specifier-seq be omitted.

However, Annex C says that "implicit int" declarations are no longer permitted. So, it would seem that the words "in function-definitions (8.4) and" should be stricken.

-> Editorial.

Function specifiers (7.1.2).

Paragraph 3 mentions that inline functions "shall have exactly the same definition in every case." This should be made more precise. Does "exactly the same" mean token equivalence or source equivalence? (Think of it this way: we know that implementations are not required to diagnose violations of the one-definition rule, and most do not. But, if an implementation wanted to diagnose violations, exactly what would they be required to check?)

-> Accepted: See 3.2 [basic.odr] for a definition of the ODR.

Namespace scope (7.3.1.3).

Paragraph 3 says that "The use of the static keyword is deprecated when declaring objects in a namespace scope." This statement belongs with a list of deprecated features in an Annex.

-> Editorial.

However, I disagree with deprecating this feature. This use of static has been around for years and does no particular harm. Leave it alone.

-> Rejected.
-> This was a deliberate decision. All of the functionality of
-> file-scope statics could be gotten with namespaces, so there was
-> really no reason to use file-scope static anymore.

The using declaration (7.3.3).

In the example after paragraph 4, the first comment should read: "//OK: B is a base of D2"

-> Editorial

Paragraph 7 says that "a using-declaration is a declaration and can therefore be used repeatedly where (and only where) multiple declarations are allowed." The"following example is provided (among others):

```
void f()
{
   using A::i;
   using A::i; // error: double declaration
}
```

I think that this double declaration should be well-formed, considering that the following is well-formed:

```
void g()
{
   extern int j;
   extern int j; // fine: redundant extern declaration
}
```

-> Rejected
-> A using declaration is not an extern declaration. It introduces the
-> name into the current scope in the same way that an ordinary
-> block scope declaration introduces a name. In these cases,
-> redeclarations are ill-formed.

Paragraph 16 says that use of access-declarations is deprecated. This should join the list of deprecated features in an Annex.

-> Editorial

Clause 8.

References (8.3.2).

In the example following paragraph 2, the third and fourth lines appear to be indented by mistake.

-> Editorial.

Clause 9.

Class members (9.2).

Paragraph 11 says, "The order of allocation of nonstatic members separated by an access-specifier is implementation-defined." Do we really want to force implementations to document this? It seems to me the order of allocation should be "unspecified."

-> Accepted.

Also, the non-normative "note" at the end of paragraph 11 seems out of place; it has nothing to do with the subject of the paragraph.

-> Editorial

Scope rules for classes (9.3).

In 2), the last letter ‘S’ should be in a constant-width font. This rule states that: "A name N used in a class S shall refer to the same declaration when re-evaluated in its context and in the completed scope of S." Line 3) says that: "If reordering member declarations in a class yields an alternate valid program under (1) and (2), the program’s behavior is undefined." However, the examples given on the next page describe violations of this rule as an "error" (which suggests that the examples are ill-formed).

Therefore, the text should be changed in one of two ways:

1. In (3), "the program’s behavior is undefined" should be changed to "the program is ill-formed."
   [You could also add that "no diagnostic is required," if this would make the implementors more comfortable.]

or,

2. In the examples, occurrences of the word "error" should be changed to "undefined behavior."

-> Editorial.
-> Option 1. describes the intended behavior.

Static member functions (9.5.1).

This subclause begins with a non-normative note, which is poor style.

-> Editorial.

Unions (9.6).

The restriction that "a union can have no static members" is gratuitous. Why this restriction?

-> Rejected, request for an extension

Bit fields (9.7).

The following statements made in this sub-clause appear to be non-normative and should be presented as "notes":

‘Fields straddle allocation units on some machines and not on others."

"Fields are assigned right-to-left on some machines, left-to-right on others."

"An unnamed bit-field is useful for padding to conform to externally-imposed layouts."
-> Editorial

Clause 10.

Member name lookup (10.2).

Paragraph 2 uses the word "we" two times. Since the rest of the document does not speak in the first person plural, the paragraph should be reworded.
-> Editorial.

Clause 11.

Access specifiers (11.1).

Paragraph 2 says that "the order of allocation of data members with separate access-specifiers is implementation-defined." I believe it should be "unspecified."
-> Accepted.

Protected member access (11.5).

The example in this subclause uses the word "illegal" several times. It should be changed to "ill-formed."
-> Editorial.

Clause 12.

This clause begins with a non-normative note, which is poor style. Normative text should be capable of standing alone. That is, if all the notes were deleted, the remaining text should still be meaningful. In this case, the first normative sentence in the clause says: "These member functions obey the usual access rules." Since it begins with the word "these," the sentence is reliant on the preceding note for context.
-> Editorial.

A footnote at the bottom of the first page of this clause says, "Volatile semantics might or might not be used." This is unacceptable. The Standard needs to specify what volatile semantics mean and when they are to be used.
-> Accepted.
-> Volatile semantics is never used in constructor.

Temporary objects (12.2).

The word "processor" occurs twice in this subclause. It should be changed to "implementation."
-> Editorial.

Conversion functions (12.3.2).

The example after paragraph 7 uses the word "illegal." It should be changed to "ill-formed."
-> Editorial.

Destructors (12.4).

Change the following sentence: "It is not possible to take the address of a destructor" to "The address of a destructor shall not be taken." (This parallels the wording used in the subclause on constructors.)

-> Editorial.

The footnote that "volatile semantics might or might not be used" is, again, unacceptable.

-> Accepted.
-> Volatile semantics is never used in destructor.

Free store (12.5).

In paragraph 3, change "will" to "shall."

-> Editorial.

Initializing bases and members (12.6.2).

Just before paragraph 4, there is a paragraph containing two notes ("if class X has a member m of class type..."). It is unclear why these two notes are non-normative. They seem to be making important points that belong in the normative text.

-> Editorial.

In the example after paragraph 7, the declaration of B’s default constructor needs to be indented.

-> Editorial.

Copying class objects (12.8).

In the example that follows paragraph 15, the last line of code before the terminating brace needs to be indented.

-> Editorial.

Clause 13.

Function call syntax (13.3.1.1).

In paragraph 2, change "first two cases" to "first and third cases."

-> Editorial.

Overloaded operators (13.5).

In paragraph 8, change the word "section" to "subclause."

-> Editorial.

Built-in operators (13.6).

In paragraph 3, change the word "section" to "subclause."

-> Editorial.

Clause 14.

Point of instantiation (14.3.2).

In paragraph 11, change "implementation quantity" to "implementation-defined quantity."

-> Editorial.

Explicit instantiation (14.4).

In paragraph 2, change "unqualifier-id" to "unqualified-id"

-> Editorial.

Template parameters (14.7).

Paragraph 6 says that a template-parameter shall not be of floating type. The example indicates, however, that a reference to a floating type is permitted. This is incomprehensible, since floating types and floating references allow the same set of values. I am aware of the rounding problems that afflict floating point values; however, values targeted by a reference may have the same problem - hence, my confusion about why this is allowed.

-> Rejected.
-> There is a difference between a reference argument (address is used)
-> and an rvalue argument (value is used).

Template arguments (14.8).

The lead-in to the example says: "Arrays as defined in 14 can be used like this." It is not clear where in "14" the text is referring to.

-> Editorial.

Template argument deduction (14.10.2).

In the example after paragraph 11, an argument of "aa" is deduced to be of type char*. I think this is wrong and potentially unsafe, since the effect of modifying a character literal is undefined and often harmful. Rather, the type deduced from a string literal should be const char*, so that if the generated function attempts to modify the string, the compiler can detect an error.

-> Rejected.
-> This would break C compatibility and C++ code too much.
-> (For example, function overload resolution would resolve
-> differently).

Overload resolution (14.10.3).

The text says: ‘For each function template, if the argument deduction succeeds, the deduced template arguments are used to generate a single template function, which is added to the candidate functions set to be used in overload resolution." This seems incorrect. A template function should be generated only after it is selected by overloaded resolution.

-> Editorial.
-> For the purpose of function overload resolution, only the template
-> function declaration is generated.

An example at the end of this subclause shows, once again, a type of char* being deduced from a string literal. I believe it should be const char*.

-> Rejected, see 14.10.2 above.

Overloading and linkage (14.10.4).

In the example, the notation f_PT_pi is used in a comment, which assumes the reader is familiar with the Cfront name-mangling scheme.

-> Editorial.

Clause 15.

Constructors and destructors (15.2).

Paragraph 2 contains the sentence: "If the object or array was allocated in a new-expression, the storage occupied by that object is sometimes deleted also." Such a statement is too wishy-washy for a standard.

-> Editorial.

Handling an exception (15.3).

Paragraph 13 says: "The exception being handled shall be rethrown if control reaches the end of a handler of the function-try-block of a constructor or destructor". This statement is ambiguous. It could mean one of two things:

If control reaches the end of a function-try-block of a constructor or destructor, the program shall behave as if the programmer explicitly coded a throw expression (with no argument) as the last statement in the block.

or

If control reaches the end of a function-try-block without the exception being rethrown, the program is ill-formed.

-> Editorial.

-> The intended meaning is the one described in option 1.

A similar ambiguity afflicts the next sentence: "Otherwise, "the function shall return when control reaches the end of a handler for the function-try-block."

-> Editorial.

Exception specifications (15.4).

Paragraph 4 says, "In other assignments or initializations, exception-specifications shall match exactly." It is not clear what "other" assignments or initializations the statement refers to.

-> Editorial.

The terminate() function (15.5.1).

The fourth dash refers to the concept of "stack unwinding," but this was defined only in a non-normative note. Either the definition should be made normative, or the sentence should be reworded.

-> Editorial.

Library Clauses.

Temporary buffers (20.4.3.5).

The prototype of return_temporary_buffer indicates a return type of void, but the description says that it "returns the buffer to which p points."

-> Editorial.

-> The return type of void is correct. The description will be clarified.

Template class auto_ptr (20.4.5)

This class should have a conversion operator to bool, so that programmers can use an auto_ptr instance in an if statement without having to code if (p.get()) ...

-> Rejected.

-> Note, however, that as a result of many comments including this one,
-> auto_ptr has been much corrected and improved.

String classes (21.1).

Paragraph 1 (on page 21-3) uses the word "we." The paragraph should be rephrased.

-> Editorial.

C type virtual functions (22.2.1.1.2).

The description of do_tolower says that it "converts a character or

characters to upper case.
-> Editorial

Sequences (23.1.1).

Paragraph 1 says that "the library provides three basic kinds of sequence containers: vector, list, and deque." It should read "four basic kinds," and bitset should be added to the list.
-> Editorial.

Iterator tags (24.1.6).

Paragraph 5 discusses the implications of an implementation providing an "additional pointer type far." However, such an implementation would be non-conforming, since implementations are not permitted to intrude on the user identifier name space. The problem would be solved by using the word __far instead of far.
-> Accepted.
-> Section 24.1.6 has been updated to use __far rather than far.

complex specializations (26.2.2).

The complex<float> specialization provides explicit converting constructors from complex<double> and complex<long double>, and the complex<double> specialization provides an explicit converting constructor from complex<long double>. All of these constructors invoke "narrowing" conversions and could result in undefined behavior if the source value falls outside the representable range of the destination type. Although we cannot prevent programmers from indulging in unsafe conversions, I do not think the Library should encourage them. These constructors should be deleted from the complex interface. (The user who wants to live dangerously could always achieve the same result through explicit casting of the real and imaginary parts of the source value.)
-> Rejected.
-> "explicit" means _non-converting_ not _converting_. There is no
-> such thing as an "explicit converting" constructor.

Standard iostream objects (27.3).

The Committee has adopted win as the wide-stream equivalent of cin. I fear this name is too likely to conflict with identifier names in existing code. For example, consider:

```
enum game_result ( win, lose );
```

I would recommend renaming the standard wide character streams to: wcin, wcout, wcerr, wclog.
-> Accepted.
-> The standard wide character streams have been renamed to:
-> wcin, wcout, wcerr, wclog.
-> See section 27.3.2 [lib.wide.stream.objects]

Class ios_base::Init(27.4.3.1.6).

I fail to see why the Init class is part of the normative Standard. It is an implementation detail -and hence, belongs in the realm of the implementor, not in the Standard.
-> Rejected.
-> The committee decided not to require that the implementation have
-> any magic for early initialization of the standard streams.
-> A library that uses the "nifty counter" trick is conforming.
-> There are times when the program needs to construct an Init object
-> explicitly and making the Init class part of the normative Standard
-> helps users to do this.

Annex A.

Templates (A.12).

The non-terminal symbols explicit-instantiation and specialization are introduced, but these occur nowhere else in the grammar. (If you fed the grammar to yacc in this form, yacc would complain that these symbols are never used.) They should be integrated into the grammar properly.

-> Editorial.

Annex B.

The implementation quantities listed are a superset of the translation limits in ISO C with one exception: ISO C has: "31 nesting levels of parenthesized declarators within a full expression." For completeness, this should be added to Annex B.

-> Rejected.
-> The Annex B on translation limits has been considered very
-> carefully by the committee and is the best compromise that was
-> acceptable to the majority of committee members.

The second-to-last item on page B1 refers to the non-terminal symbol struct-declaration-list, but there is no such symbol in the grammar.

-> Editorial.

Annex C.

Paragraph uses the term "Classic C," but this is never defined. The same paragraph uses the first-person plural "we." It should be reworded.

-> Editorial.

C++ and ISO C (C. 2).

The troff spacing macros in this section need to be adjusted; too much space appears between the sub-clause numbers and the descriptions of each change.

-> Editorial.

diff.stat (C.2.4).

In paragraph 1, the rationale says that "any use of the uninitialized object could be a disaster." In a document intended for international distribution, a word like "disaster" should not be used in a colloquial sense. Reword the sentence.

In paragraph 2, the word "processor" is used twice. Change it to "implementation."

-> Editorial.

diff.decl (C.2.6).

In paragraph 1, the code in the example does not line up properly.

In "Effect on original feature," the text says: "This feature was marked as ‘obsolescent’ in C." Change "C" to "ISO C."

In paragraph 2, the rationale section uses the word "legal." Change it to "well-formed."

In paragraph 4, the rationale section uses the words "major catastrophe." This is too colloquial for an international standard.

-> Editorial.

Anachronisms (C. 3).

I fail to see why the Standard is allowing implementations license to support so many anachronisms. Some of the anachronisms described (e.g. the overload keyword, assignment to this) are very old features that have not been a part of C++ for many years. You are giving implementations license to use an old style preprocessor: not even ISO C allowed that, and ISO C is already five years old.

I would eliminate the anachronisms altogether.

-> Rejected:
-> It is not required that a conforming implementation support these
-> features.

8- Comment from Stan Friesen
Received by email
email address: swf@elsegundoca.attgis.com

8.1 Section 1.7
paragraph 2 has a typographical error, the phrase "diagnosable error" is repeated twice, and there is incorrect punctuation.

-> Editorial.

8.2 Section 3.6.2
paragraph 1: I find this paragraph confusing. To me it appears as if the statement that objects "initializated with constant expressions are initialized before any dynamic ... initialization takes place" is in conflict with the statement that within a translation unit, "the order of inititialization of non-local objects ... is the order in which their definition appears".

-> Editorial.
-> The second sentence applies only to objects dynamically
-> initialized.

8.3 Section 3.9
The term "POD" is used before it is defined. At the very least a forward reference to the definition should be placed at the first such use.

-> Editorial.

8.4 Section 14.8
paragraph 2: In the example, the last item seems to violate the first sentence of the paragraph, in that ’p’ doesn’t look like a constant expression to me. I would think that ’p’ needs to be declared as "char * const" to make it a constant.

-> Editorial.
-> ’p’ is the address of an object with external linkage.
-> It is therefore ok to use ’p’ to initialize a template non-type
-> parameter that is a pointer.

8.5 Section 16.8
paragraph 1: I think an additional macro that is comparable to __STDC__ in that it is unique to supposedly standard conforming implementations. There are enough new features and changes that programmers may want to be able to #ifdef on ANSI conformance. I cannot see any way to do that with the set of macros you define here. [Most existing implementations already define __cplusplus, so it cannot be used in this way].

-> Accepted.
-> __cplusplus can be used in this way;
-> it is now defined to be a specific long integral value, intended to
-> represent the expected date of the official standard, currently
-> 199711L.

8.6 Section 20.4.5

(auto_ptr): In the description of operator=(), the line in the effects paragraph that says "copies the argument a to *this" is effectively redundant, since the reset() call mentioned in the next line accomplishes exactly that. I found this confusing when I was implementing this class.

-> Editorial.

8.7 Section 20.4.5

Also, neither release() nor reset() have a "Returns:" paragraph.

-> Editorial.

9- Comment from Ronald Fisher
Received by email
email address: ronald.fischer@acm.org

9.1 struct vs. class

9 par. 4 says:

"A structure is a class declared with the class-key 'struct'"

```
  struct A; // a forward declaration
  class A {public: int i; }
```

is A called a structure structure or a class?

-> Editorial.
-> Replace "declared" by "defined" in the sentence above.

9.2 Local variables and the scope for function prototype

3.3.2 says:

"In a function declaration, names of parameters have function prototype scope"

8.3.6 par. 7 says:

"Local variables shall not be used in default expressions"

Consider the following examples:

```
  // example 1
  int x;
  void f(int x, int y = x);
```

Is the default argument for y the global x or the first parameter x?

-> Editorial.
-> It is the parameter x.
-> The WP should say that a parameter is a local variable.
-> This means that the example above is ill-formed because a local
-> variable is used as a default argument.

Now let's change the example slightly:

```
  // example 2
  int y;
  void f(int x = y, int y = 0);
```

Is this valid? We know that the scope of the parameter y ends at the end of the function prototype, but where does it begin?

-> Already clear.
-> See 3.3 [basic.scope].

-> The scope of a name starts as soon as it has been declared.

9.3 base-clause of a class

9 par. 2 says:
"The name of a class can be used as a class-name even within the base-clause of the class-specifier itself"
To me, this implies that

```
class A : A {...};
```

would be legal, which is certainly not what was intended.

-> Editorial.
-> The footnote was rewritten to indicate that even if a class name is
-> previously hidden by the name of an object, function or enumerator,
-> the class name is found when used in a base-clause.

9.4 Position of cv-qualifiers in dec-specifier-seq

From the grammar follows clearly, that the cv-qualifier may appear either before or after the simple-type-specifier, i.e. that

```
const int i = 0;
```

and

```
int const i = 0;
```

are equivalent. All examples use however the first form. I suggest that, to emphasize that point, a few examples are written the other style (const after int) to clarify the point.

-> Editorial.

9.5 static array members (9.5.2))

One anomaly in C++ is the difference between the declaration of static array members and arrays which are not members at all. The latter can be defined by implicitly define the number of elements:

```
int ia[] = {5,3,4}; // has 3 elements
```

For static members, this is not possible:

```
struct S {
    static ia[3]; // number of members must be stated explicitly
};
int S::ia[] = {5,4,3};
```

-> Already allowed.
-> See 9.4.2[class.static.data] para 2.

9.6 Conversion to void

12.3.2 par. 1 says:
"If conversion-type-id is ’void’ ..., the program is ill-formed"

It seems to me an unnecessary restriction to exclude user-defined conversions to void, because it is well-defined, when voiding happens.

-> The language has been relaxed to allow declarations of user-defined
-> operator void. See 12.3.2 [class.conv.fct]

10- Second comment from Stan Friesen
Received by email
email address: swf@elsegundoca.attgis.com
Was comment T11 in the post-Monterey mailing document.

In 20.4.5.1 & 20.4.5.2: I see a possible problem with the specification of either the assignment operator or the reset() member function. Shouldn’t one or the other specify that the object pointed to by the previous pointer is deleted?

As it stands it looks as if an assignment of an auto_ptr<> would orphan any object owned by the auto_ptr<> assigned to.

-> Rejected.
-> Note, however, that as a result of many comments including this one,
-> auto_ptr has been much corrected and improved.

----

11 & 13 - Comment from Jay Zipnick /
          Intelligent Resources Integrated Systems
    Received by email
    email address: jzipnick@best.com
    Was comment T13 in the post-Monterey mailing document.

11.1 (Revision 1)
ISSUE 1) Arrays of incomplete types as formal arguments:

As per 8.3.4, Arrays, paragraph 1, "In a declaration T D where D has the form "D1 [const-expr(opt)]" ... . T shall not be a reference type, an incomplete type, ...".

```
struct foo;

void f1(int* arr  );    // legal
void f2(int  arr[]);    // legal
void f3(foo* arr  );    // legal
void f4(foo  arr[]);    // not legal
```

The bottom line, is that "void f4(foo arr[]);", above, is illegal because foo is incomplete. However I would like the committee to consider allowing this.

-> Accepted.

11.2 (Revision 1)
ISSUE 2) Function pointers and C linkage

Original code:

```
class foo
{
  // details omitted
  static int compare(void* key1, void* key2);
};
...
tree = tavl_init(foo::compare);    // pass function pointer
```

The problem is that class foo’s implementation uses a C library (for handling threaded AVL trees), and this C library needs to be passed function pointers. The seventh compiler has different calling conventions for C and C++. Seeking a *portable* solution, the following change was suggested by the compiler vendor:

```
class foo
{
  // details omitted
  static int _cdecl compare(void* key1, void* key2);
};
...
tree = tavl_init(foo::compare);    // pass function pointer
```

The problem here is that _cdecl is not part of the C++ standard.

-> Accepted.
-> The extern "C" language linkage is not part of a pointer to function
-> type.

12- Comment from Noel Yap
Received by email
email address: nyap@garban.com
Was comment T15 in the post-Monterey mailing document.

T12.1
1. friend granularity
One often wishes to grant a class, C0, or a function, f0, permission to change the value of a member, m1, of another class, C1. Usually, either a public set function is written (which grants global change permission), or C1 declares C0 or f0 as a friend (which grants to C0 or f0 complete access to C1). Since neither of these two choices is near optimal, I propose that member functions should be able to declare their friends:

```
void
C1::set_m1(int i)
{
        friend C0;
        friend f0(void);

        m1 = i;
}
```

-> Rejected.
-> This is not a very useful feature.
-> The friend declaration would be provided in the member function
-> body and (except for inline member functions) the body is visible in
-> one translation unit only.

T12.2
2. enum conversion overriding
If conversion functions from one type, C0, to an enum type, E1, were allowed, bool could then be implemented as an enum.

```
enum bool { false, true };

            bool::bool(int i)
            //      or, bool bool(int i)
            //      or, operator bool(int i)
            {
  return ((!i)   ?   false   :   true);
            }
```

-> Rejected.
-> Request for an extension.

----

14 & 16- Comment from David Sachs / Fermilab
(also unregistered comment U5)
Received by email
email address: sachs@fnal.fnal.gov
Was comment T16 in the post-Monterey mailing document.

T14.1
I) [class.mi] Section 10.1

All the examples in this section show only the case where all copies of a duplicated base class are indirect. The only discussion of the structurally simpler but lexically more complex case, in which there is a direct copy and 1 or more indirect copies, that I could find was in section 12.6.2 [class.base.init], and the language there clearly affirmed the legality of a class so designed.

In view of the clear legality of a class with distinct direct and indirect copies of the same base class, the C++ standard needs to specify proper syntax for:

a) referring to members of the distinct bases
b) casting a pointer (or reference) to an object of a derived class to a pointer (or reference) of each one of the distinct base class subobjects.

-> Editorial.
-> Words were added to the WP to indicate that a class with one direct
-> base of type A and one indirect base of type A is well-formed.

T14.2
II) [class.base.init] Section 12.6.2

There is no discussion of the case of a mem-initializer that specifies a name denoting both a nonstatic data member and a direct or virtual base class. Declaring such an initialize to be ill formed would be a reasonable resolution.

-> Editorial.
-> The names in the expression-list of a mem-initializer are first
-> evaluated in the scope of the constructor’s class and then
-> evaluated in the first enclosing namespace scope that contains the
-> constructor definition.
-> In the case mentioned above, name look up finds the member name
-> first.

T14.3
III) [class.base.init] Section 12.6.2

When are parameters of mem-initializers evaluated?

Language in this section clearly hints that the intent of the standards committee is that each mem-initializer should be treated as a complete expression with its parameters evaluated after all previous initialization. However, such a requirement is NOT stated explicitly.

This leaves in limbo code like

```
class x{
  int a;
  int b;
  x(int i) : a(i), b(a) {...}
  ...};
```

-> Accepted.
-> See 12.6.2 [class.base.init] end of paragraph 3.

T14.4
IV) [class.copy] section 12.8

The requirement that a constructor for a class X of the form X(volatile X&) or X(const volatile X&) is NOT a copy constructor, and the similar requirement for operator= should be EMPHASIZED, rather than relegated to an appendix.

-> Accepted.
-> See 12.8 [class.ctor] for a description of how volatile affects

-> copy constructors and copy assignment operators.

15- Comment from Mok-Kong Shen
Received by email
email address: Mok-Kong.Shen@lrz-muenchen.de
Was comment T17 in the post-Monterey mailing document.

Subject: Multidimensional Arrays (8.3.4)

Abstract: The C++ multidimensional arrays are inferior to those of e.g. Fortran and thus need to be improved for the language to gain wider acceptance in the fields of engineering and scientific numerical computations hithertofore absolutely dominated by Fortran. It is suggested that a new data type be added to the C++ standard for that purpose.

-> Rejected.
-> Request for an extension.

17- Comment from David Olsen
Received by email
email address: olsen@Rational.COM
Was comment T19 in the post-Monterey mailing document.

17.1
Section 2.8 [lex.key], paragraph 4 lists new[], delete[], new<%%>, and delete<%%> as tokens. new<%%> and delete<%%> are not mentioned anywhere else in the document that I can find. They should be listed in Section 2.4 [lex.digraph] as alternate representations for new[] and delete[] respectively.

-> Accepted.
-> new[], delete[], new<%%>, and delete<%%> are not tokens.
-> The draft was modified to reflect this.
-> See sections: 2.3[lex.pp.token], 2.4[lex.digraph], 2.5[lex.token]

17.2
Section 5.3.5 [expr.delete], paragraph 1 contains the following syntax for a delete-expression.

```
    delete-expression:
            ::opt delete cast-expression
            ::opt delete [ ] cast-expression
```

One more possibility should be added.

```
            ::opt delete[] cast-expression
```

If a program does not contain any whitespace between the word delete and the pair of brackets, then the compiler must interpret it as a single delete[] token, not as three separate tokens (delete, [, and]). But the delete[] token is not part of a valid delete-expression, resulting in a syntax error.

-> Accepted.
-> new[], delete[], new<%%>, and delete<%%> are not tokens.
-> The draft was modified to reflect this.
-> See sections: 2.3[lex.pp.token], 2.4[lex.digraph], 2.5[lex.token]

17.3
I have some concerns about the example in Section 9.8 [class.nest], paragraph 1. The relevent parts are quoted here:

```
    int x;
```

```
        class enclose {
        public:
            int x;
            class inner {
              void g(enclose* p, int i)
              {
                   p->x = i; // ok: assign to enclose::x
              }
            };
        };
```

I would like to argue that the line "p->x = i;" is an error because the class enclose is incomplete, but I can find no clear statement of exactly when a class becomes complete.

-> Editorial.
-> It is well-formed. The body of member functions of a nested
-> class are looked up in the scope of the class assuming the complete
-> definition of the class (and the complete definition of the class
-> enclosing classes) have been seen.
-> See 3.3.6 [basic.scope.class].

17.4

In Section 20.4.5.2 [lib.auto.ptr.members], it is never specified what the member functions auto_ptr<X>::release and auto_ptr<X>::reset should return.

-> Editorial.

17.5

Section 24.3.1.1 [lib.reverse.bidir.iter] contains the description of the template class reverse_bidirectional_iterator. The member functions base() and operator*() do not change the object on which they are called, and should therefore be constant member functions. This would affect both the class definition in 24.3.1.1 and the descriptions of the two members in 24.3.1.2.2 and 24.3.1.2.3.

The same argument applies to the template class reverse_iterator and its member functions base() and operator*() in Sections 24.3.1.3, 24.3.1.4.2, and 24.3.1.4.3.

-> Accepted.

17.6

In Section 24.3.1.2.5 [lib.reverse.bidir.iter.op--], the return value of reverse_bidirectional_iterator<B,T,R,D>::operator--() is not specefied. There is a Returns clause, but it is empty.

-> Editorial.
-> Returns: *this

17.7

Section 24.3.1.2.6 [lib.reverse.bidir.iter.op==] contains the description for reverse_bidirectional_iterator<B,T,R,D>::operator==. The Returns clause states:

```
  Returns: BidirectionalIterator(x) == BidirectionalIterator(y)
```

This assumes that there exists a conversion from a reverse_bidirectional_iterator to the BidirectionalIterator class on which it is based. This was true in early versions of STL, but is not the case in the current draft standard. The conversion operator has been replaced by the member function base(). Therefore, the Returns clause should be changed to either:

```
  Returns: x.current == y.current
```

or:

Returns: x.base() == y.base()

both of which are equivalent.

-> Accepted.

-> See 24.3.1.2.7 [lib.reverse.bidir.iter.op==].

17.8

Section 24.3.1.3 [lib.reverse.iterator] contains the description of the template class reverse_iterator. At the end of the class definition are declarations of operator==, operator<, operator-, and operator+. These should not be in the class definition, but should be non-member functions.

-> Accepted.

17.9

Section 24.3.1.4 [lib.reverse.iter.ops] does not contain any description for many of the reverse_iterator operators: the default constructor for reverse_iterator; the member functions operator+, operator+=, operator-, and operator-=; and the non-member functions operator<, operator-, and operator+.

-> Accepted.

-> 24.3.1.4.7 through 24.3.1.4.15 contain the descriptions.

17.10

Section 25.1.3 [lib.alg.find.end] describes the template function find_end. The complexity clause states:

Complexity: At most last1 - first1 applications of the corresponding predicate.

find_end is almost exactly like the template function search (25.1.9) except that it finds the last occurance rather than the first. The complexity of search is quadratic ((last1 - first1) * (last2 - first2)) rather than linear. Footnote 196 in Section 25.1.9 explains that, while a linear algorithm exists, it is slower in most practical cases. I don’t see why the reason for making search quadratic should not apply to find_end as well. In my opinion, the complexity clause for find_end should be changed to:

Complexity: At most (last1 - first1) * (last2 - first2) applications of the corresponding predicate.

-> Accepted with Complexity:

->   (last2-first2)*(last1-first1-(last2-first2)+1)

17.11

Section 25.1.4 [lib.alg.find.first.of] describes the template function find_first_of. I see problems with both the Returns and Complexity clauses.

The Returns clause states:

Returns: The first iterator i in the range [first1, last1-(last2-first2)) such that for any non-negative integer n < (last2-first2), the following corresponding conditions hold: *i == *(first2+n), pred(i, first2+n) == true. Returns last1 if no such iterator is found.

As I read this, every member of the range [first2, last2) must be equal, since the result must compare equal to every one of them. My guess is that it was intended for the result to compare equal to any one member of the range [first2, last2), in which case the Returns clause should read:

Returns: The first iterator i in the range [first1, last1) such that there exists some non-negative integer n < (last2-first2) where the following corresponding conditions hold: *i == *(first2+n), pred(i, first2+n) == true. Returns last1 if no such iterator is found.

The Complexity clause for find_first_of states:

Complexity: Exactly find_first_of(first1, last1, first2+n) applications of the corresponding predicate.

But find_first_of(first1, last1, first2+n) is not a legal function call, and find_first_of returns an iterator, not a number. So the Complexity clause just doesn’t make any sense. And given that the Returns clause didn’t make sense either, I am not sure what the complexit should be.

-> Accepted.

17.12

Section 25.1.9 [lib.alg.search] describes the template function search. There are four overloaded version of the function:

```
template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator1
    search(ForwardIterator1 first1, ForwardIterator1 last1,
           ForwardIterator2 first2, ForwardIterator2 last2);

template<class ForwardIterator1, class ForwordIterator2,
     class BinaryPredicate>
ForwardIterator1
    search(ForwardIterator1 first1, ForwardIterator1 last1,
           ForwardIterator2 first2, ForwardIterator2 last2,
           BinaryPredicate pred);

template<class ForwardIterator, class Size, class T>
ForwardIterator
    search(ForwardIterator first, ForwardIterator last,
           Size count, const T& value);

template<class ForwardIterator, class Size, class T,
     class BinaryPredicate>
ForwardIterator
    search(ForwardIterator first, ForwardIterator last,
           Size count, T value, BinaryPredicate pred);
```

But there is an overload ambiguity between the first and third versions and between the second and fourth versions. For example, given the following code:

```
int *f1, *l1, *f2, *l2;
// Set f1, l1, f2, and l2 to be valid iterators
search(f1, l1, f2, l2);
```

The call to search could match the first version with both ForwardIterator1 and ForwardIterator2 as (int *), or it could match the third version with ForwardIterator, Size, and T all as (int *). I cannot think of any case where the first or second versions would be better matches than the third or fourth versions. Therefore, I think the third and fourth versions of search should be renamed to something different.

-> Accepted.
-> The third and fourth versions were renamed search_n .

17.13

Section 26.2.1 [lib.complex] contains the definition of the template class complex. The definition contains three different constructors:

```
complex();
```

```
        complex(T re);
        complex(T re, T im);
```

Section 26.2.3 [lib.complex.members], however, only contains a description of a single constructor with default arguments:

```
      complex(T re = T(), T im = T());
```

Either of these sections should be changed to match the other one.

-> Editorial.
-> 26.2.1 should match 26.2.3.

18- Comment from Don Organ / Megatest
Received by email
email address: dorgan@megatest.com
Was comment T20 in the post-Monterey mailing document.

Provide static virtual member functions.

-> Rejected.
-> Request for an extension.

19- Comment from Babak Sehari
Received by email
email address: sehari@iastate.edu
Was comment T22 in the post-Monterey mailing document.

In order to make C++ programming language more international, the terminal input and output functions of C++ should be able to handle various languages requirement. Due to the fact that some languages such as Chinese are written from top to bottom and some other languages such as Arabic, Handi, Urdu, and Persian are written from right to left, a C++ Standard should be able to deal with input and output in these languages using all terminal functions. This can be done using a call to overload the functions and operators, such as:

```
    char term_dir( char direction);
```

where direction may be defined as:

```
                0   left to right ( normal English)
                1   right to left
                2   top to bottom
```

return value:

```
                0 unsuccessful
                1 successful
```

This call will effect behavior of functions such as printf and scanf and overloads operators such as << and >>.

For example to read and write a chinese document after English text one can write:

```
    char load_chinese_fonts(); // a function to be defined by the
                               // programmer

    main()
    {
           char answer[10],answer2[10];

           printf("The language would you prefer?)"
           scanf(answer);
           printf("\n");
```

```
          load_chinese fonts();
          term_dir(2);
          printf("chinese text");
          scanf(answer2);       // now the scanf should enter the data from
                           // top to bottom
   }
```

-> Rejected.
-> Request for an extension.

20- Comments from David Vandevoorde
Received by email
email address: vandevod@cs.rpi.edu
Was comment T29 in the post-Monterey mailing document.

Comments on the proposed <valarray> header

-> Accepted in substance at Santa Cruz meeting;
-> commenter was present during discussions.

21- Comment from WG14
Received by email
email address: pjp@plauger.com
Was comment T21 in the post-Monterey mailing document.

21.1 Core
UK Comments on C++ CD for Public Review

Clause 1.1
Paragraph 2, last sentence. Delete this sentence and Annex C.1.2.
This is the first standard for C++, what happened prior to 1985 is
not relevant to this document.
-> Rejected.
-> The committee views Annex C.1.2 as informative and helpful to users.
-> It decided that it is worth to include it in the final standard.

Clause 1.2
Paragraph 1, change "ISO/IEC 9899:1990, C Standard" to
"ISO/IEC 9899:1990 Programming Languages -- C"

Paragraph 1, change "ISO/IEC 9899:1990/DAM 1, Amendment to C
Standard"to "ISO/IEC:1990 Programming languages -- C AMENDMENT 1: C
Integrity"

Add year of current publication of ISO/IEC 2382
-> Editorial.

Clause 1.3

Paragraph 1, multibyte character. Last sentence. What is the basic
character set? Is it the basic source character set or basic
execution character set (see clause 5.2.1 of ISO 9899)? There is
an index refence for basic execution character set to this clause.

Also need to add definitions of the basic execution and basic source
character set. See ISO 9899, Clause 5.2.1.
-> This has been revised to describe basic source character set and
-> character set mapping more precisely.

Paragraph 1, undefined behaviour. ISO 9899 states that "Undefined
behaviouris otherwise indicated in this International Standard by the
words "undefined behaviour" or by the omission of any explicit
definition of behaviour".

The C++ standard should also adopt the rule that omission of explicit
defintion of behaviour results in undefined behaviour.
-> Accepted.

Paragraph 1, well-formed program. Other standards use the term
Conforming to describe this concept. The C++ standard should follow
this precedent. It should also introduce the concept of Strict
Conformance, that is a program that contains no undefined,
implementation defined or unspecified behaviours.
-> Rejected.
-> The Conformance model was discussed extensively by the committee and
-> the Conformance model proposed in the draft (see 1.7,
-> [intro.compliance]) is the best compromise that was acceptable to the
-> majority of committee members.

Clause 1.5, paragraph 1, second sentence. Contains a use of the
term "basic execution character set". See previous discussion.
-> Character discussion in clause 2 is now extensively revised. We
-> followed precedent of the C standard in defining "basic source
-> character set" precisely, then describing, rather than defining,
-> "basic execution character set".

Clause 1.8, paragraph 4. Need to include text stating that the
standard imposes no requirements on the behaviour of programs that
contain undefined behaviour.
-> Accepted.

Clause 1.8, paragraph 9, second sentence. What is a "needed
side-effect"? This paragraph, along with footnote 3 appears to be a
definition of the C standard "as-if" rule. This rule should be
defined as such.
-> Extensively revised for greater compatibility with the C standard’s
-> model of sequence points and side effects.

Clause 2.1, phase 8, first sentence. Change "The translation units
that will form a program are combined." to "The translation units
are combined to form a program."
-> Editorial.

Clause 2.2, paragraph 1. Delete and replace with wording from
C standard. "All occurrences in a source file of the following
sequences of three characters (called trigraph sequences) are replaced
with the corresponding single character. No other trigraph sequence
exists. Each ? that does not begin one of the above trigraphs listed
above is not changed."
-> Editorial.

Clause 2.3, paragraph 3, first sentence. Change "... lexically analsysed
..."
to "... parsed ...". To agree with wording in C standard.
-> Editorial.

Clause 2.3, paragraph 3, last sentence. Delete ", even if that would
cause further lexical analysis to fail". To agree with existing, clear
wording in C standard.
-> Editorial.

Clause 2.4. This is a gratuatous difference from the Addendum
to the C standard with no technical merit. It should be deleted
and replaced by the text from the Addendum.
-> Rejected.
-> There was a committee vote to differ from C in this regard.

Clause 2.8, paragraph 3. Reserving identifiers containing a double
underscore is overly restrictive. Identifiers starting with
double underscore should be reserved.

-> Rejected.

Clause 2.9.1, paragraph 1. This is a clumsy rewrite of the description in Clause 6.1.3.2 of the C standard. Replace by the text contained in the two paragraphs of the Description in Clause 6.1.3.2.
-> Editorial.

Clause 2.9.1, paragraph 2. This is a clumsy rewrite of the semantics in Clause 6.1.3.2 of the C standard. Replace by the text contained in the two paragraphs of the Semantics in Clause 6.1.3.2.
-> Editorial.

Clause 2.9.2, paragraph 1, second sentence. What is "the machine’s character set"? Is this the basic source character set that we have forgotten to define? Suggest that the wording from C standard, Clause 6.1.3.4, Semantics, first paragraph be used (it contains the important concept of mapping).
-> This has been revised to describe basic source character set and
-> character set mapping more precisely.

Clause 2.9.2, paragraph 2. Suggest that C standard, Clause 6.1.3.4, Semantics, second paragraph be used as the basis of a rewrite of this paragraph.
-> Editorial.

Clause 2.9.2, paragraph 3. Suggest that C standard, Clause 6.1.3.4, Description, paragraph 2, 3, 4, and 5 be used as the basis of a rewrite of this paragraph.
-> Editorial.

Clause 2.9.2, paragraph 4. Ditto comment on paragraph 3.
-> Editorial.

Clause 2.9, paragraph 1. Suggest that this be replaced by C standard Clause 6.1.3.1, Description, paragraph 1. Otherwise the term "missing" should be replaced by "ommitted".
-> Editorial.

Clause 2.9.4. Suggest that paragraph 1, 2 and 3 be replaced by C standard, Clause 6.1.4, all paragraphs in Description and Semantics.
-> Editorial.

Clause 2.9.4, paragraph 4. Delete. The size of a string is not equal to the number of characters it contains. The \" rule is already covered by the text from the C standard. The first paragraph belongs in an introductory text to the language.
-> Editorial.

Clause 5.16, syntax rule. Change "assignment-expression" to "conditional-expression" to agree with the C standard, ISO 9899 Clause 6.3.15
-> Rejected.
-> Explicit decision for the throw-expression (Nov 91).

Page 32 Para 9

This states :
Types bool, char, wchar_t, and the signed and unsigned integer types are collectively called integral types. 27) A synonym for integral type is integer type.

ISO 9899 does not include wchar_t as a member of the integral types, this should at least be noted in Annex C, and does raise a number of compatability issues
-> Rejected.
-> In C, wchar_t is a typedef for some integral type. The committee is

-> not aware of any strictly-conforming C program whose behavior is
-> altered by this change. Therefore, this is not listed in Annex C.

Page 84 Para 5

The underlying type of an enumeration is an integral type, not gratuitously larger than int

Is this meant to be a requirement on an implementation ?

if so then the requirement should be stated positively.
i.e. an enumeration is an integral type that can represent all enumerator values .... otherwise remove the not gratuitously ...

-> Editorial.
-> Yes, the requirement is on the implementation.

1.7 Processor compliance para 2
typo -diagnosable errors repeated

-> Editorial

Page 6 para 18
the word builtin needsd a hypen i.e built-in

-> Editorial

Paragraph 3.3.4 Page 20
Scope

File 1

```
// First file
// declare i in global namespace as per page 20 of draft
// and has external linkage

int i=5;
```

File 2

```
//Second file

static int i = 10 ;  // declare i in global namespace with internal
linkage
int y = ::i ;   // What is the value of y
                // does :: resolve linkage to external or internal ??

void f(void)
{
        int i =6;
        int j =::i; // Global namespace i internal or external

}
```

-> Editorial
-> The i with internal linkage in file 2 where it is referenced.

If an implementation is required to accept both

```
int main(){}
```

and

```
int main(int argc, char * argv[]){}
```

Is it permitted to have a prototype of both forms visible ?

```
int main();
int main(int, char **);
```

If not is a disgnostic required nn this case.

-> Already resolved.
-> See 3.6.1, para 2: "This function... cannot be overloaded..."

Page 77

The following two statements appear to contradict each other

The inline specifier is a hint to the implementation that inline substitution of the function body is to be preferred to the usual function call implementation. The hint can be ignored.

The above statement clearly indicates that inline can be ignored however the draft goes on to state:

A function (8.3.5, 9.4, 11.4) defined within the class definition "is" inline.

Is an implementation free to ignore the inline within a class definition ?

-> Editorial.
-> Inlining at the point of call is a hint.
-> The semantics of inline functions as described in 7.1.2 must always
-> be respected.

Page 45 para 7 [expr.call]

This section describes the promotions prior to a function call and refers to section 4.5 (integral promotions), however section 4.5 refers to promotion of wchar_t and bool, paragraph 7 remains silent on wchar_t and bool leaving a question over whether promotion of these takes place prior to the function call.

-> Editorial.
-> Yes, promotion of wchar_t and bool also applies.

The following are points directly relating to C.

Clause 3.9, paragraph 6, last sentence. In ISO 9899 an incomplete type is not an object type (Clause 6.1.2.5, first paragraph). Defining an "incompletely-defined object type" is a needless incompatibility with ISO 9899. Use another term.
-> Already discussed by the C++ committee and rejected.

Clause 3.9, paragraph 7, last sentence. ISO 9899 allows a typedef declaration of an array of unknown size to be later completed for a specific object (Clause 6.5.7, example 6). C++ should also allow such a usage. Disallowing this construct is a needless incompatibility.
-> Already discussed by the C++ committee and rejected.

3.6.2. The latitude with which static initialization might occur is problematic for use of the floating-point environment, viz. the floating-point exception flags and rounding direction modes required by IEC559. The sequence { clear-overflow-flag, compute, test-overflow-flag } would be defeated if the implementation chose to execute some overflowing static initializations between the clear and test. The sequence { set-special-rounding, compute, restore-usual-rounding } could affect the results of static initializations the implementation chose to execute between the set and restore. In order to support the floating-point environment, some implementations, depending on their initialization model, might

need to insulate static initialization with say { save-FP-environment, set-default-FP-environment, execute-initializations, restore-FP-environment }. A note to this effect would be helpful.
-> Rejected.
-> Commenter is encouraged to write a proposal that could be included
-> in a non-normative appendix.

3.9.1, P10, Box 21. Yes, say "at least as much range and precision". Both are desired, and one doesn’t imply the other.
-> Editorial.

5, P4. The first sentence may not be clear. I assume "where the operators really are" means the rearrangement in question would not change values. Better would be to disallow rearrangement (except by the as-if rule). "Rearrangement" is better than "regrouping", as the distributive law is problematic too.
-> Editorial.
-> This sentence was moved to 1.8 [intro.execution] paragraph 16.

!!! 5, P12. There’s no mention of license for wide evaluation of floating expressions, as in 3.2.1.5 of the C standard. Wide evaluation is needed by the host of systems based on wide registers.
-> Editorial.

21.2 Library

17.3.1.1, P10, Table 15. Typo: unititialized_fill
-> Editorial.

17.3.3.1.2, P1. This seems to say that a header can optionally declare or define any names it wishes. This statement may have been taken out of context
from the C standard, where, I thought, the optional reserved names were confined to those in the subsequent bullets.
-> Editorial.

17.3.3.2, P1. Sentence is difficult to parse.
-> Editorial.

17.3.4.2, P1. Footnote says masking macros are disallowed. Why disallow them?
-> Accepted.

!!! 17. Assuming wide expression evaluation is allowed, math functions should
be able to have return types appropriate to the implementation’s expression evaluation method. E.g. if the minimum evaluation format is double, then cos should have the prototypes

```
  double cos(float);
  double cos(double);
  long double cos(long double);
```

(Note this doesn’t affect signatures.)
-> Rejected.
-> The amount of effort required to make this change is considered
-> too large for this late in the standards process.

17.3.4.8, P3, Box 70. I think it’s right to not require C functions to throw exceptions, but why prohibit it?
-> Rejected.
-> This specific point was discussed and the wording of the WP carefully
-> choosen to reflect the view of the committee.

18.2.1.1. Is tinyness_before actually useful for any programming task? Being in the interface makes the diligent programmer worry about whether she needs to consider it. The IEEE 754 (IEC 559) standardization group regarded it as an implementation option that didn’t matter to the user.
-> Rejected.
-> This field is required to conform to the LIA-1 standard.

18.2.1.2, P23, 27. Footnote says these are equivalent to xxx_MIN_EXP and xxx_MAX_EXP, but their definitions don’t imply that. Better to use the same wording as in the C standard.
-> Accepted.

18.2.1.2, P23, 25, 27, 29. These refer to "range", which is intended to imply normalized. "Range of normalized floating-point numbers", as in the C standard, would avoid the ambiguity.
-> Accepted.

18.2.1.2, P61. round_style would be more useful if its value reflected the current execution-time rounding style, which can be changed dynamically on most systems, including all IEC559 ones.
-> Rejected.
-> No other items in this class are dynamic.
-> It is considered better to retain consistency.

18.2.1.4, P2. Example is inconsistent in that is_iec559 is true but has_denorm is false -- IEC559 requires denorms.
-> Accepted.

19.1. The hierarchy of exceptions is confusing. (1) What are the differences between domain_error, invalid_argument, and out_of_range? (2) out_of_range and range_error sound like the same thing but aren’t. (3) In mathematics (though not the C standard), domain refers to argument values and range to return values, but here out_of_range refers to argument values. (4) How do they map to the IEC559 exceptions (invalid, overflow, underflow, div-by-zero, and inexact)?
-> Editorial.

19.1. I believe (and hope) there’s not a requirement that builtin operators on builtin types or standard math functions throw any of these exceptions, but a reader might leap to the conclusion that they do.
-> Editorial.
-> 19.1 and 17.3.4.8 now seem quite clear on this point so no further action
-> is contemplated.

!!! 26.2. The complex library provides a subset of the capabilities one might expect from builtin complex types. A description of what capabilities are and are not supported would be very helpful. What conversions? Which among complex<int>, complex<long>, complex<float>, and complex<double> have implicit conversions? What (mixed mode) operations? Do integer and complex operands mix (e.g. complex_z * 2)? Is double_complex_z * 2.0L OK? Without this description the reader must infer from the overloading rules. (It appears there are no implicit conversions from complex to real nor from wider to narrower among complex<long double>, complex<double>, and complex<float>, which presumably allows for automatic "promotions" from real to complex and from narrower to wider complex types. Saying so much -- whatever is correct -- would be helpful.)
-> Rejected.
-> The standard is not a tutorial, and the reader should be able to

-> infer the allowable conversions by looking at the class descriptions.
-> It is correct that, because of the non-converting constructors,
-> widening but not narrowing is allowed.

!!! 26.2 In reviewing the complex library I'm further confounded by not being
able to try it. It uses member templates, which aren't implemented in either
of the two compilers I have access to. Are there enough implementations of
this?
-> Rejected.
-> There are lots of features in the library that are not yet
-> implemented by most compilers, member templates is one of them.
-> The standard library is meant to be used with a standard compiler.
-> If a vendor's compiler is not standard then the vendor should wait
-> to release the standard library or put in workarounds (i.e for
-> the complex component you could specialize each of the member
-> templates on float, double, and long double.)

26.2 (and elsewhere). The lack of rationale makes review more difficult.

26.2, P1. Typo in the second divide operator.
-> Editorial.

26.2.1. What are the requirements for type X?
-> Rejected.
-> The requirements for X are the same as for type T, you must be able
-> to instantiate a complex<X>.

!!! 26.2.2. Compound assignments should be overloaded for real operands.
This is CRITICAL for consistency with IEC559 and for efficiency (see section
2.3.6 of "Complex C Extensions", Chapter 6 of X3J11's TR on Numerical C
Extensions), particularly since the binary operators are defined in terms of
the compound assignments. complex_z *= 2.0 must not entail a conversion of
2.0 to complex.
-> Accepted.
-> The following member were added:

```
->
->   basic_complex<T>& operator=(T);
->   basic_complex<T>& operator+=(T);
->   basic_complex<T>& operator-=(T);
->   basic_complex<T>& operator*=(T);
->   basic_complex<T>* operator\=(T);
->   etc.
->
```

-> for basic_complex<float>, basic_complex<double>,
-> basic_complex<long double>

26.2.2. Why initialize re and im to 0?.
-> Rejected.
-> All existing complex libs known of (AT&T, DEC, etc.)
-> will initialize the real and imaginary parts of a complex
-> library to 0 if you declare complex c; (i.e. no args).

26.2.3. How do the default arguments like T re = T() apply to builtin types
like int?
-> Rejected.
-> According to the standard, float re = float() should work and it
-> should initialize re to 0 (see Section 5.2.3 of the working draft)

26.2.4. The class declarations for the compound assignments use member
templates, but they don't show up here. Likewise the complex(const
complex<X>&) constructor is missing.
-> Editorial.

!!! 26.2.5. Definitions for binary operators refer to compound assignments,
but compound assignments aren't declared for complex<T> op= T. This is a

deficiency in the compound assignments (see above). Also the semantics are wrong for T op complex<T>, as they entail a conversion of T to complex<T> (see above).
-> Accepted.

26.2.5. for ==, typo: lhsp.real
-> Editorial.

26.2.5. For ==, the Returns and Notes parts are awkward.
-> Rejected.
-> The committee doesn’t understand why this is "awkward."

26.2.5. For !=, typo in Returns part.
-> Editorial.

26.2.6. abs is missing.
-> Editorial.

26.2.6. Can’t review the two TBS.
-> Editorial.

26.2.6. I believe the term "norm" commonly refers to the square root of the squared magnitude (i.e. abs), and not the squared magnitude. Is a function for the squared magnitude needed? Note that the squared magnitude can be computed from abs with only deserved over/underflow, but not vise versa.
-> Rejected.

26.2.6. Typos in argument list for polar.
-> Editorial.

26.2.7. I don’t think atan2 should be overloaded for complex arguments? How would it be defined?
-> Rejected.
-> It would be defined as "return atan(x/y)", x being the first
-> arg and y being the second arg. complex number division would
-> occur.

26.2.7. log10(z) is easily computed as log(z)/log(10.0), so isn’t really necessary.
-> Rejected
-> The existing C standard library has both log and log10.

!!! 26.2.7. Branch cuts and ranges need to be specified for functions. See section 3 of "Complex C Extensions", Chapter 6 of X3J11’s TR on Numerical C Extensions.
-> Accepted.

26.5. There’s no long double version of ldexp.
-> Editorial.

26.5. The float version of modf is out of alphabetical order.
-> Editorial.

26.5. pow doesn’t accommodate mixed mode calls. E.g. pow(2.0f, 3.0) is ambiguous, matching both pow(float,float) and pow(float,int). pow(2.0, 3L) is ambiguous too. A description (clearer than the overloading rules) would be helpful. Maybe more overloads are desirable.
-> Rejected.
-> Two reasons: more overloads could lead to more ambiguities and it
-> was felt that mixed-mode arithmetic calls such as pow(double, float)
-> was unusual and dangerous enough that forcing the user to add casts
-> was acceptable. Someone commented "users need to be careful with
-> mixed-mode arithmetic anyway."

26.5. New overloads make math functions ambiguous for integer arguments, e.g.

```
   atan(1) would be ambiguous.  C++ would be more restrictive than C in this
   respect.  Of course, more overloads could solve the problem.
-> Rejected.
-> Same reason as above.

   !!! 26.5.  The functions in <fp.h> and <fenv.h>, specified in "Floating-Point
   C Extensions", Chapter 5 of X3J11's TR on Numerical C Extensions, support a
   substantially broader spectrum of numerical programming.
-> Rejected.
-> Not important enough to do given time considerations (No one in the
-> committe was willing to spend time writing a concrete proposal).
-> Also, some of this support is already in the numeric_limits class.

   ---------------------------------

   17.3.1.3:
   A freestanding implementation doesn't include <stdexcept>,
   which defines class exception, needed by <exception>.
   Should probably move class exception to <exception>.
-> Accepted.

   17.3.3.1:
   A C++ program must be allowed to extend the namespace std if only
   to specialize class numeric_limits.
-> Accepted.

   17.3.4.1:
   Paragraph 4 is a repeat.
-> Editorial.

   18.2.1:
   float_rounds_style should be float_round_style (correct once).
-> Accepted.

   18.2.1.1:
   Paragraph 2 is subsumed by the descriptions of radix, epsilon(),
   and round_error(). Should be removed here.
-> Accepted.

   18.2.1.1:
   Paragraph 3 is repeated as 18.2.1.2, paragraph 50, where it belongs.
   Should be removed here.
-> Accepted.

   18.2.1.1:
   Should say that numeric_limits<T> must be able to return T(0).
   Should say that round_style defaults to round_indeterminate,
   not round_toward_zero.
-> Rejected.
-> Paragraph 4 describes the default template.
-> The default for round_style is as in C.

   18.2.1.2:
   denorm_min() does *not* return the minimum positive normalized value.
   Should strike the mention of this function in paragraph 2.
-> Accepted.

   18.2.1.2:
   Paragraph 22 must supply a more precise definition of ''rounding error.''
-> Accepted.

   18.2.1.2:
   Paragraph 23 must replace ''is in range'' with
   ''is a normalized value''.
-> Accepted.
```

18.2.1.2:
Paragraph 25 must replace ‘‘is in range’’ with ‘‘is a normalized value’’.
-> Accepted.

18.2.1.2:
Paragraph 27 must replace ‘‘is in range’’ with ‘‘is a finite value’’.
-> Accepted.

18.2.1.2:
Paragraph 29 must replace ‘‘is in range’’ with ‘‘is a finite value’’.
-> Accepted.

18.2.1.2:
In paragraph 41, ‘‘flotaing’’ should be ‘‘floating’’.
-> Editorial.

18.2.1.3:
Semantics must be specified for enum float_round_style.
-> Accepted.

18.5.1:
type_info::operator!=(const type_info&) is ambiguous in the presence of the template operators in <utility>, and it is unnecessary. It should be removed.
-> Rejected.

18.6.1.1:
Paragraph 1 incorrectly states that bad_exception is thrown by the implementation to report a violation of an exception-specification. Such a throw is merely a permissible option.
-> Editorial.

18.7:
There are five Table 28s.
-> Editorial.

19.1.1:
exception(const exception&) should not be declared with the return type exception&. (Error repeated in semantic description.)
-> Editorial.

20.1:
Allocators are described in terms of ‘‘memory models’’ which is an undefined concept in Standard C++. The term should be *defined* here as the collection of related types, sizes, etc. in Table 33 that characterize how to allocate, deallocate, and access objects of som managed type.
-> Editorial.

20.1:
Paragraph 3 talks about ‘‘amortized constant time’’ for allocator operations, but gives no hint about what parameter it should be constant with respect to.
-> Rejected.
-> Clear enough for practical purposes.

20.1:
a.max_size() is *not* ‘‘the largest positive value of X::difference_type.’’ It is the largest valid argument to a.allocate(n).
-> Editorial.

20.1:
Table 33 bears little resemblance to the currently accepted version

of class allocator (though it should, if various bugs are fixed, as described later.) Essentially *every* item in the ‘expression’ column is wrong, as well as all the X:: references elsewhere in the table.
-> Editorial.

20.3:
binder1st is a struct in the synopsis, a class later. Should be a class uniformly, like binder2nd.
-> Editorial.

20.3.5:
class unary_negate cannot return anything. Should say that its operator() returns !pred(x).
-> Editorial.

20.3.6.1:
binder1st::value should have type Operation::first_argument_type, not argument_type.
-> Editorial.

20.3.6.3:
binder2nd::value should have type Operation::second_argument_type, not argument_type.
-> Editorial.

20.3.7:
‘‘Shall’’ is inappropriate in a footnote, within a comment, that refers to multiple memory models not even recognized by the Standard.
-> Editorial.

20.4:
return_temporary_buffer shouldn’t have a second (T*) parameter. It’s not in STL, it was not in the proposal to add it, and it does nothing.
-> Editorial.

20.4.1:
allocator::types<T> shows all typedefs as private. They must be declared public to be usable.
-> Editorial.

20.4.1:
It is not clear from Clause 14 whether explicit template member class specializations can be first declared outside the containing class. Hence, class allocator::types<void> should probably be declared inside class allocator.
-> Rejected. No longer applies.

20.4.1:
The explicit specialization allocator::types<void> should include:

```
    typedef const void* const_pointer;
```

It is demonstrably needed from time to time.
-> Editorial.

20.4.1:
Footnote 169 should read ‘‘An implementation,’’ not ‘‘In implementation.’’
-> Editorial.

20.4.1.1:
allocator::allocate(size_type, types<U>::const_pointer) has no semantics for the second (hint) parameter.
-> Editorial.

20.4.1.1:
allocator::allocate(size_type, types<U>::const_pointer) requires

that all existing calls of the form A::allocate(n) be rewritten as al.allocate<value_type, char>(n, 0) -- a high notational price to pay for rarely used flexibility. If the non-template form of class allocator is retained, an unhinted form should be supplied, so one can write al.allocate<value_type>(n).

-> Accepted.

20.4.1.1:

allocator::allocate(size_type, types<U>::const_pointer) should return neither new T nor new T[n], both of which call the default constructor for T one or more times. Note that deallocate, which follows, calls operator delete(void *), which calls no destructors. Should say it returns operator new((size_type)(n * sizeof (T))).

-> Accepted.

20.4.1.1:

allocator::max_size() has no semantics, and for good reason. For allocator<T>, it knew to return (size_t)(-1) / sizeof (T) -- the largest sensible repetition count for an array of T. But the class is no longer a template class, so there is no longer a T to consult. Barring a general cleanup of class allocator, at the least max_size() must be changed to a template function, callable as either max_size<T>() or max_size(T *).

-> Accepted.

20.4.1.1:

A general cleanup of class allocator can be easily achieved by making it a template class once again:

```
template<class T> class allocator {
public:
    typedef size_t    size_type;
    typedef ptrdiff_t difference_type;
    typedef T*        pointer;
    typedef const T*  const_pointer;
    typedef T&        reference;
    typedef const T&  const_reference;
    typedef T         value_type;
    pointer address(reference x) const;
    const_pointer address(const_reference x) const;
    pointer allocate(size_type n);
    void deallocate(pointer p);
    size_type init_page_size() const;
    size_type max_size() const;
    };
```

The default allocator object for a container of type T would then be allocator<T>(). All of the capabilities added with the Nov. ’94 changes would still be possible, and users could write replacement allocators with a *much* cleaner interface.

-> Accepted with amendments from N0790R1 = 95-0190

20.4.1.2:

operator new(size_t N, allocator& a) can’t possibly return a.allocate<char, void>(N, 0). It would attempt to cast the second parameter to allocator::types<void>::const_pointer, which is undefined in the specialization allocator::types<void>. If related problems aren’t fixed, the second template argument should be changed from void to char, at the very least.

-> Accepted.

20.4.1.2:

If allocator is made a template class once again, this version of operator new becomes:

```
template<class T>
    void *operator new(size_t, allocator<T>& a);
```

-> Accepted.

20.4.1.3:

The example class runtime_allocator supplies a public member allocate(size_t) obvoously intended to mask the eponymous function in the base class allocator. The signature must be allocate<T, U>(size_t, types<U>::const_pointer) for that to happen, however. The example illustrates how easy it is to botch designing a replacement for class allocator, given its current complex interface. (The example works as is with the revised template class allocator described earlier.)

-> Accepted.

20.4.2:

raw_storage_iterator<OI, T>::operator*() doesn’t return ‘‘a reference to the value to which the iterator points.’’ It returns *this.

-> Editorial.

20.4.3.1:

Template function allocate doesn’t say how it should ‘‘obtain a typed pointer to an uninitialized memory buffer of a given size.’’ Should say that it calls operator new(size_t).

-> Accepted.

20.4.3.2:

Template function deallocate has no semantics. Should say that it calls operator delete(buffer).

-> Accepted.

20.4.3.5:

get_temporary_buffer fails to make clear where it ‘‘finds the largest buffer not greater than ...’’ Do two calls in a row ‘‘find’’ the same buffer? Should say that the template function allocates the buffer from an unspecified pool of storage (which may be the standard heap). Should also say that the function can fail to allocate any storage at all, in which case the ‘first’ component of the return value is a null pointer.

-> Accepted.

20.4.3.5:

Strike second parameter to return_temporary_buffer, as before. Should say that a null pointer is valid and does nothing. Should also say that the template function renders indeterminate the value stored in p and makes the returned storage available for future calls to get_temporary_buffer.

-> Editorial.

20.4.4:

Footnote 171 talks about ‘‘huge pointers’’ and type ‘‘long long.’’ Neither concept is defined in the Standard (nor should it be). This and similar comments desperately need rewording.

-> Editorial.

20.4.4.3:

Header should be ‘‘uninitialized_fill_n’’, not ‘‘uninitialized_fill.’’

-> Editorial.

20.4.5:

When template class auto_ptr ‘‘holds onto’’ a pointer, is that the same as storing its value in a member object? If not, what can it possibly mean?

-> Editorial.

20.4.5:

auto_ptr(auto_ptr&) is supposed to be a template member function.

-> Editorial.

20.4.5:

auto_ptr(auto_ptr&) is supposed to be a template member function.
-> Editorial.

20.4.5:
auto_ptr<T>::operator= should return auto_ptr<T>&, not void, according
to the accepted proposal.
-> Editorial.

20.4.5.1:
Need to say that auto_ptr<T>::operator= returns *this.
-> Editorial.

20.4.5.2:
auto_ptr<T>::operator->() doesn’t return get()->m -- there is no m.
Should probably say that ap->m returns get()->m, for an object ap
of class auto_ptr<T>.
-> Editorial.

20.4.5.2:
auto_ptr<T>::release() doesn’t say what it returns. Should say
it returns the previous value of get().
-> Editorial.

20.4.5.2:
auto_ptr<T>::reset(X*) doesn’t say what it returns, or that it deletes
its current pointer. Should say it executes ‘‘delete get()’’ and
returns its argument.
-> Editorial.

20.5:
The summary of <ctime> excludes the function clock() and the
types clock_t and time_t. Is this intentional?
-> Editorial.

21.1:
template function operator+(const basic_string<T,tr,A> lhs,
   const_pointer rhs) should have a second argument of type
const T *rhs.
-> Accepted.

21.1:
Paragraph 1 begins, ‘‘In this subclause, we call...’’ All first person
constructs should be removed.
-> Editorial.

21.1.1.1:
string_char_traits::ne is hardly needed, given the member eq.
It should be removed.
-> Rejected.

21.1.1.1:
string_char_traits::char_in is neither necessary nor sufficient.
It simply calls is.get(), but it insists on using the basic_istream
with the default ios_traits. operator>> for basic_string still has
to call is.putback(charT) directly, to put back the delimiter that
terminates the input sequence. char_in should be eliminated.
-> Accepted.

21.1.1.1:
string_char_traits::char_out isn’t really necessary.
It simply calls os.put(), but it insists on using the basic_ostream
with the default ios_traits. char_out should be eliminated.
-> Accepted.

21.1.1.1:
string_char_traits::is_del has no provision for specifying a locale,

even though isspace, which it is supposed to call, is notoriously locale dependent. is_del should be eliminated, and operator>> for strings should stop on isspace, using the istream locale, as does the null-terminated string extractor in basic_istream.

-> Accepted.

21.1.1.1:

string_char_traits is missing three important speed-up functions, the generalizations of memchr, memmove, and memset. Nearly all the mutator functions in basic_string can be expressed as calls to these three primitives, to good advantage.

-> Accepted.

21.1.1.2:

No explanation is given for why the descriptions of the members of template class string_char_traits are ‘‘default definitions.’’
If it is meant to suggest that the program can supply an explicit specialization, provided the specialization satisfies the semantics of the class, then the text should say so (here and several other places as well).

-> Accepted.

21.1.1.2:

string_char_traits::eos should not be required to return the result of the default constructor char_type() (when specialized). Either the specific requirement should be relaxed or the function should be eliminated.

-> Accepted.

21.1.1.2:

string_char_traits::char_in, if retained, should not be required to return is >> a, since this skips arbitrary whitespace. The proper return value is is.get().

-> Not an issue because char_in is eliminated.

21.1.1.2:

string_char_traits::is_del, if retained, needs to specify the locale in effect when it calls isspace(a).

-> Not an issue because is_del is eliminated.

21.1.1.3:

Paragraph 1 doesn’t say enough about the properties of a ‘‘char-like object.’’ It should say that it doesn’t need to be constructed or destroyed (otherwise, the primitives in string_char_traits are woefully inadequate). string_char_traits::assign (and copy) must suffice either to copy or initialize a char_like element.
The definition should also say that an allocator must have the same definitions for the types size_type, difference_type, pointer, const_pointer, reference, and const_reference as class allocator::types<charT> (again because string_char_traits has no provision for funny address types).

-> Accepted.

21.1.1.4:

The copy constructor for basic_string should be replaced by two

constructors:

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

The copy constructor should copy the allocator object, unless explicitly stated otherwise.

-> Accepted.

21.1.1.4:

basic_string(const charT*, size_type n, Allocator&) should be

```
   required to throw length_error if n > max_size(). Should say:
   Requires: s shall not be a null pointer
             n <= max_size()
   Throws: length_error if n > max_size().
-> No change.

   21.1.1.4:
   basic_string(size_type n, charT, Allocator&) is required to throw

   length_error if n == npos. Should say:
   Requires: n <= max_size()
   Throws: length_error if n > max_size().
-> No change.

   21.1.16:
   basic_string::size() Notes says the member function ‘‘Uses
   traits::length(). There is no reason for this degree of
   overspecification. The comment should be struck.
-> Accepted.

   21.1.1.6:
   basic_string::resize should throw length_error for n >= max_size(),
   not n == npos.
-> Accepted.

   21.1.1.6:
   resize(size_type) should not have a Returns clause -- it’s a void
   function. Clause should be labeled Effects.
-> Accepted.

   21.1.1.6:
   resize(size_type) should call resize(n, charT()), not
   resize(n, eos()).
-> Accepted.

   21.1.16:
   basic_string::resize(size_type) Notes says the member function
   ‘‘Uses traits::eos(). It should actually use charT() instead.
   The comment should be struck.
-> Accepted.

   21.1.1.6:
   basic_string::reserve says in its Notes clause, ‘‘It is guaranteed
   that...’’ A non-normative clause cannot make guarantees. Since the
   guarantee is important, it should be labeled differently.
   (This is one of many Notes clauses that make statements that should
   be normative, throughout the description of basic_string.)
-> Accepted.

   21.1.1.8.2:
   basic_string::append(size_type n, charT c) should return
   append(basic_string(n, c)). Arguments are reversed.
-> Accepted.

   21.1.1.8.3:
   basic_string::assign(size_type n, charT c) should return
   assign(basic_string(n, c)). Arguments are reversed.
-> Accepted.

   21.1.1.8.4:
   basic_string::insert(size_type n, charT c) should return
   insert(basic_string(n, c)). Arguments are reversed.
-> Accepted.

   21.1.1.8.4:
```

basic_string::insert(iterator p, charT c) should not return p, which may well be invalidated by the insertion. It should return the new iterator that designates the inserted character.
-> Accepted.

21.1.1.8.4:
basic_string::insert(iterator, size_type, charT) should return void, not iterator. (There is no Returns clause, luckily.)
-> Accepted.

21.1.1.8.5:
basic_string::remove(iterator) says it ‘‘calls the character’s destructor’’ for the removed character. This is pure fabrication, since constructors and destructors are called nowhere else, for elements of the controlled sequence, in the management of the basic_string class. The words should be struck.
-> Accepted.

21.1.1.8.5:
basic_string::remove(iterator, iterator) says it ‘‘calls the character’s destructor’’ for the removed character(s). This is pure fabrication, since constructors and destructors are called nowhere else, for elements of the controlled sequence, in the management of the basic_string class. The words should be struck.
-> Accepted.

21.1.1.8.5:
basic_string::remove(iterator, iterator) Complexity says ‘‘the destructor is called a number of times ...’’ This is pure fabrication, since constructors and destructors are called nowhere else, for elements of the controlled sequence, in the management of the basic_string class. The Complexity clause should be struck.
-> Accepted.

21.1.1.8.6:
replace(size_type pos1, size_type, const basic_string&,...) Effects has the expression ‘‘size() - &pos1.’’ It should be ‘‘size() - pos1.’’
-> Accepted.

21.1.1.8.6:
basic_string::replace(size_type, size_type n, charT c) should return replace(pos, n, basic_string(n, c)). Arguments are reversed.
-> Accepted.

21.1.1.8.8:
basic_string::swap Complexity says ‘‘Constant time.’’ It doesn’t say with respect to what. Should probably say, ‘‘with respect to the lengths of the two strings, assuming that their two allocator objects compare equal.’’ (This assumes added wording describing how to compare two allocator objects for equality.)
-> Accepted.

21.1.1.9.1:
basic_string::find(const charT*, ...) Returns has a comma missing before pos argument.
-> Editorial.

21.1.1.9.8:
basic_string::compare has nonsensical semantics. Unfortunately, the last version approved, in July ’94 Resolution 16, is also nonsensical in a different way. The description should be restored to the earlier version, which at least has the virtue

of capturing the intent of the original string class proposal:
1) If n is less than str.size() it is replaced by str.size().
2) Compare the smaller of n and size() - pos with traits::compare.

3) If that result is nonzero, return it.
4) Otherwise, return negative for size() - pos < n, zero for
size() - pos == n, or positive for size() - pos > n.
-> Accepted.

21.1.1.10.3:
operator!=(const basic_string&, const basic_string&) is ambiguous
in the presence of the template operators in <utility>, and it is
unnecessary. It should be removed.
-> Not a problem.

21.1.1.10.5:
operator>(const basic_string&, const basic_string&) is ambiguous
in the presence of the template operators in <utility>, and it is
unnecessary. It should be removed.
-> Not a problem.

21.1.1.10.6:
operator<=(const basic_string&, const basic_string&) is ambiguous
in the presence of the template operators in <utility>, and it is
unnecessary. It should be removed.
-> Not a problem.

21.1.1.10.7:
operator>=(const basic_string&, const basic_string&) is ambiguous
in the presence of the template operators in <utility>, and it is
unnecessary. It should be removed.
-> Not a problem.

21.1.1.10.7:
operator>= with const charT* rhs should return
lhs >= basic_string(rhs), not <=.
-> Accepted.

21.1.1.10.8:
Semantics of operator>> for basic_string are vacuous.
Should be modeled after those for earlier string class.
-> Accepted.

21.1.1.10.8:
Semantics of operator<< for basic_string are vacuous.
Should be modeled after those for earlier string class.
-> Accepted.

21.1.1.10.8:
getline for basic_string reflects none of the changes adopted by
July ’94 resolution 26. It should not fail if a line exactly fills,
and it should set failbit if it *extracts* no characters, not if it
appends no characters. Should be changed to match 27.6.1.3.
-> Accepted.

21.1.1.10.8:
getline for basic_string says that extraction stops when npos - 1
characters are extracted. The proper value is str.max_size() (which
is less than allocator.max_size(), but shouldn’t be constrained
more precisely than that). Should be changed.
-> Accepted.

21.2:
There are five Table 44s.
-> Editorial.

21.2:
<cstring> doesn’t define size_type. Should be size_t.
-> Accepted.

```
   22.1:
   template operator<<(basic_ostream, const locale&) as well as
   template operator>>(basic_ostream, const locale&) now have a second
   template argument (for ios traits) added without approval. While
   this change may be a good idea, it should be applied uniformly (which
   has not happened), and only after committee approval.
-> Rejected.
-> Other parts of the library clauses were changed to match.

   22.1.1:
   locale::category is defined as type unsigned. For compatibility with
   C, it should be type int.
-> Accepted.
-> The type is int in the current Working Draft.

   22.1.1:
   class locale has the constructor locale::locale(const locale& other,
   const locale& one, category). I can find no resolution that calls
   for this constructor to be added.
-> Rejected.
-> The committee has examined this matter and elected not to recommend a
-> change to this part of the Working Draft.

   22.1.1:
   Example of use of num_put has silly arguments. First argument
   should be ostreambuf_iterator(s.rdbuf()).
-> Editorial.

   22.1.1:
   Paragraph 8 says that locale::transparent() has unspecified behavior
   when imbued on a stream or installed as the global locale. There is
   no good reason why this should be so and several reasons why the
   behavior should be clearly defined. The sentence should be struck.
-> The WG has voted to eliminate transparent locales

   22.1.1:
   Paragraph 9 says that ‘‘cach[e]ing results from calls to locale facet
   member functions during calls to iostream inserters and extractors,
   and in streambufs between calls to basic_streambuf::imbue, is
   explicitly supported.’’ In the case of inserters and extractors,
   this behavior follows directly from paragraph 8. No need to say it
   again. For basic_streambuf, the draft can (and should) say explicitly
   that the stream buffer fixates on a facet at imbue time and ignores
   any subsequent changes that might occur in the delivered facet
   until the next imbue time (if then). (An adequate lifetime for the
   facet can be assured by having the basic_streambuf object memorize
   a copy of a locale object directly containing the facet,
   as well as a pointer to the facet, for greater lookup speed.)
   In any event, saying something ‘‘is explicitly supported’’ doesn’t
   make the behavior *required.* The paragraph should be struck, and
   words added to the description of basic_streambuf to clarify the
   lifetime of an imbued codecvt facet. (More words are needed here
   anyway, for other reasons.)
-> The paragraph referred to has been rewritten by the editor:
->
-> In successive calls to a locale facet member function during a call
-> to an iostream inserter or extractor or a streambuf member function,
-> the returned result shall be identical.  [ Note: This implies that
-> such results may safely be reused without calling the locale facet
-> member function again, and that member functions of iostream classes
-> cannot safely call imbue() themselves, except as specified elsewhere.
-> ]

   22.1.1.1.1:
   Table 46 lists the ctype facets codecvt<char, wchar_t, mbstate_t>
   and codecvt<wchar_t, char, mbstate_t> as being essential, but what
```

about codecvt<char, char, mbstate_t>? Should say that this facet must be present and must cause no conversion.

-> Accepted.
-> The current Working Draft lists a codecvt<char,char,mbstate_t> facet.

22.1.1.1.1:
Table 46, and paragraph 3 following, identify the facets that implement each locale category (in the C library sense). But these words offer no guidance as to what facets should be present in the default locale (locale::classic()). The template classes listed each represent an unbounded set of possible facets. Should list the following explicit instantiations of the templates as being required, along with those explicit instantiations already listed in Table 46:

```
 num_get<char, istreambuf_iterator<char> >
 num_get<wchar_t, istreambuf_iterator<wchar_t> >
 num_put<char, ostreambuf_iterator<char> >
 num_put<wchar_t, ostreambuf_iterator<wchar_t> >
 money_get<char, istreambuf_iterator<char> >
 money_get<wchar_t, istreambuf_iterator<wchar_t> >
 money_put<char, ostreambuf_iterator<char> >
 money_put<wchar_t, ostreambuf_iterator<wchar_t> >
 time_get<char, istreambuf_iterator<char> >
 time_get<wchar_t, istreambuf_iterator<wchar_t> >
 time_put<char, ostreambuf_iterator<char> >
 time_put<wchar_t, ostreambuf_iterator<wchar_t> >
```

-> Accepted.
-> The current Working Draft lists these facets.

22.1.1.2:
As mentioned earlier, locale::locale(const locale&, const locale&, category) has been added without approval. It should be struck.

-> Rejected.
-> The committee has examined this matter and elected not to recommend a
-> change to this part of the Working Draft.

22.1.1.3:
Description of locale::use() Effects contains a nonsense statement: ‘‘Because locale objects are immutable, subsequent calls to use<Facet>() return the same object, regardless of changes to the global locale.’’ If a locale object is immutable, then changes to the global locale should *always* shine through, for any facet that is not present in the *this locale object. If the intent is to mandate cacheing semantics, as sketched out in the original locales proposal, this sentence doesn’t quite succeed. Nor should it. Cacheing of facets found in the global locale leads to horribly unpredictable behavior, is unnecessary, and subverts practically any attempt to restoee compatibility with past C++ practice and the current C Standard. The sentence should be struck.

-> The description has been changed eliminating caching semantics.

22.1.1.3:
Description of locale::use Notes uses the term ‘‘value semantics’’ and the verb ‘‘to last.’’ Are either of these terms defined within the Standard? The sentence should be reworded, or struck since it’s non-normative anyway.

-> The description has been changed eliminating the language mentioned.

22.1.1.5:
locale::transparent() Notes says ‘‘The effect of imbuing this locale into an iostreams component is unspecified.’’ If this is a normative statement, it doesn’t belong in a Notes clause. And if it’s intended to be normative, it should be struck. Imbuing a stream with locale::transparent() is the *only* way to restore the behavior of iostreams to that in effect for *every* C++ programming running today. It is also essential in providing compatible behavior with the C Standard. The sentence should be struck.

-> The member transparent() has been eliminated.

22.2.1.3.3:
ctype<char> describes this subclause as ‘‘overridden virtual functions,’’ but they’re not. A template specialization has nothing to do with any virtuals declared in the template. Should be renamed.

-> Editorial.

22.2.1.4:
Description of codecvt, paragraph 3, fails to make clear how an implementation can ‘‘provide instantiations for <char, wchar_t, mbstate_t> and <wchar_t, char, mbstate_t>.’’ Must specializations be written for the template? If so, must they also have virtuals that do the actual work? Or can the implementation add to the default locale facets derived from the template, overriding the virtual do_convert? Needs to be clarified.

-> Editorial.

22.2.1.4:
Implementations should also be required to provide an instantiation of codecvt<char, char, mbstate_t> which transforms characters one for one (preferably by returning noconv). It is needed for the very common case, basic_filebuf<char>.

-> Accepted.

-> The current Working Draft requires this facet.

22.2.1.4.2:
codecvt::do_convert uses pointer triples (from, from_next, from_end) and (to, to_next, to_end) where only pairs are needed. Since the function ‘‘always leaves the from_next and to_next pointers pointing one beyond the last character successfully converted,’’ the function must be sure to copy from to from_next and to to to_next early on. A better interface would eliminate the from and to pointers.

-> Rejected.

-> This was discussed by the committee and no change was recommended.

22.2.1.4.2:
codecvt::do_convert says ‘‘If no translation is needed (returns noconv), sets to_next equal to argument to.’’ The previous paragraph strongly suggests that the function should also set from_next to from. Presumably, the program will call do_convert once, with nothing to convert. If it returns noconv, the program will omit future calls to do_convert. If that is the intended usage, then it should be permissible to call any instance of do_convert with (mostly) null pointers, to simplify such enquiries -- and the wording should make clear how to make such a test call.

-> The Working Draft mentions a member "always_noconv()" which the

-> committee believes addresses this concern.

22.2.1.4.2:
codecvt::do_convert Notes says that the function ‘‘Does not write into *to_limit.’’ Since there is no requirement that converted characters be written into sequentially increasing locations starting at to, this is largely toothless. Effects clause should be written more precisely.

-> Editorial.

22.2.1.4.2:
codecvt::do_convert Returns says that the function returns partial if it ‘‘ran out of space in the destination.’’ But the function mbrtowc, for example, can consume the remaining source characters, beyond the last delivered character, and absorb them into the state. It would be a pity to require do_convert to undo this work. Should say that partial can also mean the function ran out of source characters partway through a conversion. Then clarify that, after a return of partial, the next call to do_convert should

begin with any characters between from_next and from_end, of which there might be none.

-> Accepted.

-> The current Working Draft clarifies this matter.

22.2.2.1:

Template class num_get defines the type ios as basic_ios<charT>, which it then uses widely to characterize parameters. Thus, this facet can be used *only* with iostreams classes that use the default traits ios_traits<charT>. Since the use of num_get is mandated for *all* basic_istream classes, this restriction rules out essentially any substitution of traits. Best fix is to make the ios parameter an ios_base parameter on all the do_get calls, then change ios accordingly. This is sufficient if setstate is moved to ios_base as proposed elsewhere. But it requires further fiddling for num_put if fill is moved *out* of ios_base, as also proposed. Must be fixed, one way or another.

-> In the current Working Draft this parameter is an ios_base&, with
-> error reporting via a separate iostate& parameter.

22.2.2.2:

Template class num_put defines the type ios as basic_ios<charT>, which it then uses widely to characterize parameters. Thus, this facet can be used *only* with iostreams classes that use the default traits ios_traits<charT>. Since the use of num_put is mandated for *all* basic_ostream classes, this restriction rules out essentially any substitution of traits. Best fix is to make the ios parameter an ios_base parameter on all the do_put calls, then change ios accordingly. This is sufficient if setstate is moved to ios_base as proposed elsewhere. But it requires further fiddling for num_put if fill is moved *out* of ios_base, as also proposed. Must be fixed, one way or another.

-> In the current Working Draft this parameter is an ios_base&, with
-> a separate fill character parameter.

22.2.3.1:

The syntax specified for numeric values is out of place in numpunct.

-> Editorial.

22.2.3.1:

Description of numpunct says, ‘‘For parsing, if the digits portion contains no thousands-separators, no grouping constraint is applied.’’ This suggests that thousands-separators are permitted in an input sequence, and that the grouping constraint is applied, but it is profoundly unclear on how this might be done. Allowing thousands-separators at all in input is risky -- requiring that grouping constraints be checked is an outrageous burden on implementors, for a payoff of questionable utility and desirability. Should remove any requirement for recognizing thoudands-separators and grouping on input. And the effect on output needs considerable clarification.

-> Rejected.

-> Discussed by the committee, no change recommended.

22.2.4:

Template classes collate, time_get, time_put, money_get, money_put, money_punct, messages, and their support classes still have only sketchy semantics -- over a year after they were originally accepted into the draft. They are based on little or no prior art, and they present specification problems that can be addressed properly only with detailed descriptions, which do not seem to be forthcoming. Even if adequate wording were to magically appear on short notice, the public still deserves the courtesy of a proper review. For all these reasons, and more, the remainder of clause 22 from this point on should be struck.

-> Rejected.

22.2.7.1:
messages_base::THE_POSIX_CATALOG_IDENTIFIER_TYPE is not a defined type.
-> Accepted.
-> The current Working Draft defines this type as int.

27.1.1:
The definition of ‘‘character’’ is inadequate. It should say that it
is a type that doesn’t need to be constructed or destroyed, and that
a bitwise copy of it preserves its value and semantics. It should
also say that it can’t be any of the builtin types for which conflicting
inserters are defined in ostream or extractors are defined in istream.
-> Rejected.
-> As stated in 27.1.1 [lib.iostreams.definitions], the WP provides two
-> definitions related to ‘‘character’’. The ‘‘character container type’’
-> definition states that a character container class shall have a
-> trivial constructor and destructor and a copy constructor and copy
-> assignment operator that preserves its value and semantics. The WP
-> definitions for ‘‘character’’ and ‘‘character container type’’ is
-> adequate in the sense that it allows implementation to work fine while
-> not restraining future character types.

27.1.2.4:
Description of type POS_T contains many awkward phrases. Needs rewriting
for clarity.
-> Editorial.

27.1.2.4:
Paragraph 2 has ‘‘alg’’ instead of ‘‘all.’’
-> Accepted.

27.1.2.4:
Footnote 207 should say ‘‘for one of’’ instead of ‘‘for one if.’’
Also, it should‘‘whose representation has at least’’ instead of
‘‘whose representation at least.’’
-> Editorial.

27.2:
Forward declarations for template classes basic_ios, basic_istream,
and basic_ostream should have two class parameters, not one. It
is equally dicey to define ios, istream, etc. by writing just one
parameter for the defining classes. All should have the second
parameter supplied, which suggests the need for a forward reference
to template class ios_char_traits as well, or at least the two usual
specializations of that class.
-> Accepted.

27.3:
<iostream> is required to include <fstream>, but it contains no overt
references to that header.
-> Rejected.
-> <iostream> is not required to include <fstream>.

27.3.1:
cin.tie() returns &cout, not cout.
-> Accepted.

27.3.2:
win.tie() returns &cout, not cout.
-> Rejected.
-> wcin.tie() returns &wcout, not cout, &cout or wcout.

27.4:
streamsize is shown as having type INT_T, but subclause 27.1.2.2
says this is the integer form of a character (such as int/wint_t).

streamsize really must be a synonym for int or long, to satisfy all
constraints imposed on it. (See Footnote 211.)
-> Accepted.
-> streamsize is now of type SZ_T.

27.4:
Synopsis of <ios> is missing streampos and wstreampos. (They appear
in later detailed semantics.) Should be added.
-> Accepted.
-> iosfwd was added to address this.

27.4:
Synopsis of <ios> has the declaration:
template <class charT> struct ios_traits<charT>;
The trailing <charT> should be struck.
-> Editorial.
-> The trailing <charT> needs to be struck.

27.4.1:
Type wstreamoff seems to have no specific use. It should be struck.
-> Rejected.

27.4.2:
ios_traits::state_type is listed as ‘‘to be specified.’’
It needs to be specified.
-> Accepted.
-> ios_traits::state_type is now defined as being of type
-> STATE_T, which is defined in 27.1.2.6 [lib.iostreams.state.t].

27.4.2:
Definition of ios_traits lists arguments backwards for is_whitespace.
Should have const ctype<char_type>& second, as in later description.
(Also, first argument should be int_type, as discussed in 27.4.2.3.)
-> Accepted.

27.4.2:
ios_traits description should make clear whether user specialization
is permitted. If it isn’t, then various operations in <locale> and
string_char_traits are rather restrictive. If it is, then the draft
should be clear that ios_traits<char> and ios_traits<wchar_t> cannot
be displaced by a user definition.
-> ios_traits and string_char_traits were deprecated in favor of
-> char_traits, so the comment is no longer valid.

27.4.2:
The draft now says ‘‘an implementation shall provide’’ instantiations
of ios_traits<char> and ios_traits<wchar_t>. It was changed without
approval from ‘‘an implementation may provide.’’ This change directly
contradicts Nov 94 Resolution 23. The proper wording should be
restored.
-> Rejected.
-> An implementation has to provide instantiations of ios_traits<char>
-> and ios_traits<wchar_t>, therefore using ‘‘shall’’ rather than
-> ‘‘may’’ reflect the correct intended meaning.

27.4.2.2:
ios_traits::not_eof should take an argument of int_type, not char_type.
-> Accepted.

27.4.2.2:
ios_traits::not_eof says nothing about the use made of its argument c.
Should say that it returns c unless it can be mistaken for an eof().
-> Accepted.
-> ios_traits and string_char_traits were deprecated in favor of
-> char_traits. The description of char_traits::not_eof is:
-> Returns: c, if !eq_int_type(c,eof()), otherwise some value v such

```
-> that !eq_int_type(v,eof()).

   27.4.2.2:
   ios_traits::not_eof has two Returns clauses. The second is an
   overspecification and should be struck.
-> Accepted.
-> See 27.4.2.1 [lib.ios.traits.values]

   27.4.2.2:
   ios_traits::length has an Effects clause but no Returns clause.
   The Effects clause should be reworded as a Returns clause.
-> Accepted.
-> ios_traits and string_char_traits were deprecated in favor of
-> char_traits.  The description of char_traits::lenght is:
-> Returns: the smallest non-negative value of i such that the
-> expression eq(s[i],charT(0)) is true.

   27.4.2.3:
   First argument to is_whitespace has been changed from int_type to
   char_type with no enabling resolution. It is also a bad idea.
   Should be restored to int_type.
-> Rejected.
-> The function is_whitespace was deprecated, so the comment is no
-> longer valid.

   27.4.2.3:
   is_whitespace supposedly behaves ‘‘as if it returns ctype.isspace(c),’’
   but that function doesn’t exist. Should say ‘‘as if it returns
   ctype.is(ctype_base::space, c).’’
-> Rejected.
-> The function is_whitespace was deprecated, so the comment is no
-> longer valid.

   27.4.2.3:
   The draft now says that ios_traits functions to_char_type, to_int_type,
   and copy are ‘‘provided from the base struct
   string_char_traits<CHAR-T>.’’ This is a substantive change made
   without approval. It is also nonsensical, since there is no such
   ‘‘base struct.’’ The wording should be struck.
-> Accepted.

   27.4.2.4:
   ios_traits::to_char_type has an Effects clause which should be
   reworded as a Returns clause.
-> Accepted.
-> See 27.4.2.3 [lib.ios.traits.convert]

   27.4.2.4:
   ios_traits::to_int_type has an Effects clause which should be
   reworded as a Returns clause.
-> Accepted.
-> See 27.4.2.3 [lib.ios.traits.convert]

   27.4.2.4:
   ios_traits::copy has an Effects clause which should be
   reworded as a Returns clause. (It returns src.)
-> Accepted.
-> Add ‘‘Returns: dest’’ as Returns clause, and keep the Effects clause.

   27.4.2.4:
   ios_traits::get_state should be specified to do more than return zero.
   Semantics are inadequate. A pos_type conceptually has three
   components: an off_type (streamsize), an fpos_t, and a state_type
   (mbstate_t, which may be part of fpos_t). It must be possible
   to compose a pos_type from these elements, in various combinations,
   and to decompose them into their three parts.
```

```
-> Accepted.
-> ios_traits and string_char_traits were deprecated in favor of
-> char_traits.  The description of char_traits::get_state is:
-> Returns: A ’state_type’ value which represents the conversion state
-> in the object ’pos’.

   27.4.2.4:
   ios_traits::get_pos should be specified to do more than return
   pos_type(pos). Semantics are inadequate. See comments on get_state.
   above.
-> Rejected.
-> char_traits::get_pos has been deprecated.

   27.4.3:
   ios_base::fill() cannot return int_type because it’s not defined.
   Should be int if fill() is left in ios_base.
-> Accepted.
-> fill() has been moved to basic_ios, and the function signature is
-> ‘‘char_type fill() const’’.

   27.4.3:
   ios_base::precision() and width() should deal in streamsize
   arguments and return values, not int. (Even more precisely,
   they should be moved to basic_ios and have all their types
   changed to traits::streamoff.)
-> Accepted for the first part.
-> ios_base::precision() and width() are using arguments and return
-> values of type streamsize ( see 27.4.3.2 ).
-> streamsize is the correct type to use versus traits::streamoff
-> ( see 27.1.2.5 and 27.1.2.3 ).

   27.4.3.1.6:
   ~Init() should call flush() for wout, werr, and wlog, not just for
   cout, cerr, and clog.
-> Accepted.

   27.4.3.2:
   ios_base::fill(int_type) cannot receive or return int_type
   because it’s not defined. Both should be int if fill(int_type)
   is left in ios_base.
-> Accepted.
-> ios_base::fill(int_type) has been moved to basic_ios, and the function
-> signature is ‘‘char_type fill(char_type ch)’’.

   27.4.3.4:
   ios_base::iword allocates an array of long, not of int.
-> Accepted.

   27.4.3.4:
   ios_base::iword Notes describes a normative limitation on the lifetime
   of a returned reference. It should not be in a Notes clause. It should
   also say that the reference becomes invalid after a copyfmt, or when
   the ios_base object is destroyed.
-> Accepted.

   27.4.3.4:
   ios_base::pword Notes describes a normative limitation on the lifetime
   of a returned reference. It should not be in a Notes clause. It should
   also say that the reference becomes invalid after a copyfmt, or when
   the ios_base object is destroyed.
-> Accepted.

   27.4.3.5:
   Protected constructor ios_base::ios_base() must *not* assign initial
   values to its member objects as indicated in Table 72. That operation
   must be deferred until basic_ios::init is called. Should say here
```

that it does no initialization, then move Table 72 to description of
basic_ios::init (27.4.4.1). Also should emphasize that the object
must be initialized before it is destroyed (thanks to reference
counting of locale objects).
-> Accepted.
-> See 27.4.3.5 [lib.ios.base.cons] and 27.4.4.1 [lib.basic.ios.cons]
-> table 83.

27.4.3.5:
Table 72 shows result of rdstate() for a newly constructed
ios_base object, but that object defines no such member function.
(Will be fixed if table is moved to basic_ios, as proposed.)
-> Accepted.
-> See 27.4.4.1 [lib.basic.ios.cons] table 83.

27.4.4.1:
basic_ios::basic_ios() has next to no semantics. Needs to be

specified:
Effects: Constructs an object of class basic_ios, leaving its
member objects uninitialized. The object *must* be initialized
by calling init(basic_streambuf *sb) before it is destroyed.
-> Accepted.

27.4.4.1:
basic_ios::init(basic_streambuf *sb) has no semantics.
Needs to be specified:
Postconditions: rdbuf() == sb, tie() == 0, ios_base initialized
according to Table 72 (currently in 27.4.3.5).
-> Accepted.

27.4.4.2:
basic_ios::tie is not necessarily synchronized with an *input*
sequence. Can also be used with an output sequence.
-> Accepted.

27.4.4.2:
basic_ios::imbue(const locale&) should call rdbuf()->pubimbue(loc)
only if rdbuf() is not a null pointer. Even better, it should not
call rdbuf()->pubimbue(loc) at all. Changing the locale that
controls stream conversions is best decoupled from changing
the locale that affects numeric formatting, etc. Anyone who knows
how to imbue a proper pair of codecvt facets in a streambuf won’t
mind having to make an explicit call.
-> Rejected.

27.4.4.2:
basic_ios::imbue(const locale&) doesn’t specify what value it returns.
Should say it returns whatever ios_base::imbue(loc) returns.
-> Accepted.

27.4.4.2:
basic_ios::copyfmt should say that both rdbuf() and rdstate() are
left unchanged, not just the latter.
-> Accepted.

27.5.2:
basic_streambuf::sgetn should return streamsize, not int_type
-> Accepted.

27.5.2:
basic_streambuf::sungetc should return int_type, not int
-> Accepted.

27.5.2:
basic_streambuf::sputc should return int_type, not int

-> Accepted.

27.5.2:
basic_streambuf::sputn should return streamsize, not int_type
-> Accepted.

27.5.2.2.3:
In in_avail Returns: gend() should be egptr() and gnext() should be gptr().
-> Accepted.

27.5.2.2.3:
basic_streambuf::sbumpc Returns should not say the function converts *gptr() to char_type. The function returns the int_type result of the call.
-> Editorial.

27.5.2.2.3:
basic_streambuf::sgetc Returns should not say the function converts *gptr() to char_type. The function returns the int_type result of the call.
-> Editorial.

27.5.2.2.3:
basic_streambuf::sgetn should return streamsize, not int.
-> Accepted.

27.5.2.2.4:
basic_streambuf::sungetc should return int_type, not int.
-> Accepted.

27.5.2.2.4:
basic_streambuf::sputc should return int_type, not int.
-> Accepted.

27.5.2.2.5:
basic_streambuf::sputc does not return *pptr(), which points at storage with undefined content. It returns traits::to_int_type(c).
-> Accepted.
-> See 27.5.2.2.5 Put area [lib.streambuf.pub.put]

27.5.2.4.2:
basic_streambuf::sync now requires that buffered input characters ‘‘are restored to the input sequence.’’ This is a change made without approval. It is also difficult, or even impossible, to do so for input streams on some systems, particularly for interactive or pipelined input. The Standard C equivalent of sync leaves input alone. Posix *discards* interactive input. This added requirement is none of the above. It should be struck.
-> Rejected.
-> The change mentioned above is not part of the draft standard.

27.5.2.4.3:
basic_streambuf::showmanyc Returns has been corrupted. The function should return the number of characters that can be read with no fear of an indefinite wait while underflow obtains more characters from the input sequence. traits::eof() is only part of the story. Needs to be restored to the approved intent. (See footnote 218.)
-> Rejected, with no further action.
-> Footnote 12 says: "The intention is not only that the calls will not
-> return eof() but that they will return "immediatly"."

27.5.2.4.3:
basic_streambuf::showmanyc Notes says the function uses traits::eof(). Not necessarily true.
-> Editorial.

-> The Notes should be removed.

27.5.2.4.3:
Footnote 217 is nonsense unless showmany is corrected to showmanyc.
-> Accepted.
-> See footnote 231.

27.5.2.4.3:
basic_streambuf::underflow has two Returns clauses. Should combine them to be comprehensive.
-> Accepted.

27.5.2.4.3:
basic_streambuf::uflow default behavior ‘‘does’’ gbump(1), not gbump(-1). It also returns the value of *gptr() *before* ‘‘doing’’ gbump.
-> Accepted.

27.5.2.4.3:
basic_streambuf::uflow has a nonsense Returns clause. Should be struck.
-> Accepted.

27.5.2.4.4:
basic_streambuf::pbackfail argument should be int_type, not int.
-> Accepted.

27.5.2.4.4:
basic_streambuf::pbackfail Notes begins a sentence with ‘‘Other calls shall.’’ Can’t apply ‘‘shall’’ to user program behavior, by the accepted conformance model.
-> Editorial.

27.6:
<iomanip> synopsis has includes for <istream> and <ostream>, but none of the declarations appear to depend on either of these headers. They should be replaced by an include for <ios>.
-> Accepted.

27.6:
<iomanip> does *not* define a single type smanip. Rather, it defines at least two different types which depend on the type of the function argument. Should probably say that each function returns some unspecified type suitable for inserting into an arbitrary basic_ostream object or extracting from a basic_istream object.
-> Accepted.

27.6.1.1:
basic_istream::seekg(pos_type&) and basic_istream::seekg(off_type&, ios_base::seekdir) should both have const first parameters.
-> Accepted.

27.6.1.1:
basic_istream paragraph 2 says extractors may call rdbuf()->sbumpc(), rdbuf()->sgetc(), or ‘‘other public members of istream except that they do not invoke any virtual members of rdbuf() except uflow().’’ This is a constraint that was never approved. Besides, rdbuf()->sgetc() invokes underflow(), as does uflow() itself, and the example of ipfx in 27.6.1.1.2 uses rdbuf()->sputbackc(). The added constraint should be struck.
-> Accepted.

27.6.1.1:
basic_istream definition, paragraph 4 is confusing, particularly in the light of similar errors in 27.6.2.1 and 27.6.2.4.2 (basic_ostream). It says, ‘‘If one of these called functions throws an exception, then

unless explicitly noted otherwise the input function calls
setstate(badbit) and if badbit is on in exception() rethrows the
exception without completing its actions.’’ But the setstate(badbit)
call may well throw an exception itself, as is repeatedly pointed
out throughout the draft. In that case, it will not return control
to the exception handler in the input function. So it is foolish to
test whether badbit is set -- it can’t possibly be. Besides, I can
find no committee resolution that calls for exceptions() to be
queried in this event.
An alternate reading of this vague sentence implies that setstate
should rethrow the exception, rather than throw ios_base::failure,
as is its custom. But the interface to setstate provides no way
to indicate that such a rethrow should occur, so these putative
semantics cannot be implemented.
The fix is to alter the ending of the sentence to read, ‘‘and if
setstate returns, the function rethrows the exception without
completing its actions.’’ (It is another matter to clarify what
is meant by ‘‘completing its actions.’’)
-> Accepted.
-> section 27.6.1.1 Template class basic_istream [lib.istream]
-> paragraph 4 has been reworded to say:
-> "If one of these called functions throws an exception, then unless
-> explicitly noted otherwise the input function set badbit in error
-> state. If badbit is on in exception(), the input function rethrows
-> the exception without completing its actions, otherwise it does not
-> throw anything and treat as an error."
-> The same change has been made in section 27.6.2.4.2 (basic_ostream).

27.6.1.1.2:

basic_istream::ipfx Notes says the second argument to traits::
is_whitespace is ‘‘const locale *’’. The example that immediately
follows makes clear that it should be ‘‘const ctype<charT>&’’.
-> Rejected.
-> Function is_whitespace was deprecated and the sentry class was
-> accepted, so the examples have been rewritten.

27.6.1.1.2:
Footnote 222 makes an apparently normative statement in
a non-normative context.
-> Accepted.
-> Footnote 222 was removed.

27.6.1.2.1:
basic_istream description is silent on how void* is converted.
Can an implementation use num_get<charT>::get for one of the
integer types? Must it *not* use this facet? Is a version of
get missing in the facet? Needs to be clarified.
-> Rejected.
-> The committee doesn’t really understand the question.

27.6.1.2.1:
Example of call to num_get<charT>::get has nonsense for first two
arguments. Instead of ‘‘(*this, 0, ’’ they should be
be ‘‘(istreambuf_iterator<charT>(rdbuf()),
istreambuf_iterator<charT>(0), ’’
-> Rejected.
-> Appropriate constructors are provided in the class istreambuf_iterator.
-> See section 24.4.3 [lib.istreambuf.iterator]

27.6.1.2.1:
Example of numeric input conversion says ‘‘the conversion occurs
‘as if’ it performed the following code fragment.’’ But that
fragment contains the test ‘‘(TYPE)tmp != tmp’’ which often has
undefined behavior for any value of tmp that might yield a true
result. The test should be replaced by a metastatement such as

‘‘<tmp can be safely converted to TYPE>’’. (Then num_get needs
a version of get for extracting type float to make it possible
to write num_get in portable C++ code.)
-> Accepted.

27.6.1.2.1:
Paragraph 4, last sentence doesn’t make sense. Perhaps ‘‘since the
flexibility it has been...’’ should be, ‘‘since for flexibility
it has been...’’ But I’m not certain. Subsequent sentences are
even more mysterious.
-> Editorial.

27.6.1.2.1:
Use of num_get facets to extract numeric input leaves very unclear
how streambuf exceptions are caught and properly reported. 22.2.2.1.2
makes clear that the num_get::get virtuals call setstate, and hence
can throw exceptions *that should not be caught* within any of the
input functions. (Doing so typically causes the input function to
call setstate(badbit), which is *not* called for as part of reporting
eof or scan failure. On the other hand, the num_get::get virtuals
are called with istreambuf_iterator arguments, whose very constructors
might throw exceptions that need to be caught. And the description of
the num_get::get virtuals is silent on the handling of streambuf
exceptions.
So it seems imperative that the input functions wrap each call to
a num_get::get function in a try block, but doing so will intercept
any exceptions thrown by setstate calls within the num_get::get
functions.
A related problem occurs when eofbit is on in exceptions and the
program attempts to extract a short at the very end of the file.
If num_get::get(..., long) calls setstate, the failure exception
will be thrown before the long value is converted and stored in
the short object, which is *not* the approved behavior.
The best fix I can think of is to have the num_get::get
functions return an ios_base::iostate mask which specifies what
errors the caller should report to setstate. The ios& argument
could be a copy of the actual ios for the stream, but with
exceptions cleared. The num_get::get functions can then continue
to call setstate directly with no fear of throwing an exception.
But all this is getting very messy for such a time critical
operation as numeric input.
-> Accepted.
-> The num_get::get functions are now taking a reference to an
-> ios_base::iostate mask, which specifies the type of error(s)
-> that occurred while extracting the numeric value.
-> See January 1996 WP, section 22.2.2.1 [lib.locale.num.get]

27.6.1.2.2:
basic_istream::operator>>(char_type *) extracts an upper limit of
numeric_limits<int>::max() ‘‘characters.’’ This is a silly and
arbitrary number, just like its predecessor INT_MAX for
characters of type char. A more sensible value is size_t(-1) /
sizeof (char_type) - 1. Could just say ‘‘the size of the largest
array of char_type that can also store the terminating null.’’
basic_istream::operator>>(bool&) has nonsense for its first
two arguments. Should be

```
  istreambuf_iterator<charT, traits>(rdbuf()),
  istreambuf_iterator<charT, traits>(0), etc.
```

-> Accepted for the first part.
-> basic_istream::operator>>(char_type *) now says: ‘‘Otherwise n is
-> the number of elements of the largest array of char_type that can
-> store a terminating eos.’’

-> Rejected for the second part.
-> Appropriate constructors are provided in the class istreambuf_iterator.
-> See section 24.4.3 [lib.istreambuf.iterator]

```
   27.6.1.2.2:
   basic_istream::(bool&  paragraph 3 describes the behavior of
   num_get::get. Description belongs in clause 22.
-> Editorial.

   27.6.1.2.2:
   basic_istream::operator>>(unsigned short&) cannot properly check
   negated inputs. The C Standard is clear that -1 is a valid field,
   yielding 0xffff (for 16-bit shorts). It is equally clear that
     0xffffffff is invalid. But num_get::get(... unsigned long&)
   delivers the same bit pattern for both fields (for 32-bit
   longs), with no way to check the origin. One fix is to have
   the extractor for unsigned short (and possibly for unsigned int)
   pick off any ’-’ flag and do the checking and negating properly,
   but that precludes a user-supplied replacement for the num_get
   facet from doing some other magic. Either the checking rules
   must be weakened over those for Standard C, the interface to
   num_get must be broadened, or the extractor must be permitted
   to do its own negation.
-> Accepted.
-> The class num_get, section 22.2.2.1 has now get member functions
-> for unsigned short, unsigned int, and unsigned long. This solve
-> the problem described above.

   27.6.1.2.2:
   basic_istream::operator>>(basic_streambuf *sb) now says, ‘‘If sb is
   null, calls setstate(badbit).’’ This requirement was added without
   committee approval. It is also inconsistent with the widespread
   convention that badbit should report loss of integrity of the stream
   proper (not some other stream). A null sb should set failbit.
-> Accepted.

   27.6.1.2.2:
   basic_istream::operator>>(basic_streambuf<charT,traits>* sb), last
   line of Effects paragraph 4 can’t happen. Previous sentence says,
   ‘‘If the function inserts no characters, it calls setstate(failbit),
   which may throw ios_base failure. Then the last sentence says,
   ‘‘If failure was due to catching an exception thrown while extracting
   characters from sb and failbit is on in exceptions(), then the caught
   exception is rethrown.’’ But in this case, setstate has already thrown
   ios_base::failure. Besides, I can find no committee resolution
   that calls for exceptions() to be queried in this event.
   In fact, the approved behavior was simply to terminate the copy
   operation if an extractor throws an exception, just as for
   get(basic_streambuf&) in 27.6.1.3. Last sentence should be struck.
-> Accepted.
-> Paragraph 4 now says:
-> "If the function inserts no characters, it calls setstate(failbit),
-> which may throw ios_base::failure. If failure was due to catching
-> an exception thrown while extracting characters from sb and failbit
-> is on in exception(), then the caught exception is rethrown."
-> Concerning the second part of the comment, section 27.6.1.1
-> Template class basic_istream [lib.istream] paragraph 4 clarifies
-> the fact that exceptions() needs to be queried.

   27.6.1.3:
   basic_istream::get(basic_streambuf& sb) Effects says it inserts
   characters ‘‘in the output sequence controlled by rdbuf().’’
   Should be the sequence controlled by sb.
-> Accepted.

   27.6.1.3:
   basic_istream::readsome refers several times to in_avail(), which
   is not defined in the class. All references should be to
   rdbuf()->in_avail(). And the description should specify what
```

happens when rdbuf() is a null pointer. (Presumably sets badbit.)
-> Accepted for the first part.
-> See 27.6.1.3 [lib.istream.unformatted].
-> There is no need to check if rdbuf() is a null pointer, because
-> in this case the istream object should be in bad state, which
-> means that the call to ipfx (or constructing the newly sentry object)
-> will return FALSE.

27.6.1.3:
basic_istream::readsome is now defined for rdbuf()->in_avail() < 0.
The original proposal defined only the special value -1. Otherwise,
it requires that rdbuf()->in_avail >= 0. Should be restored.
-> Accepted.

27.6.1.3:
basic_istream::readsome cannot return read, as stated. That
function has the wrong return type. Should return gcount().
-> Accepted.
-> The Returns: clause says ‘‘The number of characters extracted’’.

27.6.1.3:
basic_istream::putback does *not* call ‘‘rdbuf->sputbackc(c)’’.
It calls ‘‘rdbuf()->sputbackc(c)’’ and then only if rdbuf()
is not null.
-> Editorial.
-> The definition of the function is gone it as been replaced
-> by the definition of basic_istream::unget. It should say:
-> Effects: If rdbuf() is not null, calls rdbuf()->sputbackc(c).
-> If rdbuf() is null, or if sputbackc(c) returns traits::eof(),
-> calls setstate(badbit) (which may throw ios_base::failure (27.4.4.3)).
-> Returns: *this.

27.6.1.3:
basic_istream::unget does *not* call ‘‘rdbuf->sungetc(c)’’.
It calls ‘‘rdbuf()->sungetc(c)’’ and then only if rdbuf()
is not null.
-> Accepted.

27.6.1.3:
basic_istream::sync describes what happens when rdbuf()->pubsync()
returns traits::eof(), but that can’t happen in general because
pubsync returns an int, not an int_type. This is an unauthorized,
and ill-advised, change from the original EOF. Return value should
also be EOF.
-> Accepted.
-> The return value of rdbuf()->pubsync() is now -1 on failure, and
-> the description of basic_istream::sync has been changed to:
-> ‘‘... calls rdbuf()->pubsync() and, if that function returns -1 ...’’

27.6.1.3:
basic_istream::sync Notes says the function uses traits::eof().
Obviously it doesn’t, as described above. Clause should be struck.
-> Accepted.

27.6.2.1:
basic_ostream::seekp(pos_type&) and basic_ostream::seekp(off_type&,
ios_base::seekdir) should both have const first parameters.
-> The first parameters are now passed by value.

27.6.2.1:
basic_ostream definition, last line of paragraph 2 can’t happen.
It says, ‘‘If the called
function throws an exception, the output function calls
setstate(badbit), which may throw ios_base::failure, and if badbit
is on in exceptions() rethrows the exception.’’ But in this case,
setstate has already thrown ios_base::failure. Besides, I can find

no committee resolution that calls for exceptions() to be queried
in this event. Last sentence should end with, ‘‘and if setstate
returns, the function rethrows the exception.’’
-> Accepted.
-> section 27.6.2.1 Template class basic_ostream [lib.ostream]
-> paragraph 3 has been reworded to say:
-> "If one of these called functions throws an exception, then unless
-> explicitly noted otherwise the output function set badbit in error state.
-> If badbit is on in exception(), the output function rethrows the exception
-> without completing its actions, otherwise it does not throw anything
-> and treat as an error."

27.6.1.2.1:
Use of num_put facets to insert numeric output leaves very unclear
how output failure is reported. Only the ostreambuf_iterator knows
when such a failure occurs. If it throws an exception, the calling
code in basic_ostreambuf is obliged to call setstate(badbit) and
rethrow the exception, which is *not* the approved behavior
for failure of a streambuf primitive.
Possible fixes are: have ostreambuf_iterator report a specific
type of exception, have ostreambuf_iterator remember a failure
for later testing, or give up on restoring past behavior. Something
must be done in this area, however.
-> Accepted.
-> A member function ‘‘bool failed() const throw()’’ has been added
-> to the ostreambuf_iterator template class. This function returns
-> true if in any prior use of member operator=, the call to
-> sbuf_->sputc() returned traits::eof; or false otherwise.
-> See 24.4.4.2 [lib.ostreambuf.iter.ops].

27.6.2.4.1:
Table 76 is mistitled. It is not just about floating-point conversions.
-> Editorial

27.6.2.4.1:
Table 77 has an unauthorized change of rules for determining fill
padding. It gives the three defined states of flags() & adjustfield as
left, internal, and otherwise. It should be right, internal, and
otherwise. Needs to be restored to the earlier (approved) logic.
-> Rejected.
-> The current status is clear and reflect the historical behavior.

27.6.2.4.2:
basic_ostream<<operator<<(bool) should use ostreambuf_iterator,
not istreambuf_iterator. The first argument is also wrong in the
call to num_put::put.
-> Editorial.
-> It should be ostreambuf_iterator.

27.6.2.4.2:
basic_ostream::operator<<(basic_streambuf *sb) says nothing about
sb being null, unlike the corresponding extractor (27.6.1.2.2).
Should either leave both undefined or say both set failbit.
-> Accepted.
-> If sb is null, the function basic_ostream::operator<<(basic_streambuf *sb)
-> calls setstate(badbit) (the corresponding extractor does too).

27.6.2.4:
basic_ostream::operator<<(streambuf *) says nothing about the failure
indication when ‘‘inserting in the output sequence fails’’. Should
say the function sets badbit.
-> Accepted.
-> If sb is null, the function basic_ostream::operator<<(basic_streambuf *sb)
-> calls setstate(badbit) (the corresponding extractor does too).
-> See 27.6.2.4.2 [lib.ostream.inserters].

27.6.2.4.2:
basic_ostream::operator<<(basic_streambuf<charT,traits>* sb), last line of Effects paragraph 2 can’t happen. Previous sentence says that if ‘‘an exception was thrown while extracting a character, it calls setstate(failbit) (which may throw ios_base::failure).’’ Then the last sentence says, ‘‘If an exception was thrown while extracting a character and failbit is on in exceptions() the caught exception is rethrown.’’ But in this case, setstate has already thrown ios_base::failure. Besides, I can find no committee resolution that calls for exceptions() to be queried in this event. And an earlier sentence says unconditionally that the exception is rethrown. Last sentence should be struck.

-> Accepted.
-> Paragraph 3 now says:
-> "If the function inserts no characters, it calls setstate(failbit),
-> which may throw ios_base::failure. If an exception was thrown while
-> extracting a character the function set failbit in error state, and
-> if failbit is on in exceptions(), then the caught exception is
-> rethrown."
-> Concerning the second part of the comment, section 27.6.2.1
-> Template class basic_ostream [lib.ostream] paragraph 3 clarifies
-> the fact that exceptions() needs to be queried.

27.6.2.5:
basic_ostream::flush can’t test for a return of traits::eof() from basic_streambuf::pubsync. It tests for EOF.

-> Accepted.
-> The return value of rdbuf()->pubsync() is now -1 on failure, and
-> the description of basic_istream::flush has been changed to:
-> ‘‘... calls rdbuf()->pubsync(). If that function returns -1 ...’’

27.6.3:
‘‘headir’’ should be ‘‘header.’’

-> Accepted.

27.6.3:
<iomanip> does *not* define a single type smanip. Rather, it defines at least two different types which depend on the type of the function argument. Should probably say that each function returns some unspecified type suitable for inserting into an arbitrary basic_ostream object or extracting from a basic_istream object.

-> Accepted.

27.7:
<sstream> synopsis refers to the nonsense class int_charT_traits. It should be ios_traits.

-> Accepted.

27.7:
Table 77 (<cstdlib> synopsis) is out of place in the middle of the presentation of <sstream>.

-> Accepted.

27.7.1:
basic_stringbuf::basic_stringbuf(basic_string, openmode) Effects repeats the phrase ‘‘initializing the base class with basic_streambuf().’’ Strike the repetition.

-> Accepted.
-> See 27.7.1.1 [lib.stringbuf.cons].

27.7.1:
basic_stringbuf::basic_stringbuf(basic_string, openmode) Postconditions requires that str() == str. This is true only if which has in set. Condition should be restated.

-> Rejected.

```
-> This is true in any case.

   27.7.1:
   Table 78 describes calls to setg and setp with string
   arguments, for which no signature exists. Needs to be recast.
-> Accepted.
-> Table has been removed.

   27.7.1:
   basic_stringbuf::str(basic_string s) Postconditions
   requires that str() == s. This is true only if which had in set
   at construction time. Condition should be restated.
-> Rejected.
-> This is true in any case.

   27.7.1.2:
   Table 80 describes calls to setg and setp with string
   arguments, for which no signature exists. Needs to be recast.
-> Accepted.
-> Table has been removed.

   27.7.1.3:
   basic_stringbuf::underflow Returns should return int_type(*gptr()),
   not char_type(*gptr()).
-> Accepted.

   27.7.1.3:
   basic_stringbuf::pbackfail returns c (which is int_type) in first case,
   char_type(c) in second case. Both cases should be c.
-> Accepted.

   27.7.1.3:
   basic_stringbuf::pbackfail supposedly returns c when c == eof().
   Should return traits::not_eof(c).
-> Accepted.

   27.7.1.3:
   basic_stringbuf::seekpos paragraph 4 has ‘‘positionedif’’ run together.
-> Accepted.

   27.8.1.1:
   basic_filebuf paragraph 3 talks about a file being ‘‘open for reading
   or for update,’’ and later ‘‘open for writing or for update.’’
   But ‘‘open for update’’ is not a defined term. Should be struck
   in both cases.
-> Accepted.

   27.8.1.3:
   basic_filebuf::is_open allegedly tests whether ‘‘the associated file
   is available and open.’’ No definition exists for ‘‘available.’’
   The term should be struck.
-> Accepted.

   27.8.1.3:
   basic_filebuf::open Effects says the function fails if is_open()
   is initially false. Should be if initially true.
-> Accepted.

   27.8.1.3:
   basic_filebuf::open Effects says the function calls the default
   constructor for basic_streambuf. This is nonsense. Should say,
   at most, that it initializes the basic_filebuf as needed, and
   then only after it succeeds in opening the file.
-> Accepted.

   27.8.1.3:
```

Table 83 has a duplicate entry for file open mode ‘‘in | out’’.

-> Accepted.
-> Removed one entry.

27.8.1.4:
filebuf::showmanyc (and several overriden virtual functions that follow) have a Requires clause that says ‘‘is_open == true.’’ The behavior of all these functions should be well defined in that event, however. Typically, the functions all fail. The Requires clause should be struck in all cases.

-> Accepted.

27.8.1.4:
filebuf::showmanyc Effects says the function ‘‘behaves the same as basic_streambuf::showmanyc.’’ The description adds nothing and should be struck.

-> Editorial.

27.8.1.4:
basic_filebuf::underflow effects shows arguments to convert as ‘‘st,from_buf,from_buf+FSIZE,from_end,to_buf, to_buf+to_size, to_end’’. st should be declared as an object of type state_type, and n should be defined as the number of characters read into from_buf. Then the arguments should be ‘‘st, from_buf, from_buf + n, from_end, to_buf, to_buf + TSIZE, to_end’’. Also, template parameter should be ‘‘traits::state_type,’’ not ‘‘ios_traits::state_type.’’

-> Accepted.

27.8.1.4:
basic_filebuf::underflow is defined unequivocally as the function that calls codecvt, but there are performance advantages to having this conversion actually performed in uflow. If the specification cannot be broadened sufficiently to allow either function to do the translation, then uflow loses its last rationale for being added in the first place. Either the extra latitude should be granted implementors or uflow should be removed from basic_streambuf and all its derivatives.

-> Accepted, both underflow and uflow are allowed to call codecvt.

27.8.1.4:
basic_filebuf::pbackfail(traits::eof()) used to return a value other than eof() if the function succeeded in backing up the input. Now the relevant Returns clause says the function returns the metacharacter c, which is indistinguishable from a failure return. This is an unapproved change. Should probably say the function returns traits::not_eof(c).

-> Accepted.

27.8.1.4:
basic_filebuf::pbackfail Notes now says ‘‘if is_open() is false, the function always fails.’’ This is an unapproved change. The older wording should be restored.

-> Editorial

27.8.1.4:
basic_filebuf::pbackfail Notes now says ‘‘the function does not put back a character directly to the input sequence.’’ This is an unapproved change and not in keeping with widespread practice. The older wording should be restored.

-> Editorial

27.8.1.4:
basic_filebuf::pbackfail has a Default behavior clause. Should be struck.

-> Accepted.

27.8.1.4:
basic_filebuf::overflow effects shows arguments to convert as ‘‘st,b(),p(),end,buf,buf+BSIZE,ebuf’’. st should be declared as an object of type state_type. Then the arguments should be ‘‘st, b, p, end, buf, buf + BSIZE, ebuf’’. Also, template parameter should be ‘‘traits::state_type,’’ not ‘‘ios_traits::state_type.’’
-> Accepted.

27.8.1.4:
basic_filebuf::overflow doesn’t say what it returns on success. Should say it returns c.
-> Accepted.

27.8.1.4:
basic_filebuf::setbuf has no semantics. Needs to be supplied.
-> Accepted.

27.8.1.4:
basic_filebuf::seekoff Effects is an interesting exercise in creative writing. It should simply state that if the stream is opened as a text file or has state-dependent conversions, the only permissible seeks are with zero offset relative to the beginning or current position of the file. (How to determine that predicate is another matter -- should state for codecvt that even a request to convert zero characters will return noconv.) Otherwise, behavior is largely the same as for basic_stringstream, from whence the words should be cribbed. The problem of saving the stream state in a traits::pos_type object remains unsolved. The primitives described for ios_traits are inadequate.
-> Accepted.

27.8.1.4:
basic_filebuf::seekpos has no semantics. Needs to be supplied.
-> Accepted.

27.8.1.4:
basic_filebuf::sync has no semantics. Needs to be supplied.
-> Accepted.

27.8.1.4:
basic_filebuf::imbue has silly semantics. Whether or not sync() succeeds has little bearing on whether you can safely change the working codecvt facet. The most sensible thing is to establish this facet at construction. (Then pubimbue and imbue can be scrubbed completely.) Next best is while is_open() is false. (Then imbue can be scrubbed, since it has nothing to do.) Next best is to permit any imbue that doesn’t change the facet or is at beginning of file. Next best is to permit change of facet any time provided either the current or new facet does not mandate state-dependent conversions. (See comments under seekoff.)
-> Rejected.

27.8.1.7:
basic_filebuf::rdbuf should not have explicit qualifier.
-> Accepted.

27.8.1.9:
basic_ofstream::basic_ofstream(const char *s, openmode mode = out) has wrong default second argument. It should be ‘out | trunc’, the same as for basic_ofstream::open (in the definition at least).
-> Accepted in principle.
-> The default second argument is still out, but out is really equivalentto
-> to out | trunc.

27.8.1.10:
basic_ofstream::open(const char *s, openmode mode = out)

```
   has wrong default second argument. It should be ‘out | trunc’, the
   same as for basic_ofstream::open in the definition.
-> Accepted in principle.
-> The default second argument is still out, but out is really equivalentto
-> to out | trunc.

   27.8.2:
   <cstdio> synopsis has two copies of tmpfile and vprintf,
   no vfprintf or putchar.
-> Accepted.

   27.8.2:
   <cwchar> summary should also list the type wchar_t. Aside from the
   addition of the (incomplete) type struct tm, this table 84 is
   identical to table 44 in 21.2. It is not clear what purpose either
   table serves; it is less clear what purpose is served by repeating
   the table.
-> Accepted.
-> Table has been removed.

   27.8.2:
   See Also reference for <wchar> should be 7.13.2, not 4.6.2.
-> Accepted.
-> Table has been removed.

   D.2:
   Functions overloaded on io_state, open_mode, and seek_dir ‘‘call
   the corresponding member function.’’ But no hint is given as to
   what constitutes ‘‘correspondence.’’
-> Rejected.
-> We don’t understand the comment.

   D.3.1.3:
   strstreambuf::overflow has numerous references to ‘‘eof()’’, which
   no longer exists. All should be changed to EOF.
-> Accepted.
-> See D.6.1.3 [depr.strstreambuf.virtuals].

   D.3.1.3:
   strstreambuf::overflow says it returns ‘‘(char)c’’ sometimes,
   but this can pun with EOF if char has a signed representation.
   More accurate to say it returns (unsigned char)c.
-> Accepted.
-> See D.6.1.3 [depr.strstreambuf.virtuals].

   D.3.1.3:
   strstreambuf::pbackfail says it returns ‘‘(char)c’’ sometimes,
   but this can pun with EOF if char has a signed representation.
   More accurate to say it returns (unsigned char)c.
-> Accepted.

   D.3.1.3:
   strstreambuf::pbackfail says it returns ‘‘(char)c’’ when c == EOF,
   but this can pun with EOF if char has a signed representation.
   More accurate to say it returns something other than EOF.
-> Accepted.

   D.3.1.3:
   strstreambuf::pbackfail twice says it returns EOF to indicate
   failure. Once is enough.
-> Accepted.
-> See D.6.1.3 [depr.strstreambuf.virtuals].

   D.3.1.3:
   strstreambuf::setbuf has a Default behavior clause, which is not
   appropriate for a derived stream buffer. It also adds nothing to
   the definition in the base class. The entire description should
   be struck.
-> Accepted.

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
21- (continued)
    Additional comments from WG14
    Received by email
    email address: pjp@plauger.com
    Was comment T25 in the post-Monterey mailing document.

  3.2-8
  The acronym ‘‘ODR’’ has not been defined.  Also, it doesn’t
  make sense when expanded: ‘‘one definition rule rule’’.
-> 1st sentence: Accepted.
->               See 3.2[basic.def.odr]
-> 2nd sentence: Editorial.

  3.7.3.2-5
  Footnote #20 refers to ‘‘architectures’’.  Other places
  refer to ‘‘machines’’.  They should all refer to
  ‘‘implementations’’.
-> Editorial.

  3.8
  It is not clear what object ‘‘use’’ or ‘‘reuse’’ is.
-> Accepted.
-> See 3.2 [basic.def.odr] for the description of the meaning of "use"
-> for this International Standard.

  3.8-2
  The acronym ‘‘POD’’ has not been defined.  In general, each
  section should have ‘‘forward references’’, like the C
  Standard.
-> Editorial.

  3.8-3
  Awkward wording: ‘‘In particular, except as noted’’.
-> Editorial.

  3.9-2
  How can I tell that the ‘‘copy operation is well-defined’’?
  It is not clear what ‘‘well-defined’’ means here or if I can
  test for it.
-> Editorial.
-> The result of copying an object into an array of characters will
-> be described without discussing what a "well-defined copy operation"
-> means.

  3.9-4 The ‘‘value’’ of an object of type T is not
  necessarily based upon its bit represention, especially when
  the class is a handle to other data.  The ‘‘value’’ in this
  case would depend upon how the "==" operator is overloaded.
  Even if its ‘‘representation value’’ is somehow defined,
  what purpose does it serve?  Where else is this used in the
  draft?
-> Editorial.
-> The value representation of a scalar type is based on its bit
-> representation. The concept of value representation is necessary
-> to describe the correspondence between the representation of signed
-> and unsigned integral types that an implementation must support.
-> See 3.9.1 [basic.fundamental]

  3.9.1-1
  Remove ‘‘there are several fundamental types’’.
-> Editorial.
```

3.9.1-2
Use different wording than ‘‘take on’’.
-> Editorial.

3.9.1-4
Don’t refer to ‘‘machine architecture’’. See C Standard wording.
-> Editorial.

3.9.1-6
Change ‘‘laws of arithmetic modulo 2^N’’ to C Standard wording.
-> Editorial.

3.9.1-8
Reword ‘‘although values of type bool generally behave ...’’.
-> Editorial.

3.9.1-8
Reword ‘‘successfully be stored’’.
-> Editorial.

3.9.2-1
Reword ‘‘There is a conceptually infinite ...’’. Remove the words ‘‘conceptually’’ and ‘‘infinite’’.
-> Editorial.

3.9.3-1
The definition of "volatile" is missing. It isn’t in subclause 1.8 or 7.1.5.1. See the C definition: ‘‘An object that has volatile-qualified type may be modified in ways unknown to the implementation ...’’.
-> It is defined in 1.8 para 7.

3.9.3-5
Change ‘‘In this document’’ to ‘‘In this International Standard’’.
-> Editorial.

3.10-2
Footnote #30: Clarify ‘‘... in some sense refer to an object’’.
-> Editorial.

4.1-1
Reword ‘‘necessitates ... is ill-formed’’ to use ‘‘shall’’ or ‘‘shall not’’.
-> Rejected.
-> Other comments indicate that the other way around is preferred.

4.1-1
Footnote #31. Need proper reference to Standard C.
-> Editorial.

4.3-1
Footnote #32. Reword ‘‘there is no way ...’’.
-> Editorial.

4.5-1
Reword ‘‘can’’ with ‘‘shall’.
-> Editorial.

4.4-4
The sentence ‘‘That is, the member aspect ...’’ should be a

footnote.
-> Editorial.

4.5-2
Reword ‘‘can’’ with ‘‘shall’.
-> Editorial.

4.5-3
Reword ‘‘can’’ with ‘‘shall’.
-> Editorial.

4.5-3
Footnote #34: Reword ‘‘If the bit-field is larger yet, ...’’
using ‘‘shall’’ and ‘‘shall not’’. If this is a constraint,
it shouldn’t be a footnote.
-> Editorial.

4.5-4
Reword ‘‘can’’ with ‘‘shall’.
-> Editorial.

4.7-2
What is the difference here between a note and a footnote?
This should be a footnote.
-> Editorial.

5.2.2-7
A bit-field is not a type.
-> Editorial.

5.2.2-7
Change ‘‘unsigned’’ to ‘‘unsigned int’’ ‘‘int, unsigned int,
...’’.
-> Editorial.

5.2.6-1
Reword ‘‘... shall not cast away constness’’ in more precise
terms. See 5.2.9-2’s reference to 5.2.10.
-> Rejected.
-> Cast away constness is already defined in 5.2.10 and 5.2.6-1 refers
-> to 5.2.10.

5.2.7-1
Footnote #43: Does "*(p)" meet this requirement?
-> Editorial.
-> "*(p)" is now also included in the list.

5.2.7-1
Shouldn’t ‘‘then the pointer shall either be zero’’ be
‘‘then the pointer shall either be the null pointer value’’?
-> Editorial.

5.2.8-1
Reword ‘‘... shall not cast away constness’’ in more precise
terms. See 5.2.9-2’s reference to 5.2.10.
-> Rejected.
-> Cast away constness is already defined in 5.2.10 and 5.2.6-1 refers
-> to 5.2.10.

5.2.10
This section is hard to understand, especially the rules
defining casting away constness.
-> Editorial.

5.2.10-4
Does ‘‘implicit conversion’’ here refer to subclause 4.10,

pointer conversion?
-> Editorial.

5.2.10-7
The ‘‘[Note:’’ doesn’t have a closing ‘‘]’’. This appears
to be a formatting issue throughout the document.
-> Editorial.

5.2.10-7
Where are ‘‘multi-level’ and ‘‘mixed object’’ defined?
-> Editorial.

5.3.1-2
How do the ‘‘implicit conversions’’ here relate to the
‘‘implicit conversions’’ of 4.10 or 5.2.10? The term
‘‘implicit conversion’’ should be defined explicitly.
-> Editorial.
-> It refer to implicit conversions defined in 4.3.
-> See beginning of clause 4 [conv] for definition of implicit
-> conversion.

5.3.5-2
What is ‘‘(_class.conv,fct_)’’?
-> Editorial.

5.7-6
How is C compatibility maintained if a different header is
required for C++ for "ptrdiff_t"?
-> Accepted.
-> Annex D is normative. Subclause D.4 makes it clear that <stddef.h>
-> is a required header in a conforming C++ implementation and has the
-> correct semantics for this issue.

5.8-1
Why isn’t the C wording used here, especially the semantics
for unsigned integers?
-> Editorial.

5.9-2
Change ‘‘The usual arithmetic conversions’’ to ‘‘The
standard integral promotions (4.5)’’.
-> Rejected.
-> This doesn’t cover operands of type long appropriately.

5.10-1
It is not clear ‘‘.... have the same semantic restrictions,
conversions’’ what this points to. The wording should be
repeated or the reference to the associated text should be
clearer.
-> Rejected.
-> The committee found that the wording provided was good enough.

5.11-1
See 5.9-2 above on ‘‘usual arithmetic conversions’’.
-> Rejected.
-> This doesn’t cover operands of type long appropriately.

5.12-1
See 5.9-2 above on ‘‘usual arithmetic conversions’’.
-> Rejected.
-> This doesn’t cover operands of type long appropriately.

5.13-1
See 5.9-2 above on ‘‘usual arithmetic conversions’’.
-> Rejected.
-> This doesn’t cover operands of type long appropriately.

5.16-1
What was the grammar changed from C? The expression after the colon should be ‘‘conditional-expression’’.
-> Rejected.
-> This is required to support "throw-expression".

5.16-2
If both the second and third expression are throw-expressions, then what it the type of the result? According to 15-1, the resultant type of the throw expression is "void". Thus, the resultant type of "?:" is "void". This should be made clear here.
-> Editorial.

5.17-4
Change ‘‘the user’’ to ‘‘the program’’. Change all other uses of ‘‘the user’’ to something else in the rest of the draft.
-> Editorial.

7.1.2-2
Change ‘‘hint’’ wording to use C wording similar to "register" keyword.
-> Editorial.

7.1.3-5
Reword ‘‘... The typedef-name is still only a synonym for the dummy name and shall not be used where a true class name is required’’. Either ‘‘dummy name’’ should be defined or removed in this paragraph (used several times). What is a ‘‘true class name’’? If the dummy name is not specified, why do I care about it for ‘‘linkage purposes’’?
-> Editorial.

7.1.5.1-3
The draft says ‘‘CV-qualifiers are supported by the type system so that they cannot be subverted without casting’’, but it doesn’t specify that the behavior is undefined (C says it’s undefined).
-> Rejected.
-> Paragraph 4 already says this.

7.1.5.1-7
This should not be a note, but part of the standard. The same wording should be extracted from the C Standard.
-> Rejected.
-> The normative text is in 1.8 [intro.execution].

7.2-1
Reword or remove ‘‘... not gratuitously larger than int’’. If it’s implementation-defined, then say so.
-> Editorial.

7.2-6
The possibility that the compiler generates bit fields for enumerators means that it would not be object, i.e., not addressible. Since it is impossible to determine whether or not the address is taken (the "enum" might have its address taken in some other translation unit), having the compiler decide bit-field or not won’t work. If "enum" bit fields are to be supported, they should use some *obvious* syntax. Also, implicit bit fields would be incompatible with C programs.
-> Editorial.

7.3.1.2-1

If an unnamed namespace has a unique identifier that cannot be determined and cannot be linked to (even if there is external linkage -- see footnote 54), then an unnamed namespace is equivalent to "static" at file scope. The draft should change the wording to be the equivalent of "static" at file scope (a feature all linkers can provide) rather than the requirement that a unique name be created (difficult for linkers and *very* difficult for externally developed libraries). If an implementation creates something that I cannot detect then it doesn’t exist.

-> Editorial.

7.3.3-6

Remove ‘‘... (as ever)’’.

-> Editorial.

7.3.4-4

What is a ‘‘using-directive lattice’’? Where is it defined?

-> Editorial.

7.5-3

If a function has more than one linkage specification (say in different translation units) a diagnostic is required. However, the compiler and/or linker may not be able to detect this even with type-safe linking (type-safe linking doesn’t imply that the function call mechanisms are the same).

-> Editorial.

7.5-6

Reword ‘‘There is no way ...’’.

-> Editorial.

7.5-8

Change: ‘‘FORTRAN’’ is now properly spelled ‘‘Fortran’’ according to the Fortran Standard. It would be better if C++ specified that the linkage string is case insensitive and is in the ISO 646 subset. Since the linkage is all implementation-defined anyway, the linker (and the compiler) will know the true way (possibly, case-sensitive) of spelling the linkage name.

-> Editorial.
-> The WP was changed to indicate that it is implementation-defined
-> which string-literal can be used in a linkage specification and
-> whether or not the string-literal is case sensitive.

8-3

Footnote 55: Reword ‘‘A declaration with several declarations is usually equivalent’’ to remove the word ‘‘usually’’.

-> Rejected.
-> "Usually" is used because there are exceptions where this is not the
-> case. The exceptions are later listed in the footnote.

8.2-1

Reword ‘‘In that context, it surfaces ...’’ to remove ‘‘surfaces’’.

-> Editorial.

8.2-1

Remove ‘‘Just as for statements’’. The reference to which section is unclear. The level of semantics to drag in are not specified.

-> Editorial.

8.2-2
Reword ‘‘... can occur in many different contexts ...’’ to
remove the word ‘‘many’’.
-> Editorial.

8.2-3
Number the 4 examples.
-> Rejected.
-> When the draft provides multiple consecutive examples, there are not
-> numbered. It seems inappropriate to do it here.

8.3-2
Change ‘‘inductive’’ to ‘‘recursive’’.
-> Editorial.

8.3.1-3
Reword ‘‘volatile specifiers are handled similarly.’’.
Similar to what?
-> Editorial.

8.3.2-4
Reword ‘‘In particular, null references are prohibited; no
diagnostic is required.’’. What does ‘‘prohibited’’ mean?
Do you mean ‘‘undefined’’ here?
-> Editorial.

8.3.4-1
Typo: ‘‘,T’’ ==> ‘‘T,’’, ‘‘.T’’ ==> ‘‘T.’’
-> Rejected.
-> This is a style the editor has deliberately chosen and that is
-> consistent throughout the draft.

8.3.4-2
Replace with C Standard wording. The C wording is clearer
and shorter: ‘‘An array type describes a contiguously
allocated nonempty set of objects with a particular member
object type, called the element type.’’ (there is a footnote
attached that explains imcomplete types are disallowed).
-> Editorial.

8.3.4-3
Reword ‘‘When several array of specifications are adjacent’’
to remove or define the word ‘‘adjacent’’.
-> Editorial.

8.3.4-4
Reword ‘‘... (say, N) ...’’ to remove the ‘‘say, N’’.
Possibly, start the sentence ‘‘If N is the number of initial
elements, ...’’.
-> Editorial.

8.3.5-2
Is using the C "<cstdarg>" the same as "<stdarg.h>"? If
not, then the code will be incompatible.
-> Annex D is normative. Subclause D.4 makes it clear that <stdarg.h>
-> is a required header in a conforming C++ implementation and has the
-> correct semantics for this issue.

8.3.5-4
Remove ‘‘Functions shall not return arrays ... [to the end
of the paragraph]’’. This restriction has been stated
elsewhere.
-> Rejected.

8.3.5-5
Remove this paragraph. It has been stated elsewhere.

-> Rejected.

8.3.6-6
Change ‘‘out-of-line function’’ to ‘‘non-inline function’’.
-> Editorial.

8.3.6-9
Previously, the order of evaluation of function arguments was ‘‘unspecified’’. Here it’s ‘‘implementation-defined’’. Which is it?
-> Editorial.
-> unspecified, it is.

8.5-6
Need forward reference to ‘‘POD’’. It has not yet been defined. The restriction on arrays has been stated elsewhere.
-> Editorial.

18.1-3
If the C standard header "<stddef.h>" is used, do I get the same result as including "<cstddef>"? Subclause D.1 refers to compatibility, but this isn’t clear. Also, this paragraph should refer to D.1.
-> Accepted.
-> Annex D is normative. Subclause D.4 makes it clear that <stddef.h>
-> is a required header in a conforming C++ implementation and has the
-> correct semantics for this issue.

D.1-1
The names should be the same for C headers in C++. There should be no renaming. This breaks C code to rename them, especially when both should behave the same. Rather than the name "cstdlib", is should be "stdlib.h". Since every C compiler already supports this, C++ can’t claim defective linkers, filesystems, and so on. This is a gratuitous difference that just breaks working code.
-> Accepted.
-> See above (18.1-3).

22- Comments from Bob Kline
Received by email
email address: bob_kline@stream.com
Was comment T26 in the post-Monterey mailing document.

2.9.2 [lex.ccon]: A change has been made to octal escape sequences, which until now has always been a backslash followed by one, two, or three octal digits. The latest version appears to place no limit on the number of digits which can make up an octal escape sequence.
-> Accepted.

5.3.1 [expr.unary.op]: The sentence following the third example ("Neither does qualified-id, ...") is outside the square brackets enclosing the example, but continues the thought begun within the brackets. Text containing bracketed portions should read intelligibly if the bracketed material is omitted.
-> Editorial.

5.3.5 [expr.delete]: Footnote 46: "... deleted using a point ...." Should read "... deleted using a pointer ..."
-> Editorial.

7.1.5.1 [dcl.type.cv] Paragraph 2: "... for a const object of type T, if T is a class with a user-declared default constructor, the constructor for T is called, ..." This language implies that for the following code fragment

```
  class T {
  public:
      T();
      T(int);
      ....
  };
  const T t(1);
```

the default constructor would be called for t. Surely this is not what the committee intended.

-> Editorial.

7.1.5.1 [dcl.type.cv]: In paragraph 6 the semicolon is missing after definition of class Y.

-> Editorial.

8.5.1 [dcl.init.aggr]: Two pointers to footnote 62 appear: one in paragraph 1 and the other in paragraph 4. Only the one in paragraph 4 seems appropriate. Is there a footnote missing for paragraph 1?

-> Editorial.

9.3 [class.scope0]: Paragraph 1, rule 2: use the same font for S in both places.

-> Editorial.

10.3 [class.virtual]: Paragraph 4: "Even if destructors are not inherited, a destructor in a derived class overrides a base class destructor declared virtual; ...." This should read "Even though destructors ...."

-> Editorial.

12.4 [class.dtor]: Paragraph 10 gives examples of placement of an object of class X at a buffer created as

```
  static char buf[sizeof(X)];
```

Is the alignment of a static array of char guaranteed to satisfy the alignment requirements of an arbitrary class X?

-> Editorial.

12.7 [class.cdtor]: Example in paragraph 2: why is ‘D((C*)this,’ commented out?

-> Because there is already a constructor for D specified in the
-> ctor-initializer for E. The comment shows how the first constructor
-> call could be rewritten to give the code well-defined behavior.

21.1.1.4 [lib.string.cons]: Why do some constructor specifications indicate what is thrown under exceptional conditions and others not? Also, for basic_string(const chrT*), shouldn’t length_error be thrown if n >= npos (draft says ‘if n == npos’)? Also, signatures given in the tables do not always match the prototypes for the corresponding constructors; (e.g.: table 42: basic_string(size_type, charT, ...) vs. (charT, size_type); and table 43 uses identifiers instead of the type names). Also, under table 43, "Notes: see Table ___, ...": the table reference is incomplete. As a general comment, some of the library chapters appear to have received much less thorough editorial scrutiny than the chapters for the language proper.

-> Since npos is the largest possible value for size_type, it is not
-> possible for for any n to have a value: n > npos. Signatures in
-> tables are editorial.

21.1.1.6 [lib.string.capacity]: "size_type max_size() const; Returns: The maximum size of the string." This description does not convey enough information. Does this mean the maximum value that can be given to resize()? Does it reflect space for a terminating NUL? Does it reflect the amount of space currently allocated? (If so, how would this differ from capacity()?)

-> The description is made as precisely as it can be made. The member
-> max_size() can be made to return the maximum size of a string as
-> determined by the implementation. It is not necessarily the maximum
-> value that can be given to resize(). It does not necessarily
-> reflect space for a terminating NUL. Nor does it necessarily
-> reflect the amount of space currently allocated.

21.1.1.10.1:

```
 template<class charT, class traits, class Allocator>
   basic_string<charT,traits,Allocator>
     operator+(const basic_string<charT,traits,Allocator>& lhs,
               const basic_string<charT,traits,Allocator>& rhs);
 Returns lhs.append(rhs).
```

If you look back as 21.1.1.8.2, basic_string::append, you see that basic_string::append() is a non-const member function, which means that it can’t be used to implement operator+(), for which lhs is a const object. It wouldn’t make sense anyway, because that would duplicate the functionality of basic_string::operator+= (see 21.1.1.8.1). Don’t we want operator+ to create an entirely new object, not just append to lhs?

-> Accepted.

27.1.1 [lib.iostreams.definitions]: Paragraph 1, last entry: "A repositional stream, can seek to only the position where we previously encountered. On the other hand, an arbitrary-positional stream can seek to any position within the length of the stream. Every arbitrary-positional stream is repositional."

- The comma after "repositional stream" needs to be deleted.
- The third sentence contradicts the first as worded.
- The colloquial and awkward tone ("where we previously encountered") is inconsistent with the more impersonal and precise language of the rest of the standard.

-> Editorial.

27.1.2 [lib.iostreams.type.reqmts]: Last sentence: "... expects to the character container class." should read "... expects of the character container class."

-> Editorial.

27.1.2.1 [lib.iostreams.char.t]: "provides the definitions common between ..." should read "provides the definitions common to ...."

-> Accepted.
-> Now it says ‘‘The collection of these functions can be regarded as
-> the collection of the common definitions for the implementation of
-> the character container class’’.

27.1.2.3 [lib.iostreams.off.t]: footnote 207: "It is usually a synonym for one of the signed basic integral types whose representation at least as many bits as type long." Should read "... whose representation is at least as many bits as type long."

-> Editorial.

27.1.2.3 [lib.iostreams.off.t]: Paragraph 4: "[Type OFF_T is c]onvertible to type POS_T. But no validity of the resulting POS_T value is ensured, whether or not the OFF_T value is valid." Of what use is the conversion, then?

-> Rejected.
-> Users writing their own streambuf might need the conversion from
-> type OFF_T to type POS_T.

27.1.2.4 [lib.iostreams.pos.t]: Paragraph 3’s sentence is awkwardly worded ("... previous position previously obtained") and needs to be

```
  completed.
-> Accepted.
-> Now it says ‘‘With a stream buffer for a repositional stream
-> (but not an arbitrary-positional stream), a C++ program can either
-> obtain the current position of the stream buffer or specify a position
-> previously obtained’’.

  27.1.2.4 [lib.iostreams.pos.t]: table 66: first row has assertion
  "p == P(i)" but p does not appear in the expression for that row;
  also, that row has the note "a destructor is assumed" -- what does
  this mean?
-> Editorial.

  27.4.2.2 [lib.ios.traits.values]:
  "int_type not_eof(char_type c);
  Returns: a value other than the end-of-file, even if c == eof().
  Notes: It is used in basic_streambuf<charT,traits>::overflow().
  Returns: int_type(c) if c!=eof()."

  Why are the two "Returns:" sections separated?  The description of
  basic_streambuf<charT,traits>::overflow() sheds no light on the use of
  this function.  Can we have a less oblique explanation?
-> Accepted.
-> The two "Returns:" sections have been merged together.
-> The function traits::not_eof() is used in two places, in overrides
-> of overflow and pbackfail.  If you call these functions with
-> parameter traits::eof(), the function returns traits::not_eof(
-> parameter ) to indicate success ( allows to differentiate from
-> failure in which case the functions return traits::eof() ).

  27.4.2.4 [lib.ios.traits.convert]:
  "state_type get_state(pos_type pos);
  Returns: 0."

  Can we get an explanation?
-> Accepted.
-> The "Returns:" sections says now:
-> Returns: A ’state_type’ value which represents the conversion state
-> in the object ’pos’.

  27.4.3.2 [lib.fmtflags.state]:
  "int width() const;
  Returns: The field width (number of characters) to generate on certain
    output conversions."

  Should read "Returns: The minimum field width ...."
-> Accepted.

  27.4.3.4 [lib.ios.base.storage]:
  "long& iword(int idx);
  Effects: If iarray is a null pointer, allocates an array of int ...."

  Why not an array of long?  Also, "Notes: After a subsequent call to
  iword(int) for the same object, the earlier return value may no longer
  be valid." This note (and the footnote accompanying it) appear to
  imply that it would be impossible to rely on the use of this function
  to store a value in the array, then come back to read it with a second
  call to the function.
-> The Effects: clause has been changed, to say: ‘‘If iarray is a null
-> pointer, allocates an array of long of unspecified size ...’’.
-> What the WP says is: ‘‘The reference returned may become invalid
-> after another call to the object’s iword member with a different index,
-> after a call to its copyfmt member, or when the objects is destroyed’’.
-> But if you call again the function with the same index, you will get a
-> new reference, which will point at the same value (except in the case
-> where the object is destroyed).
```

27.4.3.5 [lib.ios.base.cons]: In table 72, "rdstate() [returns] goodbit if sb is not a null pointer, otherwise badbit." Where is ‘sb’ explained? Also, the fonts in this table need to be used consistently.

-> Editorial.

27.5.1 [lib.streambuf.reqts]: Paragraph 3, 3rd constraint: "If xnext is not a null pointer and xbeg < xnext for an input sequence, then a putback position is available. In this case, xnext[-1] shall have a defined value and is the next (preceding) element to store a character that is put back into the input sequence." The wording of the last sentence is fuzzy.

-> Editorial.

27.5.2.3.1 [lib.streambuf.get.area]:
"char_type* egptr() const;
Returns: The end pointer for the output sequence."

Should be "... pointer for the input sequence."

-> Accepted.

27.5.2.4.1 [lib.streambuf.virt.locales]: "Between invocations of this function a class derived from streambuf can safely cache results of calls to locale functions and to members of facets so obtained." Does this mean that changes in locale can be effectively ignored by the streambuf?

-> The only way to change the locale object imbued in the streambuf is
-> by calling the member function imbue. Therefore the description of
-> the imbue function is clear, and does not imply that changes in
-> locale can be ignored by the streambuf.

27.6 [lib.iostream.format]: under "Header <iomanip> synopsis: ‘typedef ? smanip;’ -- What does this mean?

-> Has been removed.

27.6.1.2 Formatted input: What has happened to the input operators for unsigned char?

-> ????

27.6.1.1.2 [lib.istream.prefix]: in paragraph 1: "Otherwise it calls setstate(failbit) (which may throw ios_base::failure (27.4.4.3)) and returns false."

How about "... and (if an exception is not thrown) returns false."

-> Editorial.

27.6.1.2.1 [lib.istream.formatted.reqmts]: Paragraph 3 seems to imply that if extraction of a floating-point value from a stream encounters a value which has more precision than can be held in a float, and operator>>(float&) is used, the fail bit will be set. Will this not be an unexpected outcome for most programmers?

-> Accepted.
-> There are now seperate get functions (in locale num_get facet)
-> for float and double.

27.6.1.2.1 [lib.istream.formatted.reqmts]: Paragraph 5: "In case the converting result is a value of either an integral type ... or a float type ... performing to parse and convert the result depend on the imbued locale object." This is really French converted to English by translation software, right? :->}

-> Editorial.

27.6.1.2.2 [lib.istream::extractors]: Paragraph 2: "If the function stores no characters, it calls setstate(failbit), which may throw ios_base::failure (27.4.4.3). In any case, it then stores a null

character ...." How can it store anything if an exception is thrown?
C++ does not use the resumption model for exception handling.
Different language than "In any case" is needed here.
-> Editorial.
-> The intent is clear, a null character is stored wether or not the
-> function fails. Then if the function fails the call to
-> setstate(failbit) occurs after storing the null character.

27.6.1.2.2 [lib.istream::extractors]: Paragraph 2:
"basic_istream<charT,traits>& operator>>(char_type& c);
Effects: Extracts a character, if one is available, and stores it in c.
Otherwise, the function calls setstate(failbit)."

Not eofbit?
-> Otherwise, the function calls setstate(failbit|eofbit).

27.6.1.2.2 [lib.istream::extractors]: Paragraph 3:
"basic_istream<charT,traits>& operator>>(short& n);
Effects: Converts a signed short integer, if one is available, and
stores it in n."

Why does the document identify what happens when a character is not
available (see paragraph 2), but not when a number is not available?
-> Editorial.

27.6.1.4 [lib.istream.manip]: "... saves a copy of is.fmtflags ...."
Should this not read "... saves a copy of is.flags ...."?
-> Accepted.
-> saves a copy of is.flags()

27.6.2.4.2 [lib.ostream.inserters]:
"basic_ostream<charT,traits>& operator<<(unsigned long n);
Effects: Converts the unsigned long integer n with the integral
convertsion specified preceded by l."

Should this be "... preceded by ul."?
-> Accepted.

27.7 [lib.string.streams]: table 77 ("Header <cstdlib> synopsis")
appears to be out of place. Furthermore, the top row of the table:
"Type ... Name(s)" doesn't seem to match the data in the table, which
only contains names, but no types.
-> Not there anymore, has been removed.

27.8.1 [lib.fstreams], paragraph 2: "... the type name FILE is a
synonym for the type FILE." This seems like an odd sort of synonym,
doesn't it? Also, the last sentence of this subsection, "Because of
necessity of the conversion between the external source/sink streams
and wide character sequences." is incomplete.
-> Editorial.

27.8.1.3 [lib.filebuf.members]:
"bool is_open() const;
Returns: true if the associated file is available and open.

basic_filebuf<charT, traits>* open(const char * s, ios_base::openmode
mode);
Effects: If is_open() == true, returns a null pointer. Otherwise, calls
basic_streambuf<charT,traits>::basic_streambuf() (27.5.2.1). It then
opens a file, if possible, whose name is the NTBS s ("as if" by
calling ::fopen(s,modstr))."

Why does open() only open the file if is_open() is not already true?
At best, the sequencing is confused here.
-> If is_open() is true, there is already one file attached
-> to the basic_filebuf object. Therefore you need to call the member

-> function close() before trying to open another file with the same
-> basic_filebuf object.

27.8.1.4 [lib.filebuf.virtuals]: No description is given for setbuf(char_type *, int). Also, descriptions for seekpos(), sync(), and imbue() are also missing or hopelessly jumbled (e.g., the description of imbue(const locale& loc) talks only about calling sync()).

-> Accepted.
-> Descriptions for seekpos(), sync() and setbuf() will be provided.

General comment: Initialisms (POD, for example), should be expanded at the location of their first occurrence, or (better) placed in a glossary, or (best) both.

-> Editorial.

This is probably too late to make it into the standard (unless the process rolls into further extensive revisions and balloting anyway, which -- judging from the state of the Input/Output library section -- seems likely :->}), but I'll point it out it all the same. If we really want programs to use the iostreams package instead of the FILE I/O calls, the iostreams package should provide as a minimum the same facilities as the older library. Specifically, the standard C I/O package provides a convenient method for controlling the maximum number of characters to write in formatted I/O, e.g.:

```
fprintf(fp, "FONT NAME: %.16s\n", font_desc.font_name);
```

This handles the case of a structure which has enough space for a string which will not necessarily be NUL-terminated if the maximum number of characters are stored for the string (a common enough situation when one is manipulating data structures written by someone else's software).

What are the reasons for leaving this out of the iostreams package? Also (while on the topic of rounding out iostreams to match what the competition can do), how difficult would it be to provide the ability to control the (minimum) number of digits in the exponent for a formatted floating point number written using scientific notation (as, for example, one can do in Ada)?

-> Rejected, request for an extension.

23- Comment from Donald Killen / Greenleaf Software Inc.
Received by email
email address: dkillen@iadfw.net
Was comment T10 in the post-Monterey mailing document.
(also unregistered comment U1)

All compiler vendors should use the same algorithm for mangling names.

-> Rejected.
-> This constrains the implementations too much.

24- Comments from Herb Sutter / Connected Object Solutions
Received by email
email address: herbs@interlog.com
Was comment T27 in the post-Monterey mailing document.
(also unregistered comment U11)

Proposed current_class keyword

-> Rejected, request for an extension.

25- Comments from Nigel Chapman
Received by email
email address: ???
Was comment T28 in the post-Monterey mailing document.

I write to draw your attention to an inconsistency of presentation in the C++ draft standard. In section 12.4, paragraph 9, we read ‘‘Destructors are invoked implicitly (1) when an automatic variable or temporary object goes out of scope’’. However, in section 3 the authors go to some length to define a scope as a ‘‘portion of program text’’. It only makes sense to refer to where a name goes out of scope, not when an object does. This sentence should presumably be rewritten in terms of the concepts of storage duration and lifetime, defined in sections 3.7 and 3.8. Interestingly, although the formulation in terms of scope appears in the ARM, a correct version is given in ‘The C++ Programming Language, 2nd edition’’ p170.

-> Editorial.

26- Comments from David Qualls
Received by email
email address: dqualls@ocdis01.tinker.af.mil
Was comment T30 in the post-Monterey mailing document.

26.1

Subject: Preprocessor, macro expansion, escape sequences.

Question: Are (character) escape sequences given their meaning during macro expansion? I don’t feel the book is clear on this issue.

Example:

```
#define  remove_tail( statement )   statement ## \b\b\b
remove_tail( printf("stuff"); ) %d\n", int_var);
```

Does this work as expected (per the standard)? That is, does it expand (per the standard) to

```
printf("stuff%d\n", int_var);
```

Comments: The couple of compilers I’ve tested do not interpret it this way. Enabling the pre processor this way would greatly increase it’s capability. We would (and this would be nice ANYWAY) also need an escape sequence for a simple forward space. In the example above, we can’t separate the "stuff" from the %d without it. Note: I ran squarely into this question while attempting to write an ANSI C conforming preprocessor: the book was not clear.

```
-> Rejected.
-> WG21 has, in general, preferred to make (almost) no changes to the
-> preprocessor from ISO C, and has made no changes in this area.
-> The example provided is not valid in C or C++:
->
-> remove_tail( printf("stuff"); ) %d\n", int_var);
->              printf("stuff"); ## \b\b\b %d\n", int_var);
->                              -------
-> Tries to catenate  ;  with  \
-> Doesn’t form a valid pp-token.
-> See 16.3.3 [cpp.concat].
```

26.2

Subject: Preprocessor, line continuation with ’//’ comments.

Question: The book is not explicitly clear as to how the // comments, and the '\''\n' interact. Does the // comment terminate with the '\' <newline> combination or not.

Example:
```
#define  comment_question( arg )                     \
   global_var1 = arg % 7 // this won't work with   \
   global_var2 = arg / 7 // my primary compiler!

#define  same_question( arg )                        \
   global_var1 = arg % 7 /* this DOES work, but */ \
   global_var2 = arg / 7 /* is not nestable.    */

/* History: an earlier version of 'same_question'
#define  same_question( arg )                        \
   global_var1 = arg % 6 /* OOP'S. This really  */ \
   global_var2 = arg / 6 /* goes afoul! Nesting NOT ALLOWED!*/
*/
```

Comments: Based on the examples above, it's obvious to me that during preprocessing, comment termination should occur BEFORE line concatenation. Having line concatenation occur before comment termination leaves no way to embed comments within macros that might later need to be commented out. The book says that line concatenation precedes comment removal, but r.2.2 SEEMS TO SAY that // comments should terminate on the PHYSICAL line they appear on, not the extended line (some interpretive reading between the lines there). Again, I first ran into this while trying to make my own ANSI preprocessor work with the // style comments. Please make this rule explicit.

-> Rejected.
-> This was discussed at great length. The way to embed comments into
-> multi-line macros is to use the /* style of comment.

26.3

Subject: Preprocessor, possible ANSI C extension to allow empty args

Question: If C++ is to remain a superset of C, then would it not be wise to incorporate the features which the next revision of ANSI C is likely to incorporate?

Comments: One possible new addition to C will be the ability for the preprocessor to permit empty parameters within macro calls.

-> Rejected.
-> Breaks C compatibility.
-> This subject has been discussed at great length in WG14 regarding
-> the 5-year revision of C, and no decision has been made by WG14.
-> Currently, an implementation is at liberty to support the feature,
-> but not required to do so.

26.4

Subject: C(++) as a "portable assembler"

Note: This one is my 40 pound soap box!

Commentary: I laugh every time I read where someone refers to C (or C++) as a portable assembler. It's NOT! It's definitely not an assembler, and it's not terribly portable. It is not an assembler because the language lacks a direct way to do indexed local jumps. I'm only familiar a couple of assembly languages, but I sure thought that indexed local jumps were a part of every assembler. That is, the ability (within a procedure) to jump to a code location specified within another register or memory location.

```
jmp[cd_ptr]    ;execution jumps to where cd_ptr is pointing.
```

C(++) is not very portable either because the standard headers contain no standard macros addressing how integers and structures are stored and accessed on varying platforms.

The issue of indexed local jumps could be easily fixed in C(++) by allowing pointers to labels.

#012#

Example:

```
void  example( int arg )
{
  label *lPtr[3] = { LABEL1, LABEL2, LABEL3 };
  /* 'label' is a new keyword. In the classic C sense, the  */
  /* label name is really a pointer to a code location.     */
  /* C(++) already permits forward referencing in this      */
  /* sense.  That is, you can 'goto' a label that hasn't    */
  /* been previously declared.  Some environments insist on */
  /* defaulting to 'const' to prohibit self modifying code. */

  arg = func( arg );
   /* arg gets distorted in a way that's   */
   /* too complex for the compiler to be   */
   /* able to predict all possible values. */

  goto lPtr[ arg ]; /* The code author understands */
                    /* the possible values.        */

  LABEL1: /* do some stuff */
  LABEL2: /* do some stuff */
  LABEL3: /* do some stuff */

  return;
}
```

Comments: I admit that in the example above, a switch/case statement would do the trick. The problem with switch is that some compilers simply convert switch statements into a long line of very slow running if statements. In some cases, as I've tried to allude to above, the compiler simply can't understand what possible values the arg may take on, and thus is forced into translating the code into if statements. It'd be incorrect translation to do otherwise!

The real utility of this proposed construct is when the code writer KNOWS the possible values the index can assume and the compiler simply can't figure them out. I have been very frustrated (and I suspect, so have a lot of other performance hounds who default to writing in assembler) by the lack of indexed local jumps in the C(++) language.

Now regarding portability. In order to take advantage of the low level tools which C(++) provides for us, we need a whole suite of portability macros for the integers. I'm not sure we can do much with the floating points since they are allowed to change representation while running.

#012#

Example:

```
#define  CHAR0INSHRT   1  /* least significant char in a     */
                          /* short when the short is treated */
                          /* as an array of chars.           */
#define  CHAR1INSHRT   0  /* next most significant */
```

```
#define  CHAR0INLONG   7  /* least significant char in long */
#define  CHAR1INLONG   3  /* next most significant */
#define  CHAR2INLONG   5  /* even more significant */
#define  CHAR3INLONG   1  /* continuing in significance */
#define  CHAR4INLONG   6  /* ditto */
#define  CHAR5INLONG   2  /* ditto */
#define  CHAR6INLONG   4  /* ditto */
#define  CHAR7INLONG   0  /* most significant char in long */

/* macro to access the N'th least significant char in a long */
#define  NthCHARINLONG( N, longarg )    \
             *((char*)(&longarg) + CHAR ## N ## INLONG)
```

Make similar macros for all the other integer types.

If it's decided that significance should be indicated in a different order, just reverse the order.

If an environment won't support such disection of the larger types, then just don't define them.

We also need similarly clever macros which indicate how the various types align within structures, which bit (least or most significant) is the sign bit, is zero represented by all bits set to zero or something else, how bitfields are ordered, as well as any other environment specific issues, including everything which the standard defines as "implementation dependent". A full suite of these macros will make portable programming a MUCH easier job.

-> Rejected.
-> Request for an extension.

27- Comments from Ajay Kamdar / Lehman Brothers
Received by email
email address: ajay@lehman.com
Was comment T31 in the post-Monterey mailing document.

Make the destructor of a class implicitly virtual if the class has any other virtual functions.

Discussion
----

*) Forgetting to make the destructor of a polymorphic class virtual is a common mistake made both by inexperienced and experienced C++ programmers. This makes it harder to use the language, and the resulting problems are often difficult to debug and fix. Accepting this proposal eliminate an unnecessary source of errors.

*) There are no backward compatibility issues to worry about. The behavior of deleting an object using a pointer to a static type without a virtual destructor is currently specified to be undefined if the dynamic type of the object is different from the static type.

*) There is no reason for wanting *not* to execute all the appropriate destructors.

*) There would be no change to the layout of an object because the destructor would be made implicitly virtual only if the class had at least one other virtual function.

*) A (positive) side effect of the change would be that existing

erroneous code which currently has undefined behavior would start behaving properly.

*) It would be very easy to modify compilers to implement the new behavior.

-> Rejected.

28- Comments from Darin Adler / General Magic
Received by email
email address: darin_adler@genmagic.com

1. I was able to make one program much faster by specializing iter_swap to use the swap member function of vector. Is there some way to do this generally and automatically, instead of doing it explicitly for each specific type of collection? I did something like this:

```
inline void iter_swap(collection *a, collection *b) {
    a->swap(*b);
}
```

With this speedup, sorting a vector of vectors is a lot faster and does a lot less memory allocation.

-> Already present in the standard.
-> Partial specializations of the non-member template swap() are
-> provided for all containers that define a swap() member (all
-> containers defined in Clause 23 except bitset).

2. I suggest you make the random-number generator used by random_shuffle available in <functional>. I would find it useful. Also, to make them both generally useful in production programs, there should be a way to reseed the random-number generator.

-> Rejected.
-> The committee previously rejected at least one random-number
-> generator proposal and does not wish to reopen the topic at this
-> time.

3. I think operator!= in <utility> should be parameterized on the types of both its arguments. This doesn’t hurt single-type use of the template function, and makes it more generally useful. Perhaps the same goes for operator>, operator<=, and operator>=, but for those I am not as sure.

```
template <class T1, class T2>
inline bool operator!=(const T1& x, const T2& y) {
    return !(x == y);
}
```

If the operator!= template is not parameterized on the types of both arguments, it prevents me from making my own operator!= template that is parameterized on both because it would conflict with the singly parameterized one in the standard headers. In the current situation, I have to write individual operator!= functions for various combinations.

-> Rejected.

4. There should be an output iterator template class that binds a unary function with another output iterator. This would go into the standard header <functional>. Here is a possible definition that I’ve used in my programs:

```
template <class OutputIterator, class Transform>
class transform_output_iterator : public
output_iterator {
    typedef transform_output_iterator
        <OutputIterator, Transform> self;
protected:
    OutputIterator out;
    Transform filter;
public:
    transform_output_iterator(OutputIterator i,
          Transform f)
        : out(i), filter(f) {}
    self& operator=(const Transform::argument_type&
value) {
        *out++ = filter(value);
        return *this;
    }
    self& operator*() { return *this; }
    self& operator++() { return *this; }
    self& operator++(int) { return *this; }
};
```

Here is a rough attempt at doing the same thing for binding an input iterator with a function. I haven’t used this one:

```
template <class InputIterator, class Transform,
     class Distance>
class transform_input_iterator
      : public input_iterator
          <Transform::argument_type, Distance> {
    typedef transform_input_iterator
          <InputIterator, Transform, Distance> self;
    friend bool operator==(const self&, const self&);
protected:
    InputIterator in;
    Transform filter;
public:
    transform_input_iterator(InputIterator I,
          Transform f)
        : in(i), filter(f) {}
    Transform::result_type operator*() const
          { return filter(*in); }
    self& operator++() { ++in; return *this; }
    self operator++(int) {
        self previous = *this;
        ++in;
        return previous;
    }
};

template <class InputIterator, class Transform, class
Distance>
inline bool operator==(
      const transform_input_iterator
          <InputIterator, Transform, Distance>& x,
      const transform_input_iterator
          <InputIterator, Transform, Distance>& y) {
    return x.out == y.out;
}
```

These two are powerful, because there are already many ways to combine functions, which now can be easily attached to iterators.

-> Rejected.
-> There are lots of interesting possible extensions to the STL. The
-> committee doesn’t want to consider them at this time.

5. The use of *x++ in the standard library makes inefficient some algorithms with some kinds of iterators. To implement the post-increment operator, some iterators have to keep a local copy

of the old value around. Often this leads to extra complexity and work. The algorithms derive little benefit from the freedom to use the post-increment form. I was able to make a measurable improvement in the speed of my program by altering some functions in a copy of <algo.h> to use a separate pre-increment instead of using the *x++ dereference/post-increment combination.

-> Rejected.
-> After substantial discussion it was decided that the *x++ sequence is
-> used too frequently to be eliminated.

6. I have used the streams library for a number of applications. In the programs I have been writing, the design of the functions named getline has caused some trouble. When getline is called and the stream has a line that contains no text, ios::failbit is set on the input stream. While consistent with the behavior of the similar function named get, the behavior is quite inconvenient. To illustrate, here is a fairly simple function that copies a file, reading and writing a line at a time, adding a trailing new-line if it is missing:

```
istream& copy_lines(istream& from, ostream& to) {
    string line;
    while (true) {
        getline(from, line);
        to << line << "\n";
        if (from.eof() || from.bad())
            break;
        if (from.fail())
            from.clear(from.rdstate() &
~ios::failbit);
    }
    return from;
}
```

The complexity of the function is due to the unwanted fail state caused by empty lines. A simpler function is possible if getline does not set the fail bit.

```
istream& copy_lines(istream& from, ostream& to) {
    string line;
    while (proposed_getline(from, line))
        to << line << "\n";
    return from;
}
```

This simpler behavior seems like a better definition for the getline function. It’s easy to define today’s getline in terms of the proposed one and the other way around, it’s just that the proposed getline is more useful in many contexts. Empty lines should not require special code to handle them-it’s almost like having a numeric formatting routine that fails when asked to format the number 0.

It’s possible to express today’s getline in terms of my proposed getline:

```
istream& proposed_getline(istream& stream,
                          string& line) {
    todays_getline(stream, line);
    if (stream.eof() || stream.bad())
        return stream;
    if (stream.fail())
        stream.clear(stream.rdstate() & ios::failbit);
    return stream;
}
```

```
istream& todays_getline(istream& stream, string& line)
{
    proposed_getline(stream, line);
    if (stream.good() && line.empty())
        stream.setstate(ios::failbit);
    return stream;
}
```

One drawback I can see is that this makes the getline function different from the get function in an additional way. In practice, I don’t think that this outweighs the greater convenience of the function for clients.

I’ve focused on the version of getline for the string class, but I believe similar arguments apply for getline into null-terminated buffers. The fail state would always mean that there are too many characters for the buffer, rather than sometimes indicating an empty line.

-> Accepted.
-> getline used with the default delimiter does not set failbit
-> when reading an empty line.

7. bitset::set should take a bool as its second parameter instead of an int. With the current definition, if sizeof(int) is not the same as sizeof(long), passing a non-zero long that becomes zero when cast to int will set the bit to 0. If the function is defined to take a bool instead, the bit would be set to 1, which makes sense since the value of the long is non-zero.

-> Accepted.

29- Comments from Jack Reeves / Dow Jones Telerate Systems Inc.
Received by email
email address: jack@fx.com

1. Suggestion - the container classes which provide a function "reserve()" (currently ’basic_string’ and ’vector’) also need to provide the dual function - something like "release_excess()" or "shrink_to_fit()".

Discussion -

The project I am currently working on uses STL quite extensively. We are using both the ObjectSpace STL<Toolkit> and the version of STL that works with the G++ compiler. In one part of our library we have a set of classes that represent various data types that are used throughout the rest of the system. Some of these classes are themselves containers. Naturally, we have implemented them using the appropriate STL classes. One of the most used of these is a Table. This consists logically of rows and columns. It is implemented as a vector of vectors.

We have one table that represents a screen of data -- a 80 x 20 character matrix. In the implementation, this became a vector of 20 elements, each a pointer to a vector containing a single pointer to a string. In all, we expected this data element to use less than 2Kbytes of memory. What we found was that it used over 80Kbytes. This was considered excessive overhead, even on a Unix system. Upon investigation, we discovered that the following example allocates 4K of memory for the vector.

```
    vector row;                 // allocates no storage for row
    row.push_back(element_ptr); // allocates 4K
```

We worked around this problem by changing to the following

```
    vector row;      // allocates no storage
    row.reserve(1);  // allocates (but does not initialize) 1
element
    row.push_back(element_ptr);
```

It is still the case however, that whenever a vector has to be reallocated, it doubles the memory used. This led us to implement a "shrink_to_fit()" function for those types based upon vector. This works, but it forced us to switch from having a vector as a data member of the class, to having a vector pointer, since shrink_to_fit() has to create a copy of the vector.

Ideally, "shrink_to_fit()" should be a member of the container class itself. With appropriate support from the underlying memory model (reallocate in place), the operation could be constant time. Using typically available facilities, "shrink_to_fit()" will still have to reallocate the vector and copy it, but it could do this using low level memcpy function instead of the high level copy constructors invoked by having the using program reallocate the vector. It is my understanding that the resize() function does not provided the needed capability, since its task is to change the "size" of the container, not its "capacity."

-> Rejected.
-> This type of function (and several variants) were discussed in
-> detail by the committee. It was decided not to embed too much
-> "optimization advice" about memory management in the interface
-> of vector (and basic_string). Exactly how much, if any, "excess"
-> capacity is allocated by reserve() is not specified by the
-> standard. Allocation of excess memory is purely a property
-> of the particular implementation.

2. Suggestion - the container classes need specializations defined for pointer elements. For example -

```
  template<class T> class ptrvector
  template<class T> class ptrlist
  template<class T> class ptrdeque
  template<class T> class ptrset
  template<class T> class ptrmap
  template<class T> class ptrmultiset
  template<class T> class ptrmulitmap
```

Instantiated versions would have a implementations based upon the base class instantiated for void*. The template would have to be instantiated with a pointer type (or something which could be converted to/from void*).

Discussion -
In our current project, we are using STL extensively. Of some 22 containers used in the base library alone, 12 are some variant of containers for pointers. This is hardly surprising -- in any large system, the need to deal with objects polymorphically will mean that most objects are in fact either references or pointers. While I am aware that the STL goes to lengths to be efficient, there still tends to be a lotof code replication when there are a dozen different containers instantiated, all of which could be implemented with the same code. While the container classes in the current version of the draft standard library are certainly much richer than just the dynarray and ptrdynarry that were first proposed, I regret the absence of the ptr-xxxx versions. I have found through experience that creating a ptrxxxx template class derived from xxxx<void*> is non-trivial since it involves also creating all of the appropriate iterator classes. I know that the standard library is just a base library, and in general the user is left with deriving the appropriate classes for his project. Nevertheless, this is exactly the kind of thing that should only have to be done once (preferably by somebody else). This was obviously realized when dynarray was proposed as part of the standard library. I am not aware of any counter-arguments for including it with the STL. It would certainly be useful for my projects.

-> Rejected, request for an extension.
-> Defining separate containers for pointer types (and later, defining

-> partial specializations of the containers for pointer types) was
-> previously considered and rejected by the committee.

3. Request for clarification - in the basic_string class, the description of the "size()" function states that it uses traits::length. traits::length() in turn is described as being similar to ::strlen, i.e. it looks for the terminating EOS character. Since my assumption is that a "string" is just a specialized container for a sequence of characters, irrespective of what the values of those characters are, then size() should not depend up traits::length, but solely upon how many characters have been put into the string. Stated another way, I would expect the following code fragment to work:

```
    vector<char> vec(10, '\0');  // make sure it is full of nulls
    string str(vec.begin(), vec.size());  // copy the vector into
the string
    assert(str.size() == vec.size());  // they should have the
same # characters
```

If str.size() uses traits::length() then the assert will fail, and it will not be at all clear what I would get with a call to str.data();

-> Since this comment, the note in size() which mentions traits::length()
-> has been removed.

30- Comments from Jack Reeves / Dow Jones Telerate Systems Inc.
Received by email
email address: jack@fx.com
Was comment T33 in the post-Monterey mailing document.

1. The function basic_string<>::c_str() is prototyped as

```
    const charT* c_str() const
```

The function returns a pointer to an eos() terminated string. The semantics are fine, I just think the prototype is in error. It think the correct prototype should be

```
    const charT* c_str()    // not 'const' function
```

I will accept that adding a traits::eos() character in the undefined portion of the reserved memory outside of the valid string data is philosophically not a change of the state of the object and hence can be allowed within a 'const' member function. However, adding this 'hidden' character can cause the re-allocation of the internal representation, and I draw the line at this silliness:

```
    void f(const string s)
    {
        size_t before = s.capacity();  // const function
        cout << s.c_str() << endl;     // const function???
        size_t after = s.capacity();   // const function
        assert(before == after);       // This should never fail!!!
    }
```

In general, I consider it unacceptable for a 'const' function to cause changes in the underlying state of the system irrespective of whether that function changes the "contents" of the object as seen through the interface of the abstraction. As such, I will accept c_str() as a const member function only if it is defined to never re-alloc the internal string. This could be done of course, by insisting that the memory reserved always contains room for the eos(), but I think a better approach is to simply change the definition of c_str. I note that my definition of what is "const" may be different from the definition of 'const' as used in the language standard. If so, please point me to where the definition is spelled out in the standard.

-> Rejected. The externally observable semantics of the const member
-> c_str() are exactly those desired. Requiring an implementation to
-> behave in the ways described would be over-constraining. Nowhere
-> in the definition of capacity() does the Draft state that an
-> implementation is required to return the same value on consecutive
-> calls.

2. The function basic_string<>::data() is prototyped as

   ```
        const charT* data() const
   ```

   and defined to return a null pointer if size() == 0 otherwise c_str(). I believe this is a mis-wording. data() should return the appropriate pointer (or null) but should not be required to return an eos() terminated string. There are two reasons for this. (a) If data() does not return c_str() it can truly be a 'const' member function, and this is good (see 1. above). (b) Perhaps more importantly, there is no need for data() to terminate the string. In using several different versions of string class, most of which come close to the standard, we have never found it necessary to have a function that has the semantics as data() is now defined to have. We have found many uses for a function ('const' function) that gives access to the internal data pointer. In fact, we use strings in numerous situations where '\0' is a valid data element and so terminating such strings is a waste of time since they are always dealt with in conjunction with their length().

-> Accepted.

3. The latest version of the standard adds some new member functions to class basic_string. There is now a size() function and several other changes that bring strings more in parallel with the newly defined containers. I have previously pointed out that size() is defined in terms of traits::length() which is in turned defined semantically to be the same as ::strlen(). I feel sure this is an error. I note that function length() is defined to be the same as size(). I presume that length() is retained for compatibility with previous versions of string (and may be deprecated in the future). I wonder if maybe what was really desired was that basic_string::length() should return traits::length() if this is less than size(), size() otherwise. I really doubt it, but thought I would ask.

-> The definition of size() which mentioned traits::length() has been
-> changed. The member length() returns the same value as the member
-> size(). The member length() was retained for compatibility with
-> existing practice. It is not currently being considered for
-> deprecation.

4. I note that the latest version of the standard changed the order of the parameters for one of the constructors from

   ```
        basic_string(charT c, size_type n = 1, Allocator& = Allocator())
   ```

   to

   ```
        basic_string(size_type n, charT c, Allocator& = Allocator())
   ```

   I presume the latter is correct, but wanted to verify. We have hit at one occasion where an older program had

   ```
        string s('@', 1);
   ```

   and this continued to compile correctly with the new header file (we are using G++), but silently changed its meaning.

-> The latter is correct.

5. I have already suggested the following, but will suggest it again, as I consider it important. Class basic_string has a reserve() function, but no release() function. It really needs a release() (or shrink_to_fit()) function. Partly this is just good design (pardon my arrogance) -- the reserve() function is used to indicated an anticipated increase in the size of the string, and the release() function is its opposite and is used to indicate that no more changes are anticipated and the excess reserved memory can be given back to the system. Partly, reserve() and release() can be used with a special allocator that deals with relocatable memory such as the original Macintosh or Windows -- reserve() would do a lock and release() could unlock (as well as shrink). I note two aspects about release(). The first is that it could interact somewhat poorly with c_str().

   ```
        void f(string s)
   ```

```
        {
            s.release();                 // shrink to fit
            cout << s.c_str() << endl;   // trying to re-alloc the string
                                         // to size()+1 might cause it
                                         // to have quite a bit of slop
        }
```

I would consider this annoying, but something that could be lived with. However, an alternative provides a solution to my desire for a release() function and this problem -- redefine the semantics of reserve() to allow it to function as a release() function also. Thusly -

```
        after reserve(size_type n) ::=
           if (n < size()) then capacity is set to size()
           otherwise capacity() will equal n.
```

Frankly, this would be my preference. Thus the example above would become

```
        void f(string s)
        {
            s.reserve(s.size()+1);
            cout << s.c_str() << endl;
        }
```

with the assurance that the actual memory used is the minimum necessary. The reserve() function could be prototyped as

```
        void reserve(size_type res_arg = 0)
```

where the default argument would allow the use of
     s.reserve() to be semantically equivalent to shrink-to-fit.

-> Accepted.

6. All of the above discussion about release() applies equally to the vector<> class. In fact, I like the new reserve() idea so much I think I'll go implement it in our string and STL libraries and let you know how it comes out. Let me know what you think.

-> Rejected.

31- Comments from Scott Schurr / Integrated Measurement Systems, Inc.
Received by email
email address: scotts@ims.com
Was comment T32 in the post-Monterey mailing document.

Exception specifications should be check for correctness at compile time. The current exception definition prevents compiler writers from checking for properly constructed exception hierarchies.

-> Rejected.
-> See "The Design and Evolution" of C++ by Bjarne Stroustrup for
-> explanations of why static checking was rejected for exception
-> specifications.

****

Unofficial Comments

****

U2- Comment from Jerry Anderson

I would recommend the addition of a keyword that served the following purpose:

When a function has been over-ridden in a sub-class and it is necessary to call the base class implementation of the function also. A keyword denoting the base class would be useful instead of the explicit reference that is now required.

```
      void MySubClass::SomeFunction(void)
      {
        ...
        MyBaseClass::SomeFunction();
```

```
    ...
  }
```

Replace with:

```
  void MySubClass::SomeFunction(void)
  {
    ...
    base->SomeFunction();
    ...
  }
```

-> Rejected.
-> Already considered and rejected - inherited keyword.
-> When the keyword was rejected, the committee felt that a uniform
-> coding style such as
->   typedef base inherited;
-> could be used adequately.

U3- Comment from Steve Meirowsky / IFR Systems Inc.
Received by email
email address: steve.meirowsky@nwis.com
Was comment T14 in the post-Monterey mailing document.

I think C++ should have one or two new numeric types that are integral as part of the language. A 64bit and 128bit longs. I think the 64bit longs should be mandatory! Also please choose some easy to remember name like dlong/qlong or long64/long128.

-> Rejected.
-> This was perceived as too much of an extension to be comsidered at
-> this late stage in the C++ standardization process.

U4- Comment from Steve Meirowsky / IFR Systems Inc.
Received by email
email address: steve.meirowsky@nwis.com
Was comment T14 in the post-Monterey mailing document.

[First wish is the same as U3].

The second wish list item is ranges on ’case’ statements similar to Pascal. For example, ’case 9..49:’. We really don’t care about the method...just that we have it in the language.

-> Rejected.
-> Request for an extension.

U6- Comment from Boris Rasin
Received by email
email address: brasin@netvision.net.il
Was comment T18 in the post-Monterey mailing document.

Subject: Template argument deduction [temp.deduct].
Proposed addition: Class template argument deduction.

In a call to class template constructor, class template arguments can be deduced from constructor arguments, under the rules for function template argument deduction.

Example:

```
       class Mutex { ... };
       class Semaphore { ... };
       template <class T> class Lock { ... };
```

```
        Mutex M;
        Semaphore S;
        Lock L1 (M); // Lock<Mutex> L1 (M);
        Lock L2 (S); // Lock<Semaphore> L2 (S);
```

-> Rejected.
-> Request for an extension.

U7- Comment from Greg Weidman / Kaman Sciences Corp.
Received by email
email address: weidman-alx1@kaman.com
Was comment T23 in the post-Monterey mailing document.

Point 1:

Imagine:

```
class cA { ... };
class cB : public cA {...};

main()
  {
  cB *pB = new cB [10];
  cA *pA = (cA *)pB;

  }
```

If cA and cB are different sizes, then there is no convenient way of accessing other than the first element of pA. It would have been nice to say pA[1] and gotten the same address as pB[1], but this is not the case. I understand that this would imply checking the virtual table for each of the elements in the pA array to determine their sizes, kind of destroying the default definition of [], but if I define cA::operator[](int);, then I need to access this using either

```
(*pA)[1];
or
pA[0][1];
```

neither of which is particularly convenient.

-> Rejected.
-> Request for an extension.

Point 2:

The language is really unbelievably cluttered. As long as C++ remains a superset of C, this will be true. It seems, however, that once one has defined the concept of "Reference", then one can pretty much do away with the concept of "Pointer," since both are stored as pointers within the object code. This would take C++ well away from C, but it would make the language significantly less cluttered.

-> Rejected.
-> breaks C compatibility.

Point 3:

I have run across some compilers that give "Anachronism" warnings when encountering

```
delete [n] pX;
```

These compilers seem to feel that

```
delete pX;
```

should be sufficient, although they all fail to call cX::~cX() more than once if this so-called "Anachronism" is eliminated. A clear standard on whether the delete [n] structure is needed, and whether multiple destructors should be called would be quite helpful.

-> Rejected.
-> The correct syntax to delete arrays is:
->   delete[] pX;
-> which will cause the appropriate number of destructors to be called.

U8, U9 & U10 - Comment from J. Barreiro, R. Fraley, and D. Musser
Received by email
email address: musser@cs.rpi.edu
Was comment T24 in the post-Monterey mailing document.

"Hash Tables for the Standard Template Library"

-> Rejected
-> Request for an extension.

U12- Comments from Jon Hoyle / Eastman Kodak Company
Received by email
email address: JonHoyle@aol.com
Was comment T24 in the post-Monterey mailing document.

1. For templates, allow switching on the type. For example:

```
template <class T>
void SomeFunction(T theObject)
{
        switch (T)
        {
                case int:
                case short:
                        DoSomething();
                        break;

                case double:
                case anotherType:
                        DoSomethingElse();
                        break;

                default:
                       DoEverythingElse();
                       break;
        }
}
```

This allows for fine tuning in templated functions.
-> Rejected.
-> Request for an extension

2. Define an operator @ as an additional binary operator that can be used for operator-overloading. Currently, there is no way to overload an operator for elementary types. Now this could allow us to define, say, exponentiation by:

```
int  operator@(int x, int y)
{
       if (y == 0) return 1;
       if (y > 0) return x * operator@(x, y-1);
       return (1/x)*operator@(x, y+1);
}
```

-> Rejected.
-> Request for an extension.

I would also like to commend you on your decision to add a boolean type to the standard. Currently, we always run into the problem of defining TRUE as 1, and comparing something that is true but not 1. For example,

```
      if (x & 0x007F == TRUE)
      {
             // this always fails
      }
```

I also like the idea of defaulting templated types:

```
      template <class T = int>
      class MyClass<T>
      {
           ...  /* etc. */
      }

      MyClass<> theClass;  // Default type is int
```