ISO/IEC JTC1/SC22
Programming languages, their environments and system software interfaces
Secretariat: U.S.A (ANSI)

ISO/IEC JTC1/SC22

N 1810

March 1995

TITLE:	Review by SC22/WG14 of Programming Language C++Draft Presented for CD Registration
SOURCE:	Secretariat, ISO/IEC JTC 1/SC22
WORK ITEM:	JTC 1.22.32
STATUS:	N/A
CROSS REFERENCE:	SC22 N 1709
DOCUMENT TYPE:	Comments of CD Registration
ACTION:	To SC22 Member Bodies for information. To SC22/WG21 for appropriate action.

Address reply to: ISO/IEC JTC 1/SC22 Secretariat
William C. Rinehuls
8457 Rushing Creek Court
Springfield, VA 22153
Tel: (703) 912-9680 Fax: (703) 912-2973 E-mail: rinehuls@access.digex.net

X3J16/95-0059
WG21/N0659

X3J11/95-003
WG14/N402

Date: 1 February 1995

From: ISO/IEC JTC1/SC22/WG14 (programming language C)

To: ISO/IEC JTC1/SC22/WG21 (programming language C++)

Subject: Review of C++ draft presented for CD registration

INTRODUCTION

WG14 feels an obligation to provide useful feedback to WG21 on the draft C++ Standard submitted for balloting for registration as a Committee Draft (SC22/N1709). It was our hope and expectation that we could supply at this stage a cogent list of issues whose resolution would ensure maximum compatibility between our two closely related languages. Given the current state of the C++ draft, however, that is not currently possible:

* Substantial features have been added to the draft with no accompanying semantic description. The discussion of locales (clause 22), for example, is of particular interest to the C community and is particularly lacking in explanatory detail.

* The number of typographical errors and lurches in style reveal that the document is nowhere near ready for the precise review required to determine whether compatibility between C and C++ has been adequately safeguarded.

* A number of substantive changes were made to the working draft during the balloting process. Sections were eliminated (such as the library classes bits, dyn_array, and stdiostream), others were significantly revised (complex and locales), and others were added (numeric limits, auto_ptr, and counted_ptr). Two keywords were added to the language. We thus cannot be assured that all major elements of the C++ Standard are yet in place.

Any attempt at a comprehensive review by WG14 at this stage is doubtless premature. Many of our comments would likely prove to be erroneous. Consequently, we supply here a simple compendium of comments made by various members of WG14. As a courtesy, Tom Plum has already forwarded many of them to WG21 on an informal basis, in his capacity as bidirectional liaison, to speed the editing process.

WG14 stands ready to supply additional guidance and review, to ensure that C and C++ remain ``as close as possible, but no closer." For now, however, we regret that we cannot supply a more focused critique to assist WG21.

COMMENTARY

GENERAL:

- "ANSI X3/TR-1-82:1982, American National Dictionary for Information Processing Systems" is not the correct/updated version, "ANSI Standard X3.172-1990, Dictionary for Information Systems" is the correct version.

- "may" is used in 8.3.4 Arrays paragraph 2, but "may" was changed to "can" in 6.1 Labeled statements paragraph 2, and other places in 6, but not 8. Why? Both "may" and "can" are usually used in the case of options to a standard, and can be considered synonyms. Does the C++ standard really want optional behavior? I believe the types of behavior that are consistent with a Language standard are defined behavior, undefined behavior, implementation defined behavior and unspecified behavior.

- Why do Clause 3 and 5 start with a synopsis of the clause? None of the other Clauses have this, for consistency these should be removed, or the other Clauses should have a short synopsis added.

- What is "_class.ambig_" ? It is referred to throughout the draft.

- The draft contains several clauses which deal with "scope":

1.1 Scope	[intro.scope]
3.3 Declarative regions and scopes	[basic.scope]
3.3.1 Local scope	[basic.scope.local]
3.3.2 Function prototype scope	[basic.scope.proto]
3.3.3 Function scope	
3.3.4 File scope	[basic.file.scope]
3.3.5 Namespace scope	[basic.scope.namespace]
3.3.6 Class scope	[basic.scope.class]
9.3 Scope rules for classes	[class.scope0]
10.5 Summary of scope rules	[class.scope]
14.2.2 Names from the template's enclosing scope	[temp.encl]
16.3.5 Scope of macro definitions	[cpp.scope]

It is not clear how many different "scope" concepts are dealt with, and which is referred to when the word "scope", or the expression "scope rules" is used by itself. This needs to be clarified.

- The use of "non" is inconsistent throughout the draft. Mostly two forms are used, "non-x" and "nonx", where x is a noun or adjective. Occasionally, there is also a "non- x" form, presumably a typo. If there is a rule for which form is used when, it is not clear. For consistency and clarity, only one of the two predominant forms (whichever one is correct).

In the 94-0098.j16 draft, I found:

"non-X" Form:	"nonX" Form:	"non- x" Form:
non-C++		
non-NULL		
	nonabstract	
non-array	nonarray	
non-class		
non-const	nonconst	non- const
	nonconst-class	
	nonconstant	
	nondeterministic	
	nondigit	
	nonempty	
non-function		
	nonfunctionstyle	
	nongraphic	
non-local	nonlocal	

non-member	nonmember	
	nonmodifiable	
non-mutable		
non-negative	nonnegative	
non-nested	nonnested	
non-normative		
non-null	nonnull	
non-overlapping	nonoverlapping	
non-pointer	nonpointer	
non-prototype		
	nonqualified	
non-reference		
	nonreplaced	
non-required		
non-static	nonstatic	non- static
non-template		
	nonterminal	
non-terminated		
non-trivial	nontrivial	
non-type	nontype	
non-virtual	nonvirtual	
non-volatile	non- volatile	
non-white-space		
non-zero	nonzero	nonzerodigit

- NOTE: "sub" has the same basic problem as "non". See "non" above.

- In most section headers, only the first word is capitalized. There are two exceptions which are:

  10.2 Member Name Lookup
  17.1.3 Processor Compliance

- The current draft uses "shall be", "may be", "can be", "must be" and "should be". It seem to me that "must be" is a synonym for "shall be". I suggest replacing all occurrences of "must" with "shall". For consistency use "may be" or "can be", but not both. Again this is if the standard is going to have optional behavior.

- Also "not" is associated with the same unruly group, such as "must not", "shall not", "need not" and "may not". Again I think "must" needs to be replaced with "shall". I do not know of a standard definition for "need", so "need" should be replaced.

- Where does it say that footnotes and examples are not part of the standard ?

  The ISO C Standard has the statement. "The abstract, the foreword, the examples, the footnotes, the references, and the appendixes are not part of the standard."

  Does the standard need such a statement?

- Shouldn't the "usual ... rules" be defined, this includes the expressions "usual access rules", "usual mathematical rules", "usual scope rules", etc... These can be found by looking for the string "usual" in the draft.

- Default constructor are said to be "synthesized" and "generated" as well as "declared" and "defined". For clarity and consistency, stick to "declared" and "defined" which are well-defined terms and avoid using other terms.

Symbolic header name issues:

NOTE: This is just a few of the symbolic header problems. A complete review of the symbolic header names needs to be done, or the symbolic header neames need to be completely removed from all text.

- The expression "usual arithmetic conversion" was explicitly defined in clause 4.5 of 94-0098, but this was rewritten in 94-0158, but references to "usual arithmetic conversion" are still in other parts of the draft, see 5.6 2(b) of 94-0158. Also the "_conv.arith_" has been deleted, but still referred to in the text of 94-0158 see 5.7 1(b) and 5.6 2(b).

- The symbolic header name "conv.ref" was removed, but still is referenced in 94-0158 see 10 1(k) and 14.9.3 2(f). Both of these references need to be changed, and the wording around both references reference incorrect sub-clauses.

3.6.3 Dynamic storage duration:

2b. "The library provides default definitions for them (_lib.header.new_). "

"_lib.header.new_" is not a valid symbolic header name.

3.6.3.1 Allocation functions

3a. "If an allocation function is unable to obtain an appropriate block of storage, it may invoke the currently installed new_handler and/or throw an exception (15) of class alloc (_lib.alloc_) or a class derived from alloc."

"_lib.alloc_" is not a valid symbolic header name.

5.3.4 New

10b. "A C++ program may provide alternative definitions of these functions (_lib.alternate.definitions.for.functions_), and/or class-specific versions (12.5)."

"_lib.alternate.definitions.for.functions_" is not a valid symbolic header name.

17a. "The allocation function may indicate failure by throwing an alloc exception (15, _lib.alloc_)."

"_lib.alloc_" is not a valid symbolic header name.

9.4.2 Inline member functions:

1d. "Thus the b used in x::f() is X::b and not the global b. See also _class.local.type_."

"_class.local.type_" is not a valid symbolic header name.

ISSUES BY CLAUSE:

6.1 Labeled statement:

2a. "Case labels and default labels can occur only in switch statements."

I think this should be a "shall", the wording could be the following.

"Case labels and default labels shall appear only in a switch statement."

6.2 Expression statement:

1a. "Usually expression statements are assignments or function calls."

This seems to fall into the category of usage, not language definition. I really think the standard should stay away from usage.

6.4 Selection statements:

4a. "A variable, constant, etc. in the outermost block of a statement controlled by a condition may not have the same name as a variable, constant, etc. declared in the condition."

I think "etc" should be blown up into a complete list of items. "etc" is not a well defined standard term.

NOTE: "etc" can also be found in 14.6 4(b)

6.4.2 The switch statement:

2b. "Any statement within the statement may be labeled with one or more case labels as follows:"

This in not clear to me, should it be "Any statement within the switch statement"?

7a. "Usually, the statement that is the subject of a switch is compound."

I think this sentence is about usage, and can be removed.

12.1 Constructors:

1a. "A member function with the same name as its class is called a constructor; it is used to construct values of its class type."

The expression "to construct values" needs to be defined. A suggestion is to replace it with "to allocate and initialize objects".

3b. "Default constructors and copy constructors, however, are generated (by the compiler) where needed (12.8).

Can this sentence be removed ? It comes before default and copy constructors are defined, and is redundant with text in paragraphs 4 and 5 of 12.1.

12.2 Temporary objects:

2b. "Ordinarily, temporary objects are destroyed as the last step in evaluating the (unique) expression that (lexically) contains the point where they were created and is not a subexpression of another expression."

"Ordinarily" is not clear. A suggestion is to replace it with "Except when a temporary object is used in a declarator initializer".

12.3 Conversions:

3b. "Conversions obey the access control rules (11)."

What is "the access control rules" ? Is this the same as "the usual access rules" ?

12.4 Destructors:

9a. "Invocation of destructors is subject to the usual rules for member functions, e.g., an object of the appropriate type is required (except invoking delete on a null pointer has no effect)."

Be more specific about the "usual rules".

9b. "When a destructor is invoked for an object, the object no longer exists; if the destructor is explicitly invoked again for the same object the behavior is undefined."

Clarify "the object no longer exists".

10a. "The notation for explicit call of a destructor may be used for any simple type name. For example,

```
int* p;
// ...
p->int::~int();"
```

Explain the effect of this example. The semantics of destructor calls for non-class objects are not explicitly stated.

12.5 Free store:

2a. "When a non-array object or an array of class T is created by a new-expression, the allocation function is looked up in the scope of class T using the usual rules."

Make the "usual rules" more specific. Is this the the same as "the usual scope rules" ?

7a. "When an object is deleted by a delete-expression, the deallocation function is looked up in the scope of class of the executed destructor (see 5.3.5) using the usual rules."

Same as above.

12.6 Initialization:

1a. "A class having a user-defined constructor or having a non-trivial implicitly-declared default constructor is said to require non-trivial initialization."

The definitions of a non-trivial default constructor and of non-trivial initialization circularly depend on one another. I think this wording is difficult to understand. A suggestion is to change the definition of non-trivial initialization to:

"A class is said to require non-trivial initialization if either the class has a user-defined constructor, direct virtual base, or virtual function, or if the class has a member, array of members, or direct base that requires non-trivial initialization."

12.6.1 Explicit initialization:

1f. "Overloading of the assignment operator = has no effect on initialization."

Can this be clarified?
Does it mean the overload is permitted but ignored during initialization ?
Does the default assignment operator get used instead ?

3a. "Arrays of objects of a class with constructors use constructors in initialization (12.1) just as do individual objects."

No assertion: "just as do individual objects" is vague.

10 Derived classes:

1b. The class-name in a base-specifier must denote a previously declared class (9), which is called a direct base class for the class being declared.

This definition implies that incomplete classes are allowed in base-specifiers because incomplete class types are defined in 3.7 paragraph 2 as:

"... classes that have been declared but not defined are called incomplete types ..."

Allowing incomplete base classes is in contradiction with the ARM. Can we clarify that this is intentional ? A suggestion is to change sentence 1b as follows :

"The class-name in a base-specifier must denote a previously declared, possibly undefined, class (9), which ..."

10.1 Multiple base classes:

1a. A class may be derived from any number of base classes. For example,

Clarify "any number" ? Is this up to the limit of each implementation ? Is there a minimal number of bases to be supported ? Does an implementation have to document its limits ? If these issues are not to be clarified a suggestion is to say "A class may be derived from more than one base class" instead of the current sentence.

2a. The order of derivation is not significant except possibly for default initialization by constructor (12.1), for cleanup (12.4), and for storage layout (5.4, 9.2, 11.1).

Remove "possibly". The order is significant for these issues.

Also, can the "12.1" reference be corrected? Base class order of initialization is described in 12.6.2.

10.2 Member name lookup:

1b. Name lookup can result in an ambiguity, in which case the program is ill-formed.

Remove this sentence. Ambiguity is not defined until the 5th sentence of paragraph 2, at which point this sentence becomes totally redundant.

1c. For an id-expression, name lookup begins in the class scope of this; for a qualified-id, name lookup begins in the scope of the nested-name-specifier.

Clarify "begins". It implies name lookup continues in other scopes. What are they, and where is the order in which the scopes are searched described ? If this is described elsewhere, a reference to the location of this explanation would be helpful here.

2e. If the resulting set of declarations are not all from sub-objects of the same type, or the set has a nonstatic member and includes declarations from distinct sub-objects, there is an ambiguity and the program is ill-formed.

Can an example be added to illustrate the first case described by this sentence, that is, "declarations are not all from sub-objects of the same type" ? It would be helpful especially since there are plenty of examples of the second case.

3d. The definition of ambiguity allows a nonstatic object to be found in more than one sub-object.

Can this be clarified with an example ?

3j. An explicit or implicit conversion from a pointer to or an lvalue of a derived class to a pointer or reference to one of its base classes must unambiguously refer to a unique object representing the base class. For example,

Can this sentence be removed from here ? It is not related to name lookup, and better belongs in a section on conversions, 4.10 or 4.12 for example.

10.3 Virtual functions:

2a. If a virtual member function vf is declared in a class Base and in a class Derived, derived directly or indirectly from Base, a member function vf with the same name and same parameter list as Base::vf is declared, then Derived::vf is also virtual (whether or not it is so declared) and it overrides (41) Base::vf.

_________FootNote_________

41) A function with the same name but a different parameter list (see 13) as a virtual function is not necessarily virtual and does not override.

Integrate this footnote in the main text ? It contains normative text.

10.4 Abstract classes:

2a. An abstract class is a class that can be used only as a base class of some other class; no objects of an abstract class may be created except as sub-objects of a class derived from it.

Clarify whether "except as sub-objects of a class derived from it" is meant to include objects of the abstract base declared as members of the derived class.

2e. An abstract class may not be used as an parameter type, as a function return type, or as the typ of an explicit conversion.

Change "an" to "a".

10.5 Summary of scope rules:

Clarify the difference between 10.5 and 9.3, Scope rules for classes ? Should 10.5 be placed in a different location ?

1c. This section discusses lexical scope only; see 3.4 for an explanation of linkage issues.

Clarify the difference between lexical scope and the other scopes mentioned in the draft.

2c. Only if no access control errors are found is the type of the object, function, or enumerator named considered.

Clarify what the type is "considered" for.

3a. A name used outside any function and class or prefixed by the unary scope operator :: (and not qualified by the binary :: operator or the -> or . operators) must be the name of a global object, function, or enumerator.

Can't the name be a type or class name as well ?

- The ``one definition rule'' -- why is this so nebulous, yet critical for conformance (obey syntax rules, no diagnosable semantics violation, and the One Definition Rule)?

- Why are there no constraints? Not everything can be described in terms of positive assertions (syntax, semantics, One Definition Rule). Considering that the compliance section is weak, I believe that constraints (either embedded or a separate section) will be reinserted as soon as WG21 hardens the compliance section.

- In every area where C and C++ are the same, the wording should be the same. Why go through the aggravation of creating new (less mature) words that mean the same thing? If the words are different, the reader of both standards would ask what is the difference between the two? If they mean the same thing, then choose the C words since they have already been reviewed, refined, and standardized. This would be especially helpful for people that write combined C/C++ compilers or for people that write conformance tests. The willingness of WG21 can be captured in a simple ``litmus test" of whether they want to change their development process: where appropriate, there is no technical reason not to adopt Standard C wording and it will save time for WG21. If they choose not to adopt the wording AND provide no rationale for their decision, then we've quickly learned one thing: don't spend too much time or have high hopes on the WG21 work because they have major problems that we (WG14) cannot fix. Hopefully, this won't be the case for WG21. I recommend that this (choose Standard C wording, where appropriate) be one of the issues we ``go to the wall" (Tom requested issues like this at Plano) because it is a simple test that determines how much time we (WG14) want to spend reviewing WG21's work.

- There is no rationale. This is unacceptable. How is it possible that a 700-page document has no rationale? The document has grown 36% since the last release (comparing 94-0098 to 94-0158). This would be unthinkable in any software engineering project (producing a huge system with no specifications, design documents, etc.). The book on the evolution of C++ does not substitute as a rationale.

- The document provides little help to the reader to help review the document. For example, not knowing what change bars mean (one bar vs. two bars) doesn't help me understand what has changed. There is no table of contents. Considering that there is approximately 1300 entries (26 pages) in the table of contents (I wrote a script to extract them myself), this would be helpful for guiding the reader. Similarly, an index would be helpful.

- Some editorial and administrative issues are considered low priority rather than high priority. Tom had suggested we review the document at the ``functionality level" before we review the lower levels. I agree with Tom on this approach for reviewing documents *as long as the development process uses the same approach*. This (the development process doesn't use the same approach) is where the problem is with respect to reviewing this document.

(1) The document is still in development mode (still growing) rather than in maintenance mode

(stablized). For work that is in development mode, there are 2 ways to assess quality:

- We can compare modules of the implementation (writing the standard) to higher level documentation, such as requirements, functional, and design specifications (e.g., problem definition, conceptual model, rationale, etc.). As each module (chapter, topic, etc.) completes we can compare the result to the higher-level specifications. This would be okay if there were such documents.

- We can wait until development is complete (i.e., no more substantial additions) and start our quality assessment. The drawback to this approach is that if we wait too long, the development may have gotten too far off course. This is why several of you have suggested giving input to WG21 as early as possible. Even so, with this approach, how would we measure quality? For many people, quality is ``whatever I want''. Putting that in technical terms, it's a requirements specification or functional specification. Can someone point me to such a document that tells me how I know that the C++ Standard will be complete (from a technical perspective)? We may be able to write a conformance test on the output (i.e., the C++ Standard), but not the process (i.e., is the right development process being used?).

(2) Once the document transitions into maintenance mode, there should be fewer changes. In this mode, we should be able to correlate (hopefully, small) changes in the document to committee decisions. The minutes or rationale will provide understanding of the changes.

(3) Probably the most important work to stabilize is the compliance, basic definitions, and conceptual model. It doesn't matter that there are 20 other chapters -- they are all likely to change substantially because they are dependent upon compliances, basic definitions, and conceptual model.

(4) Once compliance, basic definitions, and conceptual model have been refined, the rationale and semantics should be addressed.

(5) The development of other sections (e.g., library) could be developed in parallel, determined by their dependencies. The division between continuing work or postponing might be: if a library is dependent upon new features that have been added since the ARM, then that work should be postponed;

otherwise, libraries dependent on older, established features could continue work. This scheme might not fit exactly on top of the WG21 work list, but there should be *some* decision process that distinguishes between low-risk work that can proceed in parallel and other work that is high-risk or has dependencies on compliance, basic definitions, and conceptual model.

- My experience, as a manager who has had to jump into fires like this, suggests the following action plan for WG21:

(1) Stop doing new work.

(2) Put the document under change control. No changes to the document unless the committee approves a change.

(3) The focus of the meetings (and the work) should be first resolving basic issues:

- Compliance -- How it is determined.

- Basic definitions -- Axioms. Search for all phrases in the document that are undefined or defined poorly (e.g., usual mathematical rules, patterns, usual arithmetic conversions, common type, usual function call implementation). Creating an index is an easy way to locate all undefined terminology and loose ends (as a side effect, the document will now have an index).

- Remove non-standards wording or flag it as a work item (e.g., ``usually, expression statements are assignments or function calls", ``binary_op is assumed not to cause side effects").

- Rationale. As the committee makes each change, record the reasoning in a rationale document. Since the document is huge to begin with, adding another 300 pages of embedded rationale won't make the document much harder to read. However, embedded rationale will avoid one of the pitfalls of the existing document: locating things, i.e., you won't have to search far to find the rationale since it will be embedded.

(4) The other parts of the document, e.g., libraries and new features, can be worked on in parallel by focusing only on rationale. Even if the basic parts of the C++ Standard (see #3 above) change significantly, the rationale of these sections won't change too much. However, the standards wording of these sections is likely to change significantly if the basic parts change. Thus, the people working on these sections should concentrate on the work that

is lower risk: writing rationale prior to writing standards words.

(5) Once the basic parts (see #3 above) have been stabilized (e.g., 2/3 committee vote) can the remaining work proceed.

The penalty for not choosing this approach is that much work will be wasted, re-done, or left in the document with poor quality (quality to be measured by the number of requests for interpretation, defect reports, untestable features, etc.).

<<<Initial questions and comments:>>>

1) Is there a rationale document? If not, I strongly suggest you pull one together to head-off many questions during the public comment period. Questions such as:

2) Why are there no constraints, semantics, and example sections like the C Standard? Without them, everything is mixed together.

3) Given that the C Standard was one of the base documents, why do the sections common to both languages have no common text? That is, why wasn't the text from the C Standard used? Given the considerable differences in wording in the sections covering common topics it is hard for people familiar with the C standard to follow the C++ draft. This is an important problem given that we (the C standards community) are about to start revision of the standard and will be looking at possible C++ features to add. It will be harder to find all the relevant parts relating to any topic. It will also be harder for future C++ revisions to incorporate new features from future C standards. And it will be harder for both committees to deal with interpretations of the parts common to both languages. The differences in style and wording also make it harder for any long-term integration of the two languages.

I know we may be beyond the point of doing something about this but I can't help but feel uncomfortable about it.

<<<General editorial comments:>>>

4) In numerous places, an italicized term is followed by a parenthesized section number. However, the two are not sufficiently separated. You need to add a small amount of space between theee to make it easier to read.
3. paras 4 and 5 contain examples.

5) The term `variable' is used when you probably should say `object.'

6) The use of words `may' and `can.' Rossler spent a lot of time getting this right in the initial C draft. You should check each such use to see whether you mean `must/shall' or `is allowed to be.' `may' is a bad word to use in a standard. See 3.3.7 para 1 for an example.
7) Change `International standard' to `standard' throughout. ISO doesn't publish national standards. In any event, you say this once in the intro and then refer to it as `the standard.'

8) I found the use of) after footnote numbers unusual and quite

distracting.

9) Adjectives made up of multiple words (like `floating-point') should be hyphenated.

10) In many places, the plural case is unnecessarily used instead of the singular (that typically is used in ISO C). Given the nature of English, this can lead to ambiguity. For example, in 3.8.2 Compound types

- functions, which have parameters of given types and return objects of a given type, 8.3.5;

Saying that `functions return objects' includes the possibility of one function returning multiple objects, which, of course, is not permitted. Using the singular form will make the words much more precise.

11) More than a few section number references are not parenthesized. It appears they should be unless they are being referenced directly in the narrative. See 3.8.2 Compound Types, for example.

12) Keep in mind that more than a few readers of your standard will not have English as their first or main language. Therefore, I suggest you avoid esoteric or uncommonly used words such as `engender' and `referent.'

13) All remaining occurrences of `implementation-dependent' should be changed to `implementation-defined.'

14) Remove all `file(s)' from `header file(s).' Presumably, like ISO C, headers need not be files.

15) To paraphrase Larry Rossler ``If you try to state all the possibilities in each case, you'll miss some here and there. It's better to establish the rule in one place and refer to that rule in all other places rather than trying to restate the rule." Too many references partially restate rules, constraints, etc.

<<<Comments referring to a specific clause/section>>>

16) 1.1 Scope [intro.scope]

2 C++ is a general purpose programming language based on the C programming language as described in ISO/IEC 9899 (1.2). In addition to the facilities provided by C, C++ provides ...

However, this is contradicted by Annex C (informative) which states:

2 C++ is based on C (K&R78) and adopts most of the changes specified by the ISO C standard. Converting programs among C++, K&R C, and ISO C

The first quote says C++ provides C plus extra features while the second says most of Standard C is supported by C++. Which is it?

17) 1.1 Scope [intro.scope]

2 ...
management operators, function argument checking and type conversion,

Standard C also provides function argument checking and type conversion. You are mixing Standard C with K&R C in para 2. You need to get this straight. I think it is inappropriate to make any references to K&R C in the standard proper; in the Annex, yes, since that's useful historical information.

18) Footnote #1 printed on the wrong page.

19) 1.3 Definitions [intro.defs]

- implementation-defined behavior: ...
  The range of possible behaviors is delineated by the standard.

- unspecified behavior: ...
  The range of possible behaviors is delineated by the standard.

Do you REALLY mean you are going to enumerate the choices an implementation has for these?

20) 1.4 Syntax notation [syntax]

2 Names for syntactic categories have generally been chosen according to the following rules:

Change X's to Xs.

21) 1.5 The C++ memory model [intro.memory]

4 Certain types have alignment restrictions. An object of one of those types may appear only at an address that is divisible by a particular integer.

Addresses are not integers so don't talk about an address being divisible. Say that certain objects are required to be aligned on particular address boundaries. See ISO C words.

22) 1.6 Processor compliance [intro.compliance]

1 Every conforming C++ processor shall, ...
at least one diagnostic error message when presented with any ill-

Strike `error'; the term you define in 1.3 doesn't include it.

23) 1.6 Processor compliance [intro.compliance]

1 ...
at least one diagnostic error message when presented with any ill-formed program that contains a violation of any rule that is

2 Well-formed C++ programs are those that are constructed according to

Where do you define `ill-formed' and `well-formed' program?

24) 1.7 Program execution [intro.execution]

4 Certain other operations are described in this International Standard

Drop `International'; it's redundant.

25) 2 Lexical conventions [lex]

1 A C++ program need not all be translated at the same time. The text of the program is kept in units called source files in this standard.

A source file together with all the headers (17.1.2) and source files included (16.2) via the preprocessing directive #include, less any source lines skipped by any of the conditional inclusion (16.1) preprocessing directives, is called a translation unit. Previously translated translation units may be preserved individually or in libraries. The separate translation units of a program communicate (3.4) by (for example) calls to functions whose identifiers have external linkage, manipulation of objects whose identifiers have external linkage, or manipulation of data files. Translation units may be separately translated and then later linked to produce an executable program. (3.4).

I suggest rearranging the order of sentences in this para to something like the following; as is, it doesn't read well:

1 The text of a program is kept in units called source files in this standard. A source file together with all the headers (17.1.2) and source files included (16.2) via the preprocessing directive #include, less any source lines skipped by any of the conditional inclusion (16.1) preprocessing directives, is called a translation unit. Translation units may be separately translated and then later linked to produce an executable program. (3.4). The translation units making up a C++ program need not all be translated at the same time. Previously translated translation units may be preserved individually or in libraries. The translation units of a program communicate (3.4) by (for example) calls to functions whose identifiers have external linkage, manipulation of objects whose identifiers have external linkage, or manipulation of data files.

26) 2.1 Phases of translation [lex.phases]

I see no reference to digraphs here. Is one appropriate?

27) 2.1 Phases of translation [lex.phases]

7 White-space characters separating tokens are no longer significant. Each preprocessing token is converted into a token. (See 2.4). The resulting tokens are syntactically and semantically analyzed and translated. The result of this process starting from a single source file is called a translation unit.

The term `translation unit' was defined in 2. At least you should drop the italics since this is not the first use of this term. The solution might be to make the last sentence a footnote instead since it's a `by the way,...'

28) 2.2 Trigraph sequences [lex.trigraph]

1 Before any other processing takes place, each occurrence of one of the

Use `During Phase 1 ...' instead. Better still, use ISO C words. We spent a lot of time getting them the way they are.

29) 2.4 Digraph sequences [lex.7digraph]

1 Alternate representations are provided for the operators and punctuators whose primary representations use the national characters.

What is a `national character?"

30) 2.4 Digraph sequences [lex.7digraph]

2 In translation phase 3 (2.1) the digraphs are recognized as
...

Table 2-identifiers that are treated as operators

alternate	primary	alternate	primary	alternate	primary

...

Table contains tokens other than identifiers so change the table caption.

31) 2.5 Tokens [lex.token]

1 There are five kinds of tokens: identifiers, keywords, literals (which include strings and character and numeric constants), operators, and other separators.
...
below, are ignored except as they serve to separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords, and literals.

For `other separators' do you mean punctuators? You may as well use the term you have in the grammar. You can't mean white space since this is an enumerated list of token types.

Drop last sentence; you already say that in 2.3 para 2 and give an example in para 5.

32) 2.6 Comments [lex.comment]

1 The characters /* start a comment, which terminates with the characters */. These comments do not nest. The characters // start a comment, which terminates the next new-line character. If there is a

... which terminates AT the next new-line character.

33) 2.7 Identifiers [lex.name]

1 An identifier ...
The first character must be a letter; the underscore _ counts as a letter. ...

How about ``The first character must be a letter or underscore (_)."

34) 2.8 Keywords [lex.key]

1 The identifiers shown in Table 3 are reserved for use as keywords, and

Identifiers can't be keywords. Say `The tokens shown ...' or `The names shown ...'

35) 2.8 Keywords [lex.key]

3 In addition, identifiers containing a double underscore (__) are reserved for use by C++ implementations and standard libraries and should be avoided by users; no diagnostic is required.

Since this is about identifiers and NOT about keywords, it should be moved to 2.7.

36) 2.8 Keywords [lex.key]

Paras 4-7 don't belong in 2.8 but, rather, should be in sections of their own.

37) 2.8 Keywords [lex.key]

4 The ASCII representation of C++ programs uses as operators or for punctuation the characters shown in Table 5.

ASCII? Is this standard dependent on ASCII?

38) 2.8 Keywords [lex.key]

7 Certain implementation-dependent properties, such as the type of a sizeof (5.3.3) and the ranges of fundamental types (3.8.1), are

What is a sizeof? `... such as the type of the result produced by the sizeof operator' perhaps?

39) 2.9 Literals [lex.literal]

In general, the wording in this section is pretty loose. I strongly suggest you look at using at least some of the ISO C words.

40) 2.9 Literals [lex.literal]

1 There are several kinds of literals (often referred to as constants).

I find the use of `literal' instead of `constant' confusing, particularly when you look at the index entries.

In C, a string literal is a separate token category from constants. It seems confusing for them to be grouped together in C++ particularly when you say literals are referred to as constants yet string literals might be modifiable.

Why aren't enumerations addressed in this section?

41) 2.9.2 Character literals [lex.ccon]

Unlike ISO C, there is no mention of multibyte characters here.

42) 2.9.2 Character literals [lex.ccon]

1 A character literal is one or more characters enclosed in single quotes, as in 'x', optionally preceded by the letter L, as in L'x'. Single character literals that do not begin with L have type char, with value equal to the numerical value of the character in the machine's character set. Multicharacter literals that do not begin with L have type int and implementation-defined value.

You define `character literal' and then proceed to use `single character literal' and `multicharacter literal.' I think you mean `single-character character literal' (or `a character literal containing one character') and `multi-character character literal' (or `a character literal containing multiple characters').

43) 2.9.2 Character literals [lex.ccon]

2 A character literal that begins with the letter L, such as L'ab', is a wide-character literal. ...
They are intended for character sets where a character does not fit into a single byte.

L'ab' is a bad example since it is a multicharacter character literal as well as a wide character literal. Use L'a' instead since your example IS NOT a character that needs multiple bytes. Besides, L'a' is portable.

44) 2.9.2 Character literals [lex.ccon]

3 Certain nongraphic characters, the single quote ', the double quote ", ?, and the backslash \, may be represented according to Table 9.

Table 9-escape sequences

new-line	NL (LF)	\n
horizontal tab	HT	\t
vertical tab	VT	\v
backspace	BS	\b
carriage return	CR	\r
form feed	FF	\f
alert	BEL	\a
backslash	\	\\
question mark	?	\?
single quote	'	\'
double quote	"	\"
octal number	ooo	\ooo
hex number	hhh	\xhhh

I don't understand the purpose of the second column in this table. Are you introducing abbreviations for these characters? If so, where do you use them? I suggest you remove them, particularly (LF). Also, set column 3 in constant width.

45) 2.9.2 Character literals [lex.ccon]

4 The escape \ooo consists of the backslash followed by one, two, or

The escape SEQUENCE \ooo ...

46) 2.9.2 Character literals [lex.ccon]

4 The escape \ooo consists of the backslash followed by one, two, or ... The value of a
character literal is implementation dependent if it exceeds that of the largest char.

What is `the largest char?' Do you mean CHAR_MAX defined in limits.h? Needs better wording.

47) 2.9.3 Floating literals [lex.fcon]

1 A floating literal consists of an integer part, a decimal point, a fraction part, an e or E, an optionally signed integer exponent, and an optional type suffix. The integer and fraction parts both consist of a sequence of decimal (base ten) digits. Either the integer part

You start by saying a floating literal needs all of these things. Then two sentences later you say ``well actually, some of these are optional." Needs better wording.

48) 2.9.4 String literals [lex.string]

1 A string literal is a sequence of characters (as defined in 2.9.2)
... Whether all string literals
are distinct (that is, are stored in nonoverlapping objects) is
implementation dependent.

ISO C says this is undefined or unspecified (I'm not sure which category since it doesn't say anything). Do you mean to require the vendor to document this? If so, it should be listed as a difference from ISO C in annex C.

49) 2.9.4 String literals [lex.string]

2 ... Concatenation of ordinary and wide-character string literals
is undefined.

Imprecise English. Does ``I drive red and blue cars" mean ``I drive cars that are red and blue" or ``I drive red cars and I drive blue cars"? What you mean is that you can't concatenate mixed flavors of strings. See ISO C words.

50) 2.9.4 String literals [lex.string]

4 After any necessary concatenation '\0' is appended so that programs

Wouldn't a wide string literal have L'\0' appended?

51) 2.9.4 String literals [lex.string]

4 After any necessary concatenation '\0' is appended so that programs
that scan a string can find its end.

What about "abc\0xyz"? This is a well-formed string literal but it's not a string in the usual sense; that is, you can't find its end. ISO C has a footnote re this.

52) 2.9.5 Boolean literals [lex.bool]

1 The Boolean literals ... They are not lvalues.

True they are not lvalues. However, you don't say if character constants, et al, are lvalues. Maybe this should be stated at the beginning of 2.9. But a string literal is an lvalue, no? Not necessarily a modifiable one though.

53) 3 Basic concepts [basic]

5 Every name ...
which that name can possibly be valid. In general, each particular

What is a `valid' name?

54) 3 Basic concepts [basic]

6 ... same as its potential scope..

Remove one of the trailing periods.

55) 3.1 Declarations and definitions [basic.def]

3 The following, for example, are definitions:

```
              ...
    static int y;          // declares static data member y
```

As the comment suggests, this is NOT a definition yet the intro uses that term. Either it doesn't belong here or the intro words need to be changed.

56) 3.1 Declarations and definitions [basic.def]

4 ...

```
    main()
    {
      C a;
      C b=a;
      b=a;
    }
```

Add spaces around each =.

57) 3.2 One definition rule [basic.def.odr]

3 Exactly one definition in a program is required for a non-local
 variable with static storage duration, unless it has a builtin type or

How can a non-local object have other than static storage duration?

58) 3.2 One definition rule [basic.def.odr]

4 At least one definition of a class is required in a translation unit
 if the class is used other than in the formation of a pointer type.

I think C++ allows benign redefinition of a class whereas C does not. If that is the case, do you need to say that if multiple definitions are present, they must be equivalent (identical?)

59) 3.2 One definition rule [basic.def.odr]

4 ...

```
 +-------              BEGIN BOX 9              -------+
 There may be other situations that do not require a class to be
 defined: extern declarations (i.e. "extern X x;"), declaration of
 static members, others???
 +-------              END BOX 9                -------+
```

I know this para is talking about classes but what about enum type definitions such as

```
    enum {....} f(...) { }
```

Are they allowed?

60) 3.3.2 Function prototype scope [basic.scope.proto]

1 In a function declaration, names of parameters (if supplied) have function prototype scope, which terminates at the end of the function declarator.

Consider the following:

```
void (*signal(int sig, void (*func)(int i)))(int i);
```

I presume this is invalid since the identifier i appears twice, even though each applies to a different function `prototype' level.

61) 3.3.4 File scope [basic.file.scope]

1 ...
Names declared with file scope are said to be global.

I think it is a bad idea to use the term `global' when it really might have internal linkage.

62) 3.4 Program and linkage [basic.link]

6 Typedef names (7.1.3), enumerators (7.2), and template names (14) do not have external linkage.

It seems to me that only identifiers designating objects and functions have linkage. If that is true, either say nothing about linkage and typedef, et al, or say linkage doesn't apply to them. But to say they don't have external linkage leaves one wondering ``what linkage do they have?"

63) 3.5.1 Main function [basic.start.main]

1 A program shall contain a function called main, which is the designated start of the program.

Does C++ support the notion of C's freestanding environment? If so, you need words to the effect that the main program need not be called `main.' If not, why not? (rationale?)

64) 3.5.1 Main function [basic.start.main]

2 This function is not predefined by the compiler, it cannot be overloaded, and its type is implementation dependent. The two examples below are allowed on any implementation. It is recommended that any further (optional) parameters be added after argv. The function main() may be defined as

```
int main() { /* ... */ }
```

or

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

Sine the type of main is implementation-defined, I read that as saying I can have an implementation in which main returns type void or whatever. Is that what you want?

Why not REQUIRE extra arguments to follow argc and argv? I see no value in letting the implementor change the long-establish rule of argc first then argv, then envp, etc. Besides, should a standard make recommendations? In ISO C we suggest style by declaring certain uses obsolescent. For example, the position of storage class and type qualifier keywords relative to each other.

65) 3.5.1 Main function [basic.start.main]

2 This function ...
 In the latter form argc shall be the number of arguments passed to
 the program from an environment in which the program is run. If argc

Change `an environment' to `the environment.'

66) 3.5.1 Main function [basic.start.main]

2 This function ...
 is nonzero these arguments shall be supplied as zero-terminated
 strings in argv[0] through argv[argc-1] and argv[0] shall be the

The term `zero-terminated strings' sounds like it should be `strings.' Once you have defined a term there is no need to spell it out each usage. Actually, where do you define the term `string?' (I don't mean `string literal' either.) I think ISO C does it in the library somewhere.

67) 3.5.1 Main function [basic.start.main]

2 This function ...
 the program from an environment in which the program is run. If argc
 is nonzero these arguments shall be supplied as zero-terminated
 strings in argv[0] through argv[argc-1] and argv[0] shall be the
 name used to invoke the program or "". It is guaranteed that

Like ISO C, you need to have the constraint that ``The value of argc shall be nonnegative." As written, if argc is -2, argv[-3] results in undefined behavior.

68) 3.5.1 Main function [basic.start.main]

3 The function main() shall not be called from within a program. The

I know what you mean but it's hard to execute a program if it can never get started. You need words to say something like ``The function main() shall not be called except by the environment." or ``... shall not be called by the user's program."

You need rationale as to why the linkage of main needs to be implementation-defined. It isn't clear to me.

69) 3.5.1 Main function [basic.start.main]

4 Calling the function

```
void exit(int);
```

 declared in <stdlib.h> (17.2.4.4) terminates the program without
 leaving the current block and hence without destroying any local
 variables (12.4). The argument value is returned to the program's
 environment as the value of the program.

5 A return statement in main() has the effect of leaving the main

function (destroying any local variables) and calling exit() with the return value as the argument. If control reaches the end of main without encountering a return statement, the effect is that of executing

    return 0;

Since you call-out exit specifically, why not abort too? Perhaps a scaled-back approach to paras 4 and 5 (like in ISO C) would be better. That is, strike para 4 altogether since that info is provided in the library.

The wording implies that return; DOES NOT get turned into return 0; That only happens if NO return statement is seen. Is that what you want?

70) 3.7.2 Automatic storage duration [basic.stc.auto]

1 Local objects not declared static or explicitly declared auto have automatic storage duration and are associated with an invocation of a block.

... declared auto OR REGISTER have ...

71) 3.7.2 Automatic storage duration [basic.stc.auto]

3 A named automatic object with a constructor or destructor with side effects may not be destroyed before the end of its block, nor may it be eliminated even if it appears to be unused.

Change both `mays' to `shalls' to match wording in para 2 of previous section, which says:

2 Note that if an object of static storage class has a constructor or a destructor with side effects, it shall not be eliminated even if it appears to be unused.

72) 3.8 Types [basic.types]

1 There are two kinds of types: fundamental types and compound types. Types may describe objects, references (8.3.2), or functions (8.3.5).

Strike `may.' (I'm assuming types describe these ONLY; there are no other things types can describe, are there?)

73) 3.8 Types [basic.types]

2 Arrays ...
an instance of the type is unknown. Also, the void type represents an empty set of values, so that no objects of type void ever exist; void is an incomplete type. The term incompletely-defined object type

Drop the detailed discussion of void since you cover it in para 11.

74) 3.8 Types [basic.types]

2 Arrays of unknown ...
is a synonym for incomplete type; the term completely-defined object type is a synonym for complete type;

Does the last sentence imply the two terms can be used interchangeably?

The type of int f(void) is complete yet it doesn't involve any objects!

75) 3.8 Types [basic.types]

4 Variables ...

```
        void bar()
        {
           xp = &x;          // okay; type is ``pointer to X"
           arrp = &arr;       // ill-formed: different types
```

I don't see why the second assignment is ill-formed? Can't a pointer to an unknown size array be initialized with the address of a known size array? I believe it is in ISO C based on the rules of compatible types. If this is not true in C++, where is it stated? Without a discussion of C's `compatible and composite types' I find the issue of type compatibility in C++ hard to figure out. (BTW, my six C++ compilers are evenly divided on this one, and almost all of my C compilers say it's OK.)

76) 3.8.1 Fundamental types [basic.fundamental]

1 There are several fundamental types. The standard header <limits.h> specifies the largest and smallest values of each for an implementation.

I presume floating-point types are fundamental types too so why not mention float.h? Better still, why mention either header here except maybe in a footnote?

I see you mention float.h in para 9. Maybe you should keep the reference in para 1 but put it in a sentence about integer types, not fundamental types.

77) 3.8.1 Fundamental types [basic.fundamental]

2 Objects declared as characters (char) are large enough to store any member of the implementation's basic character set. If a character from this set is stored in a character variable, its value is equivalent to the integer code of that character. Characters may be explicitly declared unsigned or signed. Plain char, signed char, and unsigned char are three distinct types. A char, a signed char, and an unsigned char consume the same amount of space.

Be careful using the term `character.' This is a constant source of confusion in ISO C. I suggest you drop that word from the first sentence and just say char like ISO C does in this section.

BTW, where do you define `basic character set?'

I suggest you replace `consume' with `occupy.'

78) 3.8.1 Fundamental types [basic.fundamental]

3 An enumeration comprises a set of named integer constant values. Each distinct enumeration constitutes a different enumerated type. Each constant has the type of its enumeration.

I thought that, unlike C, in C++, enumerations are not integer types. If that is the case you should strike the word `integer' from the first sentence. If they are integer constants they can't also have the type of their enumeration. In enum {red}; int i = red;, there is an implicit conversion from enumerated type to int; so they are compatible in this direction.

79) 3.8.1 Fundamental types [basic.fundamental]

4 There are four signed integer types: signed char, short int, int, and long int. In this list, each type provides at least as much storage as those preceding it in the list, but the implementation may otherwise make any of them equal in storage size. Plain ints have the natural size suggested by the machine architecture; the other signed integer types are provided to meet special needs.

What does `equal in storage size' mean? I think you mean each must accommodate the range of previous members of that list. Be careful to not confuse the amount of memory allocated to an object and the range of values of an object. ISO C does not require that all bits allocated to an object actually be used to represent that object. For example, on 32-bit RISC machines, an 80-bit IEEE long double typically has sizeof == 12 for alignment reasons. Also, on CRAY 2s, sizeof(short) == 8 yet only 32 bits are used.

Bottom line: Are you requiring that sizeof(short) <= sizeof(int) or,
Are you requiring that SHRT_MIN <= INT_MIN or,
Both?

A plain int can have any size an implementer wants it to have provided they satisfy the minimum criteria. The discussion of `natural size' and `special needs' belongs in the rationale, not in the standard proper.

80) 3.8.1 Fundamental types [basic.fundamental]

6 For each of the signed integer types, there exists a corresponding (but different) signed integer type: unsigned char, unsigned short int, unsigned int, and unsigned long int, each of which occupies

... (but different) UNsigned integer ...

Also see comments re allocation size in para 4 above.

81) 3.8.1 Fundamental types [basic.fundamental]

6 For each ...
(1.5) as the corresponding signed integer type.[4] An alignment requirement is an implementation-dependent restriction on the value of a pointer to an object of a given type (5.4, 1.5).

Drop the last sentence since alignment was defined in clause 1.

82) 3.8.1 Fundamental types [basic.fundamental]

7 Unsigned integers, declared unsigned, obey the laws of arithmetic

You mean types containing `unsigned' not just the type `unsigned [int]' The solution is to strike `, declared unsigned.'

83) 3.8.1 Fundamental types [basic.fundamental]

8 Values of type bool can be either true or false.[5] There are no signed, unsigned, short, or long bool types or values. As

described below, bool values behave as integral types. Thus, for example, they participate in integral promotions (4.1, 5.2.3). Although values of type bool generally behave as signed integers, for example by promoting (4.1) to int instead of unsigned int, a bool value can successfully be stored in a bit-field of any (nonzero) size.

Why say the second sentence? Can I have a bool int since you don't exclude that?

They `generally behave...' When don't they? What if the bit-field is signed?

84) 3.8.1 Fundamental types [basic.fundamental]

9 There are three floating types: float, double, and long double. The type double provides at least as much precision as float, and the type long double provides at least as much precision as double. Each implementation defines the characteristics of the fundamental floating point types in the standard header <float.h>.

What about range? It's not much good if a double has more precision but less range than a float. See ISO C words.

85) 3.8.1 Fundamental types [basic.fundamental]

10Types bool, char, and the signed and unsigned integer types are collectively called integral types. A synonym for integral type is integer type. Enumerations (7.2) are not integral, but they can be promoted (4.1) to signed or unsigned int. Integral and floating types are collectively called arithmetic types.

So are you saying an enumeration is or is not an arithmetic type. If it is not, then why mention enumerations here at all? If it is, say so.

Can an enumeration be promoted to signed/unsigned long too? Doesn't this stuff belong in `conversions?'

86) 3.8.2 Compound types [basic.compound]

The descriptions in the bullet list say very little. I'd add some of the stuff from the `derived type' section of ISO C to flesh them out.

87) 3.8.2 Compound types [basic.compound]

1 There is a conceptually infinite number of compound types constructed

Drop `conceptually.'

88) 3.8.2 Compound types [basic.compound]

1 There ...

- functions, which have parameters of given types and return objects of a given type, 8.3.5;

By saying `return object' you preclude void. See the ISO C words.

89) 3.8.2 Compound types [basic.compound]

- pointers to objects or functions (including static members of classes) of a given type, 8.3.1;

What about pointers to incomplete types?

90) 3.8.2 Compound types [basic.compound]

- references to objects or functions of a given type, 8.3.2;

Can you have a reference to an incomplete type?

91) 3.8.2 Compound types [basic.compound]

- unions, which are classes capable of containing objects of different types at different times, 9.6;

Not precise enough. See ISO C words. You need to say that a union contains only one of a set of members. The problem is compounded by the use of the plural.

92) 3.8.2 Compound types [basic.compound]

3 Any type ...
The cv-qualified or unqualified versions of a type are distinct

I think `or' should be `and.'

93) 3.8.2 Compound types [basic.compound]

4 A pointer to objects of a type T is referred to as a pointer to T.

Change `a type T' to `type T.'

94) 3.8.2 Compound types [basic.compound]

5 Objects of cv-qualified (3.8.3) or unqualified type void* (pointer to void), can be used to point to objects of unknown type. A void* must have enough bits to hold any object pointer.

Is a void * required to have exactly the same representation and alignment requirements as char * like in ISO C? If so, say so. If not, give rationale for why not.

The last sentence isn't very standards-like. How about ``A void* must be large enough so it can accurately represent the address of any object." There are probably better words somewhere in ISO C.

95) 3.8.2 Compound types [basic.compound]

6 Except for pointers to static members, text referring to pointers does not apply to pointers to members.

I'd move this earlier in the section since it changes the way you read everything relating to pointers.

96) 3.8.3 CV-qualifiers [basic.type.qualifier]

1 There ...
_________Footnote_________ ...
(that is, the processor may not assume that the object continues to hold a previously held value).

Change `processor" to `implementation' to be consistent and to avoid confusion with CPU.

97) 3.8.3 CV-qualifiers [basic.type.qualifier]

3 A pointer ... For example, a pointer to const int may point to an
  unqualified int, but a well-

`Unqualified' is the opposite of `qualified.' Since you have specifically introduced the term `cv-qualified' it's opposite must be `non-cv-qualified.' Yep it's messy but that what we had to do in ISO C. And if you don't want to do that then why not use `qualified' instead of `cv-qualified?' If the cv prefix is important then its ALWAYS important, in both positive and negative senses.

98) 3.8.4 Type names [basic.type.name]

1 Fundamental and compound types can be given names by the typedef

... can be given ALTERNATE names ...

99) 3.9 Lvalues and rvalues [basic.lval]

Where is `rvalue' defined?

It is interesting to note that ISO C contains `rvalue' in only one place; a footnote. We have them; we just don't call them that (confusing) term.

100) 3.9 Lvalues and rvalues [basic.lval]

1 Every expression is either an lvalue or rvalue.

What about a void expression?

101) 3.9 Lvalues and rvalues [basic.lval]

2 An lvalue refers to an object or function. Some rvalue expressions-

I don't understand how a function can be an lvalue. Certainly, a function can return an lvalue. I can't find any use of lvalue that doesn't refer to an object.

In ISO C, an lvalue designates an object only.

102) 3.9 Lvalues and rvalues [basic.lval]

4 Some builtin operators expect lvalue operands, for example the builtin
  assignment operators all expect their left hand operands to be
  lvalues. Other builtin operators yield rvalues, and some expect them.

While there are occasional references to the terms `modifiable lvalue' and `non-modifiable lvalue' they are not used enough. For example, the assignment operators require their left operand to be a modifiable lvalue. Also, I don't see where these terms are defined in this section.

Wait, move para 14 way earlier in the section and use the term `modifiable' consistently.

103) 3.9 Lvalues and rvalues [basic.lval]

8 Whenever an lvalue that refers to a non-array[11] non-class object appears in a context where an lvalue is not expected, the value contained in the referenced object is used. When this occurs, the

How about `designated object' instead of `referenced object' since reference already has another meaning.

104) 3.9 Lvalues and rvalues [basic.lval]
105
8 Whenever ...

```
        const int* cip;
        int i = *cip   // "*cip" has type int
```

If this type is incomplete, the program is ill-formed.

```
+-------              BEGIN BOX 21                 -------+
In C this is undefined.
+-------               END BOX 21                 -------+
```

Assuming you add a semicolon to the second line, and you initialize cip to some reasonable place, I think this is very well defined in C; the value of the underlying const int is assigned to i. What about it is undefined?

Some general points.

1)
There are lots of places within the draft where wording is essentially rationale or commentary, rather than standard. In a strict ISO sense this should not be present however rather than try and remove all of this it could simply be labelled as overview for each section and identify it as not part of the standard up front somewhere, rather like a foot note.

2)
When is a diagnostic message required ?

According to Plum, the Working Group actually agreed in Munich that _all_requirements require diagnostic message except for those that are specifically indicated as "no diagnostic required". But the introduction has not been updatedto refelect this. This action needs to be performed to make the points where the draft says "no diagnostic required" make any sense.

3)
I believe it to be the intent that when either
A program is ill formed or an initialisation is ill formed
a diagnostic is required, The draft does not appear to say this anywhere.

4)
The draft uses the term must and may in many places verbage that is inappropriate in a standard. I am sure many others will comment on this but verbal usage in a standard should conform to the following format:

Shall - requirement
May - requirement with options
Should - guidance only, no requirement

Note the term "must" is reserved in many countries for regulatory requirements rather than standards. It would be impossible to publish this wording "as is" for a standard in the United Kingdom.

[intro]

As per box 1 References to ISO 9899 are incomplete from an ISO perspective

The reference should read ISO/IEC 9899:1990 Programming Languages C

[intro.defs]

Within the document frequent reference is made to the term "rules" which is not in the definitions, I would suggest adding something on the following lines:

-Rules: A group of related requirements

[intro.execution]

Para 6 volatile should not be in a different font in this case as it is using volatile in the sense of generic changeable data rather than the qualified type.

Space missing between Forexpression(5.18)

[lex]
No definition of the term libraries. (Means directories to IBM users)
No definition of linked

Suggest the following added to definitions

Library, a set of translation units
Linked, Linking; pseudonym for translation phase 8

[basic]
Para 3 hyphen missing from subobject under description of entity

Para 5 reads like a bureaucrats dream.
Would it be possible to simplify the wording ?

[basic.start..term]
Para 2 Should at best be a note, it is not the purpose of the C++ standard to describe how a joint C/C++ environment functions.
See my general note 1.

Footnote 42 defines the acronym POD this should be in the definitions section.

[basic.stc.static]
Para 2 Needs a reference to elimination

[conv......]
There is no section on conversion of an enumeration type although it is described later under Static cast

[expr.prim]

Para 4
There is an incomplete sentence. (The results is an lvalue if the identifier is.??????)

Consider the following code:

```
// File 1
int I =16;        // Global I ???

// File 2

static int I=20;  // File scope I hides I=16 does this count as hidden


extern int I;     // refers to static I in presence of static otherwise
                // would be I =16

int x = ::I;      // does x = 20 or 16.
```

i.e.[basic.scope.exqual] and[expr.prim] Para 4 say similar but not identical things, and neither is conclusive.

[expr.call]

Para 10 Recursive calls are permitted.

A reference to[basic.start.main] Para 3 which states that main is not recursive would be helpful.

[expr.unary.op]

Para 5 typo shal
Para 8 use of must

[expr.sizeof]

Para 1 second Para, The sizeof operator may not, suggest change to shall not.

Para 5 change to Types shall not be defined in a sizeof expression.

[expr.new]
Para 16 poor English.

Access and ambiguity control are not done
I suggest using the word performed

[expr.mptr.oper]
Para 4 last sentence
suggest change the result is undefined to
the result is undefined behaviour.

[expr.mul]
Para 2 change must 's to shall 's

[expr.rel]

Para 1 ...but this fact is not very useful
suggest move to foot note or remove ....since it not vey usefull!

Para 2 & 3 replace must's with shall's
Para 3 suggest change higher to greater

suggest add
are
....provided the two members are not separated by an access-specific label.

[expr.eq]

term truth-value is not defined, suggest add term to lex.bool

[expr.log.and]

Para 1 tells us that & does not guarantee left to right evaluation
The draft should tell us this under[expr.bit.and] rather than[expr.log.and] Similar story for logical OR operator

[expr.cond]
change must to shall

[stmt.label]

Para 1 change Labels cannot be re-declared to shall not be re-declared

[stmt.expr]

Para 1 ;last sentence it is useful...
Suggest make a footnote or overview

[stmt.block]
Last sentence make a footnote or overview

[stmt.if]
...The else ambiguity, does not describe or reference this

[stmt.break]

change may in first line to shall
The break statement shall occur only in an.....

[stmt.goto]
Change must to shall.

[stmt.dcl]
Para 5 must to shall

[dcl.link]
Is it necessary to specify C linkage before inclusion of C standard headers ?

[dcl.dcl]
Para 1 incomplete reference _temp.dcls_

Para 8 Generally speaking the....
delete "speaking the" leaving
Generally names declared by a declaration ....

[dcl.stc]

Para 1 change ....may appear in a given to ....shall appear in a give
Para 2 last sentence one use of auto ....
Suggest make a footnote or overview

[dcl.fct.spec]

Para 2 change must's to shall's

Also last sentence. A call to an inline function shall not precede.....

Para 4 The virtual specifier may be used only in declarations
change to shall be used only

[dcl.typedef]

Change may to shall

[dcl.type.cv]

Para 2 after example change must to shall.

[dcl.ref]

Para 3 Last sentence uses term valid object. How does a valid object relate to the objects and sub-objects described in[intro.memory] ?

[dcl.array]

Para 4 last sentence states that any of the expressions x3d, x3d[i], x3d[i][j], x3d[i][j][k] may reasonably appear in an expression.
(Refers to a static int x3d[3][5][7];)

In C, the Defect Report 17 Question 16 stated that for an array

int a[4][5];
the expression a[1][7] = 0; /* undefined */

Is the meaning of the above sentence limited to
x3d
x3d[0-2]
x3d[0-2][0-4],
x3d[0-2][0-4][0-6]

[class.mem] almost at [class.scope0]

Accessing a stored value
- a character type. the result is undefined.

suggest "undefined behavior" in line with definitions

[class.scope]

Change various uses of must to shall.

[class.access.dcl]

Para 3, change various use of may to shall

[class.temporary]

This point and other places uses the word compiler

Suggest in the intro somewhere say that "compiler" means any form of language translator (including "interpreter")

[class.dtor]

Para 7 typo , ...dealloation function

[class.dtor]

The draft is very quiet on the destruction of const objects. It tells us that we can invoke a destructor on a const object, but not whether we can expect a useful result.

In addition[basic.type.qualifier] states that the program may not change a const object.

What is the resolution of these two positions ?

[class.free]

Para 2 would be helpful to have a reference at the end of the sentence to the "usual rules"

[class.init]

I belive the earlier "boxed question" to the need for a definiton of non-trivial is answered by this section, a definition is needed.

[over.load]

Parameter declarations that differ only in presence or absence of const and/or volatile are equivalent.

Is this also true for register ?

Plum suggests prefacing the entire sub-paragraph with "Referring to the specifications in 8.3.5, note that ..." Also the previous sub-paragraph about * vs[]. Then after the "const" sub-paragraph also add "(Note also that 8.3.5 specifies that register is ignored in forming the type of a function.)"

[over.load]

typo in para 3 says "inter to T" not "pointer to T"

[over.match.call]

Para 1, ..Recall from 5.2.2

How about "As specified in 5.2.2, a ..."

[over.call.func]

Para 1, Of interest in this subclause...

Suggest chnage to: This subclause refers to only .....

[over.match.oper]
Para 7, space before comma after binary operator

[over.match.viable]

In C for a program to access variable arguments it has to include <stdarg.h> otherwise it is undefined behaviour. Does a C++ program have to include <stdarg.h> to allow overloading with a function that has an ellipsis in its parameter list ?

[temp.res]

Para 3 The requirement to diagnose errors at point of a template declaration or later is awkward. Surely a single requirement to diagnose by a certain point would be a more measurable requirement and would not stop early diagnoses.

[C++.predefined]

__STDC__ the least a C++ standard should do is to require __STDC__ not to be any of those specified in the C standard.
This will not stop vendors defining what they want but at least it does not endorse misleading behaviour.

[lib.definitions]

Reserved functions .. uses must

[lib.iterator.types]
[lib.allocator.types]
First para is commentary, should not be part of the standard

llib.container.types]

Para 4 past-the-end value should be in a different font.

[lib.exit]

What happens if one of the functions registered by calling atexit(f) is exit.

Does the statement "all" funtions registered with atexit(f), are called, in reverse order of their registration or the fact that exit does not return to its caller win out ?

``Amortized'' constant time has no operational definition.

(Example is in 17.2.2.2).

``Reentrant" has a variety of meanings. (Example is in 17.3.4.5.)

``Global scope" and ``file scope" are used interchangeably.
(Example is 17.3.3.1.2.)

Use of ``clause" and ``subclause" is often erroneous. If the number contains a dot, it's a subclause; otherwise it's a clause. (Example from 17.3: Subclauses 18 through 27 specify the requirements of individual entities within the library.)

Clause 17 frequently refers to definitions that apply ``in this clause," which should also apply to later library clauses. (Example from 17.2.4.4: Such cases are explicitly stated in this clause, and indicated by writing the required type name in constant-width italic characters.)

Some terms are left undefined. (Example from 17.1: A class member function (9.4), other than constructors, assignment, or destructor, that alters the ``state" of an object of the class.)

Many statements are non-normative. (Example from 17.2.2.3: One of the common problems in portability is to be able to encapsulate the information about the memory model. This information includes the knowledge of pointer types, the type of their difference, the type of the size of objects in this memory model, as well as the memory allocation and deallocation primitives for it.)

``Shall" sometimes applies to implementation, rather than to a conforming program. (Example from 17.3.1.1: All library entities shall be defined within the namespace std.)

Description sometimes lapses into first person plural. (Example from 17.2.2: In some cases we present the semantic requirements using C++ code.)

Use of Oxford comma is erratic. (Example from 17.2.2: Depending on the operations defined on them, there are five categories of iterators: input iterators, output iterators, forward iterators, bidirectional iterators and random access iterators, as shown in Table 15.

List of standard macros in Table 26 is missing entries. (Example: _IOFBF.)

Distinction between ``values" in Table 27 and macros in Table 26 is unclear. (Example: Why is LC_ALL a macro and CHAR_BIT a value?)

List of standard types in Table 28 is missing entries. (Example: time_t.)

List of standard template classes in Table 29 is missing entries. (Example: ptr_dyn_array.)

Table 33 lists ptr_dyn_array as a standard class. It's a standard template class.

Table 34 lists struct tm as defined in <cwchar>. It is not.
The list is also missing entries. (Example: div_t.)

Table 37 lists errno as a standard object. It's a macro.

Table 46 omits class type_info.

Table 48 list *_MIN_DIG instead of *_MIN_EXP (three occurrences).

Table 55 lists NDEBUG as a macro defined in <cassert>. It is not.

Table 56 lists errno as an object. It is not.

Subclause 17.3.3.5 lists three types as ``handler functions."
A function and a function type are distinct entities.

Subclause 18.5.2.2 omits `const string&' argument to constructor
for bad_typeid.

Subclause 21.1.1.3.11 says basic_string::put_at: `Throws out_of_range
if pos > len. Otherwise, if pos == len, the function replaces the string
controlled by *this with a string of length len + 1 whose first len
elements are a copy of the original string and whose remaining element
is initialized to c. Otherwise, the function assigns c to ptr[pos].'
The behavior for pos == len was specifically eliminated in a Mar 94
proposal, It should be treated the same as pos > len.

Subclause 27.1.2.5 says streamsize `is a synonym for one of the
signed basic integral types.' Various subtle problems occur if
streamsize does not have the same representation as int.

Subclause 27.1.3.1 should declare members precision and width
(and stored precision and width, and arguments to manipulators
setprecision and setw) streamsize, not int.

Subclause 27.1.3.1.9 says operator bool() `returns a non-null
pointer (whose value is otherwise unspecified) if failbit | badbit
is set in state.' It should say `returns false if ...' (It should
not say `returns true if ...')

Subclause 27.2.1.2 declares the member showmany. The accepted
proposal calls it showmanyc.

Subclause 27.2.2.1 declares readsome with int second argument and
return value. Both should be streamsize.

Subclause 27.2.2.1 says that on end of file a member function
`completes its actions and does setstate(eofbit) before returning'.
Whether actions are completed was left open in the accepted proposal.
Hence, this is a SUBSTANTIVE CHANGE that was NOT approved by the
committee. It is also probably undesirable.

Subclause 27.2.2.1 says that on end of file a member function
`does setstate(eofbit) before returning'. `Does' is colloquial --
`calls' is more precise. (A widespread problem.)

Subclause 27.2.2.1 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 sb.exception() rethrows
the exception without completing its actions. The clause `if badbit
is on in sb.exception()' is a SUBSTANTIVE CHANGE that was NOT approved
by the committee. It is also probably undesirable.

Subclause 27.2.2.1.5 says sync returns type int, but it must also

return a value of type eof(), which may not be convertible to int
and distinguishable from zero. Should probably say it returns a
nonzero value instead of eof().

Subclause 27.2.2.2 says bool `converts a signed short int', but the
`behaves as if' code says it extracts an int. Which is it?

Subclause 27.2.2.2 says, for the streambuf extractor: `an exception
occurs (in which case the exception is caught). setstate(badbit) is
not called.' The second sentence is not normative.

Subclause 27.2.2.2 says, for the streambuf extractor: `If the function
inserts no characters, it calls setstate(failbit). If failure was due
to catching an exception thrown while extracting characters from sb
and failbit is on in except then the caught exception is rethrown.'
The second sentence is a SUBSTANTIVE CHANGE that was NOT approved
by the committee. It is also probably undesirable.

Subclause 27.2.2.3.1 says get returns type int, but it must return
`character' values, plus a value of type eof(), which may not be
convertible to int without loss of information. Should return type T.

Subclauses 27.2.2.3.1, 27.2.2.3.2, 27.2.2.3.3, and 27.2.2.3.6
redundantly say when eofbit is set.

Subclause 27.2.2.3.6 says readsome sets eofbit if not all chars
read. It should also set failbit for consistency with read.

Subclause 27.2.2.3.6 says `If navail is 1.' It should say `If
navail is -1.'

Subclause 27.2.2.3.6 says that readsome `determines the number
of characters to extract m as the smaller of n and navail, and
returns read(s, m).' But read returns an istream&, for which no
conversion to int is defined. readsome should probably return
an istream&, always of value *this, as similar functions do.
(Or it should return gcount() in this situation.)

Subclause 27.2.4.1 says `If the called function throws an exception,
the output function calls setstate(badbit) and if badbit is on in
except rethrows the exception. The clause `if badbit is on in
except' is a SUBSTANTIVE CHANGE that was NOT approved by the committee.
It is also probably undesirable. This wording also differs in niggling
ways from 27.2.2.1, which suggests differences that should not be present.

Subclause 27.2.4.2.2 says, for the streambuf inserter: `If the function
inserts no characters or if it stopped because an exception was thrown
while extracting a character, it calls setstate(failbit).' The second
condition (after the `or') is a SUBSTANTIVE CHANGE that was NOT
approved by the committee. It is also probably undesirable.

Subclause 27.2.4.2.2 says, for the streambuf inserter: `If an exception
was thrown while extracting a character and failbit is on in excedptions
[sic] the caught exception is rethrown.' This sentence is a SUBSTANTIVE
CHANGE that was NOT approved by the committee. It is also probably
undesirable.

Subclause 27.3.1 should define basic_imanip in terms of basic_istream,
not basic_ios, to be consistent with the original accepted proposal.

Subclause 27.3.2 should define basic_omanip in terms of basic_ostream, not basic_ios, to be consistent with the original accepted proposal.

Six function signatures in <wchar.h> need the same double declaration that the six in <string.h> already have.

•